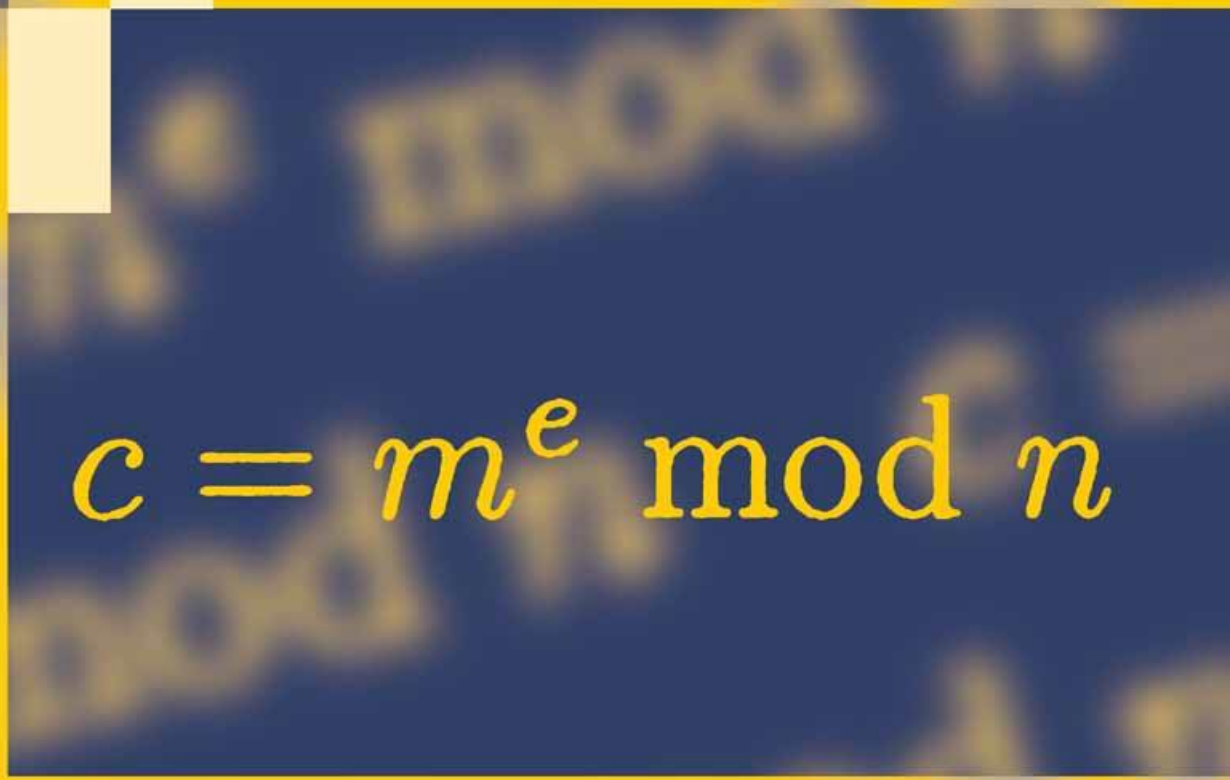
Buchmann

Einführung in die Kryptographie

4., erweiterte Auflage



Springer

Springer-Lehrbuch

Johannes Buchmann

Einführung in die Kryptographie

Vierte, erweiterte Auflage

Springer

Prof. Dr. Johannes Buchmann
Fachbereich Informatik
Technische Universität Darmstadt
Hochschulstraße 10
64289 Darmstadt
buchmann@cdc.informatik.tu-darmstadt.de

ISBN 978-3-540-74451-1 e-ISBN 978-3-540-74452-8

DOI 10.1007/978-3-540-74452-8

Springer-Lehrbuch ISSN 0937-7433

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Mathematics Subject Classification (2000): 94A60, 68P25, 11Y16

Satz: Datenerstellung durch den Autor unter Verwendung eines Springer TEX-Makropakets
Herstellung: LE-TEX Jelonek, Schmidt & Vöckler GbR, Leipzig
Umschlaggestaltung: WMX Design GmbH, Heidelberg

Gedruckt auf säurefreiem Papier

9 8 7 6 5 4 3 2 1

springer.de

Für Almut, Daniel und Jan

Vorwort zur vierten Auflage

In der vierten Auflage meiner Einführung in die Kryptographie habe ich auch diesmal den Stand der Forschung im Bereich Faktorisieren und Berechnung diskreter Logarithmen aktualisiert. Neu aufgenommen wurde das Merkle-Signaturverfahren. Dieses Verfahren wurde etwa zeitgleich mit dem RSA-Signaturverfahren erfunden. Nachdem Peter Shor gezeigt hat, dass Quantencomputer das Faktorisierungsproblem und die in der Kryptographie relevanten Diskrete-Logarithmen-Probleme in Polynomzeit lösen können, hat das Merkle-Verfahren neue Relevanz bekommen. Es stellt nämlich eine Alternative zu den heute verwendeten Signaturverfahren dar, die alle unsicher würden, wenn genügend große Quantencomputer gebaut werden können. Außerdem habe ich die Fehler, die mir seit Erscheinen der zweiten Auflage bekannt geworden sind, korrigiert. Für die vielen Hinweise, die ich von Lesern erhalten habe, bedanke ich mich sehr.

Darmstadt, im Dezember 2007 *Johannes Buchmann*

Vorwort zur dritten Auflage

In die dritte Auflage meiner Einführung in die Kryptographie habe ich Aktualisierungen und einige neue Inhalte aufgenommen. Aktualisiert wurde die Diskussion der Sicherheit von Verschlüsselungs- und Signaturverfahren und der Stand der Forschung im Bereich Faktorisieren und Berechnung diskreter Logarithmen. Neu aufgenommen wurde die Beschreibung des Advanced Encryption Standard (AES), des Secure Hash Algorithmus (SHA-1) und des Secret-Sharing-Verfahrens von Shamir. Außerdem habe ich die Fehler, die mir seit Erscheinen der zweiten Auflage bekannt geworden sind, korrigiert. Für die vielen Hinweise, die ich von Lesern erhalten habe, bedanke ich mich sehr.

Darmstadt, im Mai 2003 *Johannes Buchmann*

Vorwort zur zweiten Auflage

In die zweite Auflage meiner Einführung in die Kryptographie habe ich eine Reihe neuer Übungsaufgaben aufgenommen. Außerdem habe ich die Fehler, die mir seit Erscheinen der ersten Auflage bekannt geworden sind, korrigiert und einige Stellen aktualisiert. Für die vielen Hinweise, die ich von Lesern erhalten habe, bedanke ich mich sehr.

Darmstadt, im Dezember 2000 *Johannes Buchmann*

Vorwort

Kryptographie ist als Schlüsseltechnik für die Absicherung weltweiter Computernetze von zentraler Bedeutung. Moderne kryptographische Techniken werden dazu benutzt, Daten geheimzuhalten, Nachrichten elektronisch zu signieren, den Zugang zu Rechnernetzen zu kontrollieren, elektronische Geldgeschäfte abzusichern, Urheberrechte zu schützen usw. Angesichts dieser vielen zentralen Anwendungen ist es nötig, daß die Anwender einschätzen können, ob die benutzten kryptographischen Methoden effizient und sicher genug sind. Dazu müssen sie nicht nur wissen, wie die kryptographischen Verfahren funktionieren, sondern sie müssen auch deren mathematische Grundlagen verstehen.

Ich wende mich in diesem Buch an Leser, die moderne kryptographische Techniken und ihre mathematischen Fundamente kennenlernen wollen, aber nicht über die entsprechenden mathematischen Spezialkenntnisse verfügen. Mein Ziel ist es, in die Basistechniken der modernen Kryptographie einzuführen. Ich setze dabei zwar mathematische Vorbildung voraus, führe aber in die Grundlagen von linearer Algebra, Algebra, Zahlentheorie und Wahrscheinlichkeitstheorie ein, soweit diese Gebiete für die behandelten kryptographischen Verfahren relevant sind.

Das Buch ist aus einer Vorlesung entstanden, die ich seit 1996 in jedem Sommersemester an der Technischen Universität Darmstadt für Studenten der Informatik und Mathematik gehalten habe. Ich danke den Hörern dieser Vorlesung und den Mitarbeitern, die die Übungen betreut haben, für ihr Interesse und Engagement. Ich danke allen, die das Manuskript kritisch gelesen und verbessert haben. Besonders bedanke ich mich bei Harald Baier, Gabi Barking, Manuel Breuning, Safuat Hamdy, Birgit Henhapl, Andreas Kottig, Markus Maurer, Andreas Meyer, Stefan Neis, Sachar Paulus, Thomas Pfahler, Marita Skrobic, Tobias Straub, Edlyn Teske, Patrick Theobald und Ralf-Philipp Weinmann. Ich danke auch dem Springer-Verlag, besonders Martin Peters, Agnes Herrmann und Claudia Kehl, für die Unterstützung bei der Abfassung und Veröffentlichung dieses Buches.

Darmstadt, im Juli 1999 *Johannes Buchmann*

Inhaltsverzeichnis

1. Einleitung

Ursprünglich war die Kryptographie die Lehre von der Datenverschlüsselung. Verschlüsselung wird gebraucht, um *vertrauliche* Nachrichten auszutauschen oder vertrauliche Informationen sicher zu speichern. Techniken, die Vertraulichkeit ermöglichen, sind besonders im Zeitalter des Internet von großer Bedeutung. Aber moderne kryptographische Techniken dienen nicht nur der *Vertraulichkeit*. Im Laufe der letzten Jahrzehnte hat die Kryptographie eine Reihe weiterer Aufgaben bekommen. Die wichtigsten Schutzziele, die mit kryptographischen Mitteln erreicht werden sollen, sind *Vertraulichkeit*, *Authentizität*, *Integrität* und *Zurechenbarkeit*. In diesem einleitenden Kapitel beschreiben wir diese Sicherheitsziele und erläutern ihre Bedeutung. Danach wird die Struktur des Buches erkärt.

Kryptographische Methoden garantieren die *Vertraulichkeit* elektronischer Daten. Einige Beispiele, in denen Vertraulichkeit wichtig ist: Bankkunden informieren sich im Internet über ihren Kontostand. Firmen bereiten Strategieentscheidungen durch Email-Kommunikation vor. Ärzte speichern die Krankheitsgeschichte ihrer Patienten auf ihren Computern und übermitteln Untersuchungsergebnisse an die Krankenkassen. Gutachter protokollieren ihre Untersuchungsergebnisse auf ihren Laptops. All diese elektronischen Informationen müssen vertraulich bleiben. Um diese Vertraulichkeit zu gewährleisten, stellt die Kryptographie Verschlüsselungsverfahren zur Verfügung.

Kryptographische Techniken geben Aufschluss über die *Identität* des Absenders elektronischer Nachrichten und garantieren damit ihre *Authentizität*. Zum Beispiel darf der Webserver einer Fluggesellschaft nur autorisierten Nutzern Informationen über die Dienstpläne des fliegenden Personals geben, und für einen Internet-Broker ist es wichtig, sicher zu wissen, welcher Kunde Aktien bei ihm geordert hat.

Kryptographische Verfahren beweisen die *Integrität* elektronischer Daten, d.h. sie zeigen, dass Dateien oder Programme nicht verändert wurden. Zum Beispiel muss eine Bank festellen können, ob in einer elektronischen Überweisung der Betrag nachträglich verändert wurde. Wird ein Computer für sicherheitskritische Anwendungen benutzt wie zum Beispiel für die Steuerung von Hochgeschwindigkeitszügen, muss vor der Anwendung der entsprechenden Programme überprüft werden können, ob sie verändert wurden.

Schließlich machen kryptographische Methoden die elektronische Dokumente *zurechenbar*, d.h. sie machen es Dritten gegenüber beweisbar, dass ein Dokument von einem bestimmten Absender kommt. Wird über das Internet zum Beispiel ein Vertrag geschlossen, müssen alle Vertragspartner einem Richter gegenüber nachweisen können, dass der Vertrag tatsächlich geschlossen wurde, und wer die Vertragspartner sind. Diesen Zweck erfüllen *digitale Signaturen*.

In diesem Buch beschreibe ich wichtige kryptographische Verfahren, die Vertraulichkeit, Authentizität, Integrität und Zurechenbarkeit elektronischer Dokumente garantieren und diskutiere ihre Effizienz und Sicherheit.

Diese Verfahren sind mathematische *Algorithmen*, also Berechnungsverfahren. Ein Verschlüsselungsverfahren ist zum Beispiel ein Algorithmus, der aus einem Text, der verschlüsselt werden soll, und einem geheimen Schlüssel den verschlüsselten Text berechnet. Ein Signaturverfahren ist ein Algorithmus, der aus einem Text, der signiert werden soll, und einem geheimen Signaturschlüssel die digitale Signatur berechnet.

Um die Funktionsweise solcher kryptographischer Verfahren präzise beschreiben und begründen zu können, benötige ich grundlegende mathematische Begriffe und Sachverhalte besonders aus der Algebra und Zahlentheorie, die in den Kapiteln 2 und 3 bereitgestellt werden. Ich setze dabei keine besonderen Vorkenntnisse voraus. Leser, die solche Vorkenntnisse haben, können die entsprechenden Abschnitte überschlagen.

In Kapitel 4 bespreche ich Verschlüsselungsverfahren. Ich erläutere Blockchiffren und Stromchiffren. Ich erkläre, wie man Blockchiffren zur Verschlüsselung beliebig langer Dateien verwenden kann.

Die moderne Kryptographie stellt eine Vielzahl effizienter und sicherer Verfahren bereit, um diese Ziele zu erreichen.

2. Ganze Zahlen

Ganze Zahlen spielen eine fundamentale Rolle in der Kryptographie. In diesem Kapitel stellen wir grundlegende Eigenschaften der ganzen Zahlen zusammen und beschreiben fundamentale Algorithmen.

2.1 Grundlagen

Wir schreiben, wie üblich, $\mathbb{N} = \{1, 2, 3, 4, 5, \ldots\}$ für die *natürlichen Zahlen* und $\mathbb{Z} = \{0, \pm 1, \pm 2, \pm 3, \ldots\}$ für die *ganzen Zahlen*. Die rationalen Zahlen werden mit $\mathbb{Q}$ bezeichnet und die reellen Zahlen mit $\mathbb{R}$.

Es gilt $\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R}$. Reelle Zahlen (also auch natürliche, ganze und rationale Zahlen) kann man addieren und multiplizieren. Das wird als bekannt vorausgesetzt.

Außerdem werden folgende grundlegende Regeln benutzt:

Wenn das Produkt von zwei reellen Zahlen Null ist, dann ist wenigstens ein Faktor Null. Es kann also nicht sein, daß beide Faktoren von Null verschieden sind, aber das Produkt Null ist. Wir werden später sehen, daß es Zahlbereiche gibt, in denen das nicht gilt.

Reelle Zahlen kann man vergleichen. Z.B. ist $\sqrt{2}$ kleiner als 2, aber größer als 1. Wenn eine reelle Zahl α kleiner als eine andere reelle Zahl β ist, schreiben wir $\alpha < \beta$. Wenn α kleiner als β oder α gleich β ist, schreiben wir $\alpha \leq \beta$. Wenn α größer als β ist, schreiben wir $\alpha > \beta$. Wenn α größer als β oder α gleich β ist, schreiben wir $\alpha \geq \beta$. Ist γ eine weitere reelle Zahl, dann folgt aus $\alpha < \beta$ auch $\alpha + \gamma < \beta + \gamma$. Entsprechendes gilt für $\leq, >$ und $\geq$. Wenn $0 < \alpha$ und $0 < \beta$, dann folgt $0 < \alpha\beta$.

Eine Menge M von reellen Zahlen heißt *nach unten beschränkt*, wenn es eine reelle Zahl γ gibt, so daß alle Elemente von M größer als γ sind. Man sagt dann auch, daß M nach unten durch γ beschränkt ist. Die Menge der natürlichen Zahlen ist z.B. nach unten durch 0 beschränkt. Die Menge der geraden ganzen Zahlen ist aber nicht nach unten beschränkt. Eine wichtige Eigenschaft der ganzen Zahlen ist, daß jede nach unten beschränkte Menge ganzer Zahlen ein kleinstes Element besitzt. Z.B. ist die kleinste natürliche Zahl 1. Entsprechend definiert man nach oben beschränkte Mengen reeller Zahlen. Jede nach oben beschränkte Menge ganzer Zahlen hat ein größtes Element.

Für eine reelle Zahl α schreiben wir

$$\lfloor \alpha \rfloor = \max\{b \in \mathbb{Z} : b \leq \alpha\}.$$

Die Zahl $\lfloor \alpha \rfloor$ ist also die größte ganze Zahl, die kleiner als oder gleich α ist. Diese Zahl existiert, weil die Menge $\{b \in \mathbb{Z} : b \leq \alpha\}$ nach oben beschränkt ist.

Beispiel 2.1.1. Es ist $\lfloor 3.43 \rfloor = 3$ und $\lfloor -3.43 \rfloor = -4$.

Schließlich benötigen wir noch das Prinzip der *vollständigen Induktion*. Ist eine Aussage, in der eine unbestimmte natürliche Zahl n vorkommt, richtig für $n = 1$ und folgt aus ihrer Richtigkeit für alle natürlichen Zahlen m mit $1 \leq m \leq n$ (oder auch nur für n) ihre Richtigkeit für $n+1$, so ist die Aussage richtig für jede natürliche Zahl n.

In diesem Kapitel bezeichnen kleine lateinische Buchstaben ganze Zahlen.

2.2 Teilbarkeit

Definition 2.2.1. *Man sagt a* teilt *n, wenn es eine ganze Zahl b gibt mit $n = ab$.*

Wenn a die Zahl n teilt, dann heißt a *Teiler* von n und n *Vielfaches* von a und man schreibt $a \mid n$. Man sagt auch, n ist durch a *teilbar*. Wenn a kein Teiler von n ist, dann schreibt man $a \nmid n$.

Beispiel 2.2.2. Es gilt $13 \mid 182$, weil $182 = 14 * 13$ ist. Genauso gilt $-5 \mid 30$, weil $30 = (-6) * (-5)$ ist. Die Teiler von 30 sind ± 1, ± 2, ± 3, ± 5, ± 6, ± 10, ± 15, ± 30.

Jede ganze Zahl a teilt 0, weil $0 = a * 0$ ist. Die einzige ganze Zahl, die durch 0 teilbar ist, ist 0 selbst, weil aus $a = 0 * b$ folgt, daß $a = 0$ ist.

Wir beweisen einige einfache Regeln.

Theorem 2.2.3. *1. Aus $a \mid b$ und $b \mid c$ folgt $a \mid c$.*
2. Aus $a \mid b$ folgt $ac \mid bc$ für alle c.
3. Aus $c \mid a$ und $c \mid b$ folgt $c \mid da + eb$ für alle d und e.
4. Aus $a \mid b$ und $b \neq 0$ folgt $|a| \leq |b|$.
5. Aus $a \mid b$ und $b \mid a$ folgt $|a| = |b|$.

Beweis. 1. Wenn $a \mid b$ und $b \mid c$, dann gibt es f, g mit $b = af$ und $c = bg$. Also folgt $c = bg = (af)g = a(fg)$. 2. Wenn $a \mid b$, dann gibt es f mit $b = af$. Also folgt $bc = (af)c = f(ac)$. 3. Wenn $c \mid a$ und $c \mid b$, dann gibt es f, g mit $a = fc$ und $b = gc$. Also folgt $da + eb = dfc + egc = (df + eg)c$. 4. Wenn $a \mid b$ und $b \neq 0$, dann gibt es $f \neq 0$ mit $b = af$. Also ist $|b| = |af| \geq |a|$. 5. Gelte $a \mid b$ und $b \mid a$. Wenn $a = 0$, dann gilt $b = 0$ und umgekehrt. Wenn $a \neq 0$ und $b \neq 0$, dann folgt aus 4., daß $|a| \leq |b|$ und $|b| \leq |a|$, also $|a| = |b|$ gilt. □

Das folgende Ergebnis ist sehr wichtig. Es zeigt, daß Division mit Rest von ganzen Zahlen möglich ist.

Theorem 2.2.4. *Wenn a, b ganze Zahlen sind, $b > 0$, dann gibt es eindeutig bestimmte ganze Zahlen q und r derart, daß $a = qb + r$ und $0 \leq r < b$ ist, nämlich $q = \lfloor a/b \rfloor$ und $r = a - bq$.*

Beweis. Gelte $a = qb + r$ und $0 \leq r < b$. Dann folgt $0 \leq r/b = a/b - q < 1$. Dies impliziert $a/b - 1 < q \leq a/b$, also $q = \lfloor a/b \rfloor$. Umgekehrt erfüllen $q = \lfloor a/b \rfloor$ und $r = a - bq$ die Behauptung des Satzes. □

In der Situation von Theorem 2.2.4 nennt man q den (ganzzahligen) *Quotient* und r den *Rest* der Division von a durch b. Man schreibt $r = a \bmod b$. Wird a durch $a \bmod b$ ersetzt, so sagt man auch, daß a modulo b *reduziert* wird.

Beispiel 2.2.5. Wenn $a = 133$ und $b = 21$ ist, dann erhält man $q = 6$ und $r = 7$, d.h. $133 \bmod 21 = 7$. Entsprechend gilt $-50 \bmod 8 = 6$.

2.3 Darstellung ganzer Zahlen

In Büchern werden ganze Zahlen üblicherweise als Dezimalzahlen geschrieben. In Computern werden ganze Zahlen in Binärentwicklung gespeichert. Allgemein kann man ganze Zahlen mit Hilfe der sogenannten g-adischen Darstellung aufschreiben. Diese Darstellung wird jetzt beschrieben. Für eine natürliche Zahl $g > 1$ und eine positive reelle Zahl α bezeichnen wir mit $\log_g \alpha$ den Logarithmus zur Basis g von α. Für eine Menge M bezeichnet M^k die Menge aller Folgen der Länge k mit Gliedern aus M.

Beispiel 2.3.1. Es ist $\log_2 8 = 3$, weil $2^3 = 8$ ist. Ferner ist $\log_8 8 = 1$, weil $8^1 = 8$ ist.

Beispiel 2.3.2. Die Folge $(0, 1, 1, 1, 0)$ ist ein Element von $\{0, 1\}^5$. Ferner ist $\{1, 2\}^2 = \{(1, 1), (1, 2), (2, 1), (2, 2)\}$.

Theorem 2.3.3. *Sei g eine natürliche Zahl, $g > 1$. Für jede natürliche Zahl a gibt es eine eindeutig bestimmte natürliche Zahl k und eine eindeutig bestimmte Folge*

$$(a_1, \ldots, a_k) \in \{0, \ldots, g-1\}^k$$

mit $a_1 \neq 0$ und

$$a = \sum_{i=1}^{k} a_i g^{k-i}. \tag{2.1}$$

Dabei gilt $k = \lfloor \log_g a \rfloor + 1$, und a_i ist der ganzzahlige Quotient der Division von $a - \sum_{j=1}^{i-1} a_j g^{k-j}$ durch g^{k-i} für $1 \leq i \leq k$.

Beweis. Sei a eine natürliche Zahl. Wenn es eine Darstellung von a wie in (2.1) gibt, dann gilt $g^{k-1} \leq a = \sum_{i=1}^{k} a_i g^{k-i} \leq (g-1)\sum_{i=1}^{k} g^{k-i} = g^k - 1 < g^k$. Also ist $k = \lfloor \log_g a \rfloor + 1$. Dies beweist die Eindeutigkeit von k. Wir beweisen die Existenz und Eindeutigkeit der Folge $(a_1, \ldots, a_k)$ durch Induktion über k.

Für $k = 1$ setze $a_1 = a$. Dann ist (2.1) erfüllt, und eine andere Wahl für a_1 hat man nicht.

Sei $k > 1$. Wir beweisen zuerst die Eindeutigkeit. Wenn es eine Darstellung wie in (2.1) gibt, dann gilt $0 \leq a - a_1 g^{k-1} < g^{k-1}$ und daher $0 \leq a/g^{k-1} - a_1 < 1$. Damit ist a_1 der ganzzahlige Quotient der Division von a durch g^{k-1}, also eindeutig bestimmt. Setze $a' = a - a_1 g^{k-1} = \sum_{i=2}^{k} a_i g^{k-i}$. Entweder ist $a' = 0$. Dann ist $a_i = 0$, $2 \leq i \leq n$. Oder $a' = \sum_{i=2}^{k} a_i g^{k-i}$ ist die eindeutig bestimmte Darstellung von a' nach Induktionsannahme. Es ist jetzt auch klar, daß eine Darstellung (2.1) existiert. Man braucht nur $a_1 = \lfloor a/g^{k-1} \rfloor$ zu setzen und die anderen Koeffizienten aus der Darstellung von $a' = a - a_1 g^{k-1}$ zu nehmen.

Definition 2.3.4. *Die Folge* $(a_1, \ldots, a_k)$ *aus Theorem 2.3.3 heißt* g-adische Entwicklung *von* a. *Ihre Glieder heißen* Ziffer. *Ihre* Länge *ist* $k = \lfloor \log_g a \rfloor + 1$. *Falls* $g = 2$ *ist, heißt diese Folge* Binärentwicklung *von* a . *Falls* $g = 16$ *ist, heißt die Folge* Hexadezimalentwicklung *von* a.

Die g-adische Entwicklung einer natürlichen Zahl ist nur dann eindeutig, wenn man verlangt, daß die erste Ziffer von Null verschieden ist. Statt $(a_1, \ldots, a_k)$ schreibt man auch $a_1 a_2 \ldots a_k$.

Beispiel 2.3.5. Die Folge `10101` ist die Binärentwicklung der Zahl $2^4 + 2^2 + 2^0 = 21$. Wenn man Hexadezimaldarstellungen aufschreibt, verwendet man für die Ziffern $10, 11, \ldots, 15$ die Buchstaben A, B, C, D, E, F. So ist `A1C` die Hexadezimaldarstellung von $10 * 16^2 + 16 + 12 = 2588$.

Theorem 2.3.3 enthält ein Verfahren zur Berechnung der g-adischen Entwicklung einer natürlichen Zahl. Das wird im nächsten Beispiel angewandt.

Beispiel 2.3.6. Wir bestimmen die Binärentwicklung von 105. Da $64 = 2^6 < 105 < 128 = 2^7$ ist, hat sie die Länge 7. Wir erhalten: $a_1 = \lfloor 105/64 \rfloor = 1$. $105 - 64 = 41$. $a_2 = \lfloor 41/32 \rfloor = 1$. $41 - 32 = 9$. $a_3 = \lfloor 9/16 \rfloor = 0$. $a_4 = \lfloor 9/8 \rfloor = 1$. $9 - 8 = 1$. $a_5 = a_6 = 0$. $a_7 = 1$. Also ist die Binärentwicklung von 105 die Folge 1101001.

Die Umwandlung von Hexadezimalentwicklungen in Binärentwicklungen und umgekehrt ist besonders einfach. Sei $(h_1, h_2, \ldots, h_k)$ die Hexadezimalentwicklung einer natürlichen Zahl n. Für $1 \leq i \leq k$ sei $(b_{1,i}, b_{2,i}, b_{3,i}, b_{4,i})$ der Bitstring der Länge 4, der h_i darstellt, also $h_i = b_{1,i}2^3 + b_{2,i}2^2 + b_{3,i}2 + b_{4,i}$, dann ist $(b_{1,1}, b_{2,1}, b_{3,1}, b_{4,1}, b_{1,2}, \ldots, b_{4,k})$ die Binärentwicklung von n.

Beispiel 2.3.7. Betrachte die Hexadezimalzahl $n = 6EF$. Die auf Länge 4 normierten Binärentwicklungen der Ziffern sind $6 = 0110$, $E = 1110$, $F = 1111$. Daher ist 011011101111 die Binärentwicklung von n.

Die Länge der Binärentwicklung einer natürlichen Zahl wird auch als ihre *binäre Länge* bezeichnet. Die binäre Länge von 0 wird auf 1 gesetzt. Die binäre Länge einer ganzen Zahl ist die binäre Länge ihres Absolutbetrags. Die binäre Länge einer ganzen Zahl a wird auch mit $\mathrm{size}(a)$ oder $\mathrm{size}\, a$ bezeichnet.

2.4 O- und Ω-Notation

Beim Design eines kryptographischen Algorithmus ist es nötig, abzuschätzen, welchen Berechnungsaufwand er hat und welchen Speicherplatz er benötigt. Um solche Aufwandsabschätzungen zu vereinfachen, ist es nützlich, die O- und die Ω-Notation zu verwenden.

Seien k eine natürliche Zahl, $X, Y \subset \mathbb{N}^k$ und $f : X \to \mathbb{R}$, $g : Y \to \mathbb{R}$ Funktionen. Man schreibt $f = O(g)$, falls es positive reelle Zahlen B und C gibt derart, daß für alle $(n_1, \ldots, n_k) \in \mathbb{N}^k$ mit $n_i > B$, $1 \le i \le k$, folgendes gilt:

1. $(n_1, \ldots, n_k) \in X \cap Y$, d.h. $f(n_1, \ldots, n_k)$ und $g(n_1, \ldots, n_k)$ sind definiert,
2. $f(n_1, \ldots, n_k) \le Cg(n_1, \ldots, n_k)$.

Das bedeutet, daß fast überall $f(n_1, \ldots, n_k) \le Cg(n_1, \ldots, n_k)$ gilt. Man schreibt dann auch $g = \Omega(f)$. Ist g eine Konstante, so schreibt man $f = O(1)$.

Beispiel 2.4.1. Es ist $2n^2 + n + 1 = O(n^2)$, weil $2n^2 + n + 1 \le 4n^2$ ist für alle $n \ge 1$. Außerdem ist $2n^2 + n + 1 = \Omega(n^2)$, weil $2n^2 + n + 1 \ge 2n^2$ ist für alle $n \ge 1$.

Beispiel 2.4.2. Ist g eine natürliche Zahl, $g > 2$ und bezeichnet $f(n)$ die Länge der g-adischen Entwicklung einer natürlichen Zahl n, so gilt $f(n) = O(\log n)$, wobei $\log n$ der natürliche Logarithmus von n ist. Diese Länge ist nämlich $\lfloor \log_g n \rfloor + 1 \le \log_g n + 1 = \log n / \log g + 1$. Für $n > 3$ ist $\log n > 1$ und daher ist $\log n / \log g + 1 < (1/\log g + 1) \log n$.

2.5 Aufwand von Addition, Multiplikation und Division mit Rest

In vielen kryptographischen Verfahren werden lange ganze Zahlen addiert, multipliziert und mit Rest dividiert. Um die Laufzeit solcher Verfahren abschätzen zu können, muß man untersuchen, wie lange diese Operationen brauchen. Man legt dazu ein Rechenmodell fest, das den tatsächlichen Computern möglichst ähnlich ist. Dies wird sehr sorgfältig und ausführlich in [3]

und [4] gemacht. Hier wird nur ein naives Modell beschrieben, das aber eine gute Abschätzung für die benötigte Rechenzeit liefert.

Es seien a und b natürliche Zahlen, die durch ihre Binärentwicklungen gegeben seien. Die binäre Länge von a sei m und die binäre Länge von b sei n. Um $a+b$ zu berechnen, schreibt man die Binärentwicklungen von a und b untereinander und addiert Bit für Bit mit Übertrag.

Beispiel 2.5.1. Sei $a = 10101$, $b = 111$. Wir berechnen $a + b$.

	1	0	1	0	1
+			1	1	1
Übertrag		1	1	1	
	1	1	1	0	0

Wir nehmen an, daß die Addition von zwei Bits Zeit $O(1)$ braucht. Dann braucht die gesamte Addition Zeit $O(\max\{m,n\})$. Entsprechend zeigt man, daß man b von a in Zeit $O(\max\{m,n\})$ subtrahieren kann. Daraus folgt, daß die Addition zweier ganzer Zahlen a und b mit Binärlänge m und n Zeit $O(\max\{m,n\})$ kostet.

Auch bei der Multiplikation gehen wir ähnlich vor wie in der Schule.

Beispiel 2.5.2. Sei $a = 10101$, $b = 101$. Wir berechnen $a * b$.

	1	0	1	0	1	*	1	0	1
					1	0	1	0	1
+			1	0	1	0	1		
Übertrag				1		1			
			1	1	0	1	0	0	1

Man geht b von hinten nach vorn durch. Für jede 1 schreibt man a auf und zwar so, daß das am weitesten rechts stehende Bit von a unter der 1 von b steht. Dann addiert man dieses a zu dem vorigen Ergebnis. Jede solche Addition kostet Zeit $O(m)$ und es gibt höchstens $O(n)$ Additionen. Die Berechnung kostet also Zeit $O(mn)$. In [3] wird die Methode von Schönhage und Strassen erläutert, die zwei n-Bit-Zahlen in Zeit $O(n \log n \log\log n)$ multipliziert. In der Praxis ist diese Methode für Zahlen, die eine kürzere binäre Länge als 10000 haben, aber langsamer als die Schulmethode.

Um a durch b mit Rest zu dividieren, verwendet man ebenfalls die Schulmethode.

Beispiel 2.5.3. Sei $a = 10101$, $b = 101$. Wir dividieren a mit Rest durch b.

1	0	1	0	1	=	1	0	1	*	1	0	0	+	1
1	0	1												
	0	0	0											
	0	0	0											
		0	0	1										
		0	0	0										
				1										

Analysiert man diesen Algorithmus, stellt man folgendes fest: Sei k die Anzahl der Bits des Quotienten. Dann muß man höchstens k-mal zwei Zahlen mit binärer Länge $\leq n+1$ voneinander abziehen. Dies kostet Zeit $O(kn)$.

Zusammenfassend erhalten wir folgende Schranken, die wir in Zukunft benutzen wollen.

1. Die Addition von a und b erfordert Zeit $O(\max\{\operatorname{size} a, \operatorname{size} b\})$.
2. Die Multiplikation von a und b erfordert Zeit $O((\operatorname{size} a)(\operatorname{size} b))$.
3. Die Division mit Rest von a durch b erfordert Zeit $O((\operatorname{size} b)(\operatorname{size} q))$, wobei q der Quotient ist.

Der benötigte Platz ist $O(\operatorname{size} a + \operatorname{size} b)$.

2.6 Polynomzeit

Bei der Analyse eines kryptographischen Verfahrens muß man zeigen, daß das Verfahren in der Praxis effizient funktioniert, aber nicht effizient gebrochen werden kann. Wir präzisieren den Begriff "effizient".

Angenommen, ein Algorithmus bekommt als Eingabe ganze Zahlen z_1, ..., z_n. Man sagt, daß dieser Algorithmus *polynomielle Laufzeit* hat, wenn es nicht negative ganze Zahlen $e_1, \ldots, e_n$ gibt, so daß der Algorithmus die Laufzeit

$$O((\operatorname{size} z_1)^{e_1} (\operatorname{size} z_2)^{e_2} \cdots (\operatorname{size} z_n)^{e_n})$$

hat. Der Algorithmus gilt als effizient, wenn er polynomielle Laufzeit hat. Man muß allerdings beachten, daß der Algorithmus nur dann als praktisch effizient gelten kann, wenn die O-Konstanten und die Exponenten e_i klein sind.

2.7 Größter gemeinsamer Teiler

Wir führen den größten gemeinsamen Teiler zweier ganzer Zahlen ein.

Definition 2.7.1. *Ein* gemeinsamer Teiler *von a und b ist eine ganze Zahl c, die sowohl a als auch b teilt.*

Theorem 2.7.2. *Unter allen gemeinsamen Teilern zweier ganzer Zahlen a und b, die nicht beide gleich 0 sind, gibt es genau einen (bezüglich $\leq$) größten. Dieser heißt* größter gemeinsamer Teiler (ggT) *von a und b und wird mit* $\gcd(a, b)$ *bezeichnet. Die Abkürzung gcd steht für greatest common divisor.*

Beweis. Sei $a \neq 0$. Nach Theorem 2.2.3 sind alle Teiler von a durch $|a|$ beschränkt. Daher muß es unter allen Teilern von a und damit unter allen gemeinsamen Teilern von a und b einen größten geben. □

Der Vollständigkeit halber wird der größte gemeinsame Teiler von 0 und 0 auf 0 gesetzt, also $\gcd(0,0) = 0$. Der größte gemeinsame Teiler zweier ganzer Zahlen ist also nie negativ.

Beispiel 2.7.3. Der größte gemeinsame Teiler von 18 und 30 ist 6. Der größte gemeinsame Teiler von -10 und 20 ist 10. Der größte gemeinsame Teiler von -20 und -14 ist 2. Der größte gemeinsame Teiler von 12 und 0 ist 12.

Der größte gemeinsame Teiler von ganzen Zahlen $a_1, \ldots, a_k$, $k \geq 1$, wird entsprechend definiert: Ist wenigstens eine der Zahlen a_i von Null verschieden, so ist $\gcd(a_1, \ldots, a_k)$ die größte natürliche Zahl, die alle a_i teilt. Sind alle a_i gleich 0, so wird $\gcd(a_1, \ldots, a_k) = 0$ gesetzt.

Wir geben als nächstes eine besondere Darstellung des größten gemeinsamen Teilers an. Dazu brauchen wir eine Bezeichnung.

Sind $\alpha_1, \ldots, \alpha_k$ reelle Zahlen, so schreibt man

$$\alpha_1\mathbb{Z} + \ldots + \alpha_k\mathbb{Z} = \{\alpha_1 z_1 + \ldots + \alpha_k z_k : z_i \in \mathbb{Z}, 1 \leq i \leq k\}.$$

Dies ist die Menge aller *ganzzahligen Linearkombinationen* der α_i.

Beispiel 2.7.4. Die Menge der ganzzahligen Linearkombinationen von 3 und 4 ist $3\mathbb{Z} + 4\mathbb{Z}$. Sie enthält die Zahl $1 = 3 * (-1) + 4$. Sie enthält auch alle ganzzahligen Vielfachen von 1. Also ist diese Menge gleich $\mathbb{Z}$.

Der nächste Satz zeigt, daß das Ergebnis des vorigen Beispiels kein Zufall ist.

Theorem 2.7.5. *Die Menge aller ganzzahligen Linearkombinationen von a und b ist die Menge aller ganzzahligen Vielfachen von* $\gcd(a,b)$*, also*

$$a\mathbb{Z} + b\mathbb{Z} = \gcd(a,b)\mathbb{Z}.$$

Beweis. Für $a = b = 0$ ist die Behauptung offensichtlich korrekt. Also sei angenommen, daß a oder b nicht 0 ist.

Setze

$$I = a\mathbb{Z} + b\mathbb{Z}.$$

Sei g die kleinste positive ganze Zahl in I. Wir behaupten, daß $I = g\mathbb{Z}$ gilt. Um dies einzusehen, wähle ein von Null verschiedenes Element c in I. Wir müssen zeigen, daß $c = qg$ für ein q gilt. Nach Theorem 2.2.4 gibt es q, r mit $c = qg + r$ und $0 \leq r < g$. Also gehört $r = c - qg$ zu I. Da aber g die kleinste positive Zahl in I ist, muß $r = 0$ und $c = qg$ gelten.

Es bleibt zu zeigen, daß $g = \gcd(a,b)$ gilt. Da $a, b \in I$ ist, folgt aus $I = g\mathbb{Z}$, daß g ein gemeinsamer Teiler von a und b ist. Da ferner $g \in I$ ist, gibt es x, y mit $g = xa + yb$. Ist also d ein gemeinsamer Teiler von a und b, dann ist d auch ein Teiler von g. Daher impliziert Theorem 2.2.3, daß $|d| \leq g$ gilt. Damit ist g der größte gemeinsame Teiler von a und b. □

Das Ergebnis von Beispiel 2.7.4 hätte man direkt aus Theorem 2.7.5 folgern können. Es ist nämlich $\gcd(3,4) = 1$ und daher $3\mathbb{Z} + 4\mathbb{Z} = 1\mathbb{Z} = \mathbb{Z}$.

Theorem 2.7.5 hat einige wichtige Folgerungen.

Korollar 2.7.6. *Für alle a, b, n ist die Gleichung $ax + by = n$ genau dann durch ganze Zahlen x und y lösbar, wenn $\gcd(a,b)$ ein Teiler von n ist.*

Beweis. Gibt es ganze Zahlen x und y mit $n = ax + by$, dann gehört n zu $a\mathbb{Z} + b\mathbb{Z}$ und nach Theorem 2.7.5 damit auch zu $\gcd(a,b)\mathbb{Z}$. Man kann also $n = c\gcd(a,b)$ schreiben, und das bedeutet, daß n ein Vielfaches von $\gcd(a,b)$ ist.

Ist umgekehrt n ein Vielfaches von $\gcd(a,b)$, dann gehört n zu der Menge $\gcd(a,b)\mathbb{Z}$. Nach Theorem 2.7.5 gehört n also auch zu $a\mathbb{Z}+b\mathbb{Z}$. Es gibt daher ganze Zahlen x und y mit $n = ax + by$. □

Korollar 2.7.6 sagt uns, daß die Gleichung

$$3x + 4y = 123$$

eine Lösung hat, weil $\gcd(3,4) = 1$ ist und 123 ein Vielfaches von 1 ist. Wir kennen aber noch keine effiziente Methode, um eine Lösung x und y zu berechnen. Man kann das mit dem euklidischen Algorithmus machen, der im nächsten Abschnitt erklärt wird.

Korollar 2.7.7. *Es gibt ganze Zahlen x und y mit $ax + by = \gcd(a,b)$.*

Beweis. Weil $\gcd(a,b)$ ein Teiler von sich selbst ist, folgt die Behauptung unmittelbar aus Korollar 2.7.6. □

Wir geben noch eine andere nützliche Charakterisierung des größten gemeinsamen Teilers an. Diese Charakterisierung wird auch häufig als Definition des größten gemeinsamen Teilers verwendet.

Korollar 2.7.8. *Es gibt genau einen nicht negativen gemeinsamen Teiler von a und b, der von allen gemeinsamen Teilern von a und b geteilt wird. Dieser ist der größte gemeinsame Teiler von a und b.*

Beweis. Der größte gemeinsame Teiler von a und b ist ein nicht negativer gemeinsamer Teiler von a und b. Außerdem gibt es nach Korollar 2.7.7 ganze Zahlen x und y mit $ax + by = \gcd(a,b)$. Daher ist jeder gemeinsame Teiler von a und b auch ein Teiler von $\gcd(a,b)$. Damit ist gezeigt, daß es einen nicht negativen gemeinsamen Teiler von a und b gibt, der von allen gemeinsamen Teilern von a und b geteilt wird.

Sei umgekehrt g ein nicht negativer gemeinsamer Teiler von a und b, der von jedem gemeinsamen Teiler von a und b geteilt wird. Ist $a = b = 0$, so ist $g = 0$, weil nur 0 von 0 geteilt wird. Ist a oder b von Null verschieden, dann ist nach Theorem 2.2.3 jeder gemeinsame Teiler von a und b kleiner oder gleich g. Damit ist $g = \gcd(a,b)$. □

Es bleibt die Frage, wie $\gcd(a, b)$ berechnet wird und wie ganze Zahlen x und y bestimmt werden, die $ax + by = \gcd(a, b)$ erfüllen. Der Umstand, daß diese beiden Probleme effiziente Lösungen besitzen, ist zentral für fast alle kryptographische Techniken.

Beide Probleme werden mit dem euklidischen Algorithmus gelöst, der im nächsten Abschnitt erläutert wird.

2.8 Euklidischer Algorithmus

Der euklidische Algorithmus berechnet den größten gemeinsamen Teiler zweier natürlicher Zahlen sehr effizient. Er beruht auf folgendem Satz:

Theorem 2.8.1. *1. Wenn $b = 0$ ist, dann ist $\gcd(a, b) = |a|$.*
2. Wenn $b \neq 0$ ist, dann ist $\gcd(a, b) = \gcd(|b|, a \bmod |b|)$.

Beweis. Die erste Behauptung ist offensichtlich korrekt. Wir beweisen die zweite. Sei $b \neq 0$. Nach Theorem 2.2.4 gibt es eine ganze Zahl q mit $a = q|b| + (a \bmod |b|)$. Daher teilt der größte gemeinsame Teiler von a und b auch den größten gemeinsamen Teiler von $|b|$ und $a \bmod |b|$ und umgekehrt. Da beide größte gemeinsame Teiler nicht negativ sind, folgt die Behauptung aus Theorem 2.2.3. □

Wir erläutern den euklidischen Algorithmus erst an einem Beispiel.

Beispiel 2.8.2. Wir möchten $\gcd(100, 35)$ berechnen. Nach Theorem 2.8.1 erhalten wir $\gcd(100, 35) = \gcd(35, 100 \bmod 35) = \gcd(35, 30) = \gcd(30, 5) = \gcd(5, 0) = 5$.

Zuerst ersetzt der euklidische Algorithmus a durch $|a|$ und b durch $|b|$. Dies hat in unserem Beispiel keinen Effekt. Solange b nicht Null ist, ersetzt der Algorithmus a durch b und b durch $a \bmod b$. Sobald $b = 0$ ist, wird a zurückgegeben. In Abbildung 2.1 ist der euklidische Algorithmus im Pseudocode dargestellt.

Theorem 2.8.3. *Der euklidische Algorithmus berechnet den größten gemeinsamen Teiler von a und b.*

Beweis. Um zu beweisen, daß der euklidische Algorithmus abbricht und dann tatsächlich den größten gemeinsamen Teiler von a und b zurückgibt, führen wir folgende Bezeichnungen ein: Wir setzen

$$r_0 = |a|, r_1 = |b| \tag{2.2}$$

und für $k \geq 1$ und $r_k \neq 0$

$$r_{k+1} = r_{k-1} \bmod r_k. \tag{2.3}$$

```
euclid(int a, int b, int gcd)
begin
  int r
  a = |a|
  b = |b|
  while (b != 0)
    r = a%b
    a = b
    b = r
  end while
  gcd = a
end
```

Abb. 2.1. Der euklidische Algorithmus

Dann ist $r_2, r_3, \ldots$ die Folge der Reste, die in der `while`-Schleife des euklidischen Algorithmus ausgerechnet wird. Außerdem gilt nach dem k-ten Durchlauf der `while`-Schleife im euklidischen Algorithmus

$$a = r_k, \quad b = r_{k+1}.$$

Aus Theorem 2.8.1 folgt, daß sich der größte gemeinsame Teiler von a und b nicht ändert. Um zu zeigen, daß der euklidische Algorithmus tatsächlich den größten gemeinsamen Teiler von a und b berechnet, brauchen wir also nur zu beweisen, daß ein r_k schließlich 0 ist. Das folgt aber daraus, daß nach (2.3) die Folge $(r_k)_{k\geq 1}$ streng monoton fallend ist. Damit ist die Korrektheit des euklidischen Algorithmus bewiesen. □

Der euklidische Algorithmus berechnet $\gcd(a, b)$ sehr effizient. Das ist wichtig für kryptographische Anwendungen. Um dies zu beweisen, wird die Anzahl der Iterationen im euklidischen Algorithmus abgeschätzt. Dabei wird der euklidische Algorithmus Schritt für Schritt untersucht. Zur Vereinfachung nehmen wir an, daß

$$a > b > 0$$

ist. Dies ist keine Einschränkung, weil der euklidische Algorithmus einen Schritt braucht, um $\gcd(a, b)$ zu bestimmen (wenn $b = 0$ ist) oder diese Situation herzustellen.

Sei r_n das letzte von Null verschiedene Glied der Restefolge (r_k). Dann ist n die Anzahl der Iterationen, die der euklidische Algorithmus braucht, um $\gcd(a, b)$ auszurechnen. Sei weiter

$$q_k = \lfloor r_{k-1}/r_k \rfloor, \quad 1 \leq k \leq n. \tag{2.4}$$

Die Zahl q_k ist also der Quotient der Division von r_{k-1} durch r_k und es gilt

$$r_{k-1} = q_k r_k + r_{k+1}. \tag{2.5}$$

Beispiel 2.8.4. Ist $a = 100$ und $b = 35$, dann erhält man die Folge

k	0	1	2	3	4
r_k	100	35	30	5	0
q_k		2	1	6	

Um die Anzahl n der Iterationen des euklidischen Algorithmus abzuschätzen, beweisen wir folgendes Hilfsresultat. Hierin ist $a > b > 0$ vorausgesetzt.

Lemma 2.8.5. *Es gilt $q_k \geq 1$ für $1 \leq k \leq n-1$ und $q_n \geq 2$.*

Beweis. Da $r_{k-1} > r_k > r_{k+1}$ gilt, folgt aus (2.5), daß $q_k \geq 1$ ist für $1 \leq k \leq n$. Angenommen, $q_n = 1$. Dann folgt $r_{n-1} = r_n$ und das ist nicht möglich, weil die Restefolge streng monoton fällt. Daher ist $q_n \geq 2$. □

Theorem 2.8.6. *Im euklidischen Algorithmus sei $a > b > 0$. Setze $\Theta = (1+\sqrt{5})/2$. Dann ist die Anzahl der Iterationen im euklidischen Algorithmus höchstens $(\log b)/(\log \Theta) + 1 < 1.441 * \log_2(b) + 1$.*

Beweis. Nach Übung 2.12.19 können wir annehmen, daß $\gcd(a, b) = r_n = 1$ ist. Durch Induktion wird bewiesen, daß

$$r_k \geq \Theta^{n-k}, \quad 0 \leq k \leq n \tag{2.6}$$

gilt. Dann ist insbesondere

$$b = r_1 \geq \Theta^{n-1}.$$

Durch Logarithmieren erhält man daraus

$$n \leq (\log b)/(\log \Theta) + 1,$$

wie behauptet.

Wir beweisen nun (2.6). Zunächst gilt

$$r_n = 1 = \Theta^0$$

und nach Lemma 2.8.5

$$r_{n-1} = q_n r_n = q_n \geq 2 > \Theta.$$

Sei $n - 2 \geq k \geq 0$ und gelte die Behauptung für $k' > k$. Dann folgt aus Lemma 2.8.5

$$\begin{aligned} r_k &= q_{k+1} r_{k+1} + r_{k+2} \geq r_{k+1} + r_{k+2} \\ &\geq \Theta^{n-k-1} + \Theta^{n-k-2} = \Theta^{n-k-1}\left(1 + \frac{1}{\Theta}\right) = \Theta^{n-k}. \end{aligned}$$

Damit sind (2.6) und das Theorem bewiesen. □

2.9 Erweiterter euklidischer Algorithmus

Im vorigen Abschnitt haben wir gesehen, wie man den größten gemeinsamen Teiler zweier ganzer Zahlen berechnen kann. In Korollar 2.7.7 wurde gezeigt, daß es ganze Zahlen x, y gibt, so daß $\gcd(a, b) = ax + by$ ist. In diesem Abschnitt erweitern wir den euklidischen Algorithmus so, daß er solche Koeffizienten x und y berechnet. Wie in Abschnitt 2.8 bezeichnen wir mit $r_0, \ldots, r_{n+1}$ die Restefolge und mit $q_1, \ldots, q_n$ die Folge der Quotienten, die bei der Anwendung des euklidischen Algorithmus auf a, b entstehen.

Wir erläutern nun die Konstruktion zweier Folgen (x_k) und (y_k), für die $x = (-1)^n x_n$ und $y = (-1)^{n+1} y_n$ die gewünschte Eigenschaft haben.

Wir setzen

$$x_0 = 1, x_1 = 0, y_0 = 0, y_1 = 1.$$

Ferner setzen wir

$$x_{k+1} = q_k x_k + x_{k-1}, \quad y_{k+1} = q_k y_k + y_{k-1}, \quad 1 \leq k \leq n. \tag{2.7}$$

Wir nehmen an, daß a und b nicht negativ sind.

Theorem 2.9.1. *Es gilt* $r_k = (-1)^k x_k a + (-1)^{k+1} y_k b$ *für* $0 \leq k \leq n + 1$.

Beweis. Es ist

$$r_0 = a = 1 * a - 0 * b = x_0 * a - y_0 * b.$$

Weiter ist

$$r_1 = b = -0 * a + 1 * b = -x_1 * a + y_1 * b.$$

Sei nun $k \geq 2$ und gelte die Behauptung für alle $k' < k$. Dann ist

$$\begin{aligned} r_k &= r_{k-2} - q_{k-1} r_{k-1} \\ &= (-1)^{k-2} x_{k-2} a + (-1)^{k-1} y_{k-2} b - q_{k-1}((-1)^{k-1} x_{k-1} a + (-1)^k y_{k-1} b) \\ &= (-1)^k a(x_{k-2} + q_{k-1} x_{k-1}) + (-1)^{k+1} b(y_{k-2} + q_{k-1} y_{k-1}) \\ &= (-1)^k x_k a + (-1)^{k+1} y_k b. \end{aligned}$$

Damit ist das Theorem bewiesen. □

Man sieht, daß insbesondere

$$r_n = (-1)^n x_n a + (-1)^{n+1} y_n b$$

ist. Damit ist also der größte gemeinsame Teiler von a und b als Linearkombination von a und b dargestellt.

Beispiel 2.9.2. Wähle $a = 100$ und $b = 35$. Dann kann man die Werte r_k, q_k, x_k und y_k aus folgender Tabelle entnehmen.

k	0	1	2	3	4
r_k	100	35	30	5	0
q_k		2	1	6	
x_k	1	0	1	1	7
y_k	0	1	2	3	20

Damit ist $n = 3$ und $\gcd(100, 35) = 5 = -1 * 100 + 3 * 35$.

Der erweiterte euklidische Algorithmus berechnet neben $\gcd(a, b)$ auch die Koeffizienten

$$x = (-1)^n x_n \quad y = (-1)^{n+1} y_n$$

Den erweiterten euklidischen Algorithmus findet man in Abbildung 2.2.

Die Korrektheit dieses Algorithmus folgt aus Theorem 2.9.1.

2.10 Analyse des erweiterten euklidischen Algorithmus

Als erstes werden wir die Größe der Koeffizienten x und y abschätzen, die der erweiterte euklidische Algorithmus berechnet. Das ist wichtig dafür, daß der erweiterte euklidische Algorithmus von Anwendungen effizient benutzt werden kann.

Wir brauchen die Matrizen

$$E_k = \begin{pmatrix} q_k & 1 \\ 1 & 0 \end{pmatrix}, \quad 1 \leq k \leq n,$$

und

$$T_k = \begin{pmatrix} y_k & y_{k-1} \\ x_k & x_{k-1} \end{pmatrix}, \quad 1 \leq k \leq n+1.$$

Es gilt

$$T_{k+1} = T_k E_k, \quad 1 \leq k \leq n$$

und da T_1 die Einheitsmatrix ist, folgt

$$T_{n+1} = E_1 E_2 \cdots E_n.$$

Setzt man nun

$$S_k = E_{k+1} E_{k+2} \cdots E_n, \quad 0 \leq k \leq n,$$

wobei S_n die Einheitsmatrix ist, so gilt

$$S_0 = T_{n+1}.$$

Wir benutzen die Matrizen S_k, um die Zahlen x_n und y_n abzuschätzen. Schreibt man

$$S_k = \begin{pmatrix} u_k & v_k \\ u_{k+1} & v_{k+1} \end{pmatrix}, \quad 0 \leq k \leq n,$$

```
xeuclid(int a, int b,int gcd, int x, int y) {
begin

  int q, r, xx, yy, sign
  int xs[2], ys[2]

  // Die Koeffizienten werden initialisiert.

  xs[0] = 1 xs[1] = 0
  ys[0] = 0 ys[1] = 1
  sign = 1

  // Solange b != 0 ist, wird a durch b und b durch a%b
  // ersetzt.
  // Ausserdem werden die Koeffizienten neu berechnet.

  while (b != 0)
    r = a%b
    q = a/b
    a = b
    b = r
    xx = xs[1]
    yy = ys[1]
    xs[1] = q*xs[1] + xs[0]
    ys[1] = q*ys[1] + ys[0]
    xs[0] = xx
    ys[0] = yy
    sign = -sign
  end while

  // Die Koeffizienten werden endgueltig berechnet.

  x = sign*xs[0]
  y = -sign*ys[0]

  // Der ggT wird berechnet

  gcd = a
end
```

Abb. 2.2. Der erweiterte euklidische Algorithmus

so gelten wegen

$$S_{k-1} = E_k S_k, \quad 1 \le k \le n$$

die Rekursionen

$$u_{k-1} = q_k u_k + u_{k+1}, \quad v_{k-1} = q_k v_k + v_{k+1}, \quad 1 \le k \le n. \tag{2.8}$$

Eine analoge Rekursion gilt auch für die Reste r_k, die im euklidischen Algorithmus berechnet werden.

Die Einträge v_k der Matrizen S_k werden jetzt abgeschätzt.

Lemma 2.10.1. *Es gilt* $0 \le v_k \le r_k/(2\gcd(a,b))$ *für* $0 \le k \le n$.

Beweis. Es gilt $0 = v_n < r_n/(2\gcd(a,b))$. Außerdem ist $q_n \ge 2$ nach Lemma 2.8.5 und $v_{n-1} = 1$. Daher ist $r_{n-1} = q_n r_n \ge 2\gcd(a,b) \ge 2\gcd(a,b)v_{n-1}$. Angenommen, die Behauptung stimmt für $k' \ge k$. Dann folgt $v_{k-1} = q_k v_k + v_{k+1} \le (q_k r_k + r_{k+1})/(2\gcd(a,b)) = r_{k-1}/(2\gcd(a,b))$. Damit ist die behauptete Abschätzung bewiesen. □

Aus Lemma 2.10.1 können wir Abschätzungen für die Koeffizienten x_k und y_k ableiten.

Korollar 2.10.2. *Es gilt* $x_k \le b/(2\gcd(a,b))$ *und* $y_k \le a/(2\gcd(a,b))$ *für* $1 \le k \le n$.

Beweis. Aus $S_0 = T_{n+1}$ folgt $x_n = v_1$ und $y_n = v_0$. Aus Lemma 2.10.1 folgt also die behauptete Abschätzung für $k = n$. Da aber $(x_k)_{k\ge1}$ und $(y_k)_{k\ge0}$ monoton wachsende Folgen sind, ist die Behauptung für $1 \le k \le n$ bewiesen. □

Für die Koeffizienten x und y, die der erweiterte euklidische Algorithmus berechnet, gewinnt man daraus die folgende Abschätzung:

Korollar 2.10.3. *Es gilt* $|x| \le b/(2\gcd(a,b))$ *und* $|y| \le a/(2\gcd(a,b))$.

Wir können auch noch die Koeffizienten x_{n+1} und y_{n+1} bestimmen.

Lemma 2.10.4. *Es gilt* $x_{n+1} = b/\gcd(a,b)$ *und* $y_{n+1} = a/\gcd(a,b)$.

Den Beweis dieses Lemmas überlassen wir dem Leser.

Wir können jetzt die Laufzeit des euklidischen Algorithmus abschätzen. Es stellt sich heraus, daß die Zeitschranke für die Anwendung des erweiterten euklidischen Algorithmus auf a und b von derselben Größenordnung ist wie die Zeitschranke für die Multiplikation von a und b. Das ist ein erstaunliches Resultat, weil der erweiterte euklidische Algorithmus viel aufwendiger aussieht als die Multiplikation.

Theorem 2.10.5. *Sind a und b ganze Zahlen, dann braucht die Anwendung des erweiterten euklidischen Algorithmus auf a und b Zeit* $O((\text{size}\,a)(\text{size}\,b))$.

Beweis. Wir nehmen an, daß $a > b > 0$ ist. Wir haben ja bereits gesehen, daß der erweiterte euklidische Algorithmus nach höchstens einer Iteration entweder fertig ist oder diese Annahme gilt. Es ist leicht einzusehen, daß dafür Zeit $O(\text{size}(a)\,\text{size}(b))$ nötig ist.

Im euklidischen Algorithmus wird die Restefolge $(r_k)_{2\le k\le n+1}$ und die Quotientenfolge $(q_k)_{1\le k\le n}$ berechnet. Die Zahl r_{k+1} ist der Rest der Division von r_{k-1} durch r_k für $1 \le k \le n$. Wie in Abschnitt 2.5 dargestellt, kostet die Berechnung von r_{k+1} höchstens Zeit $O(\text{size}(r_k)\,\text{size}(q_k))$, wobei q_k der Quotient der Division ist.

Wir wissen, daß $r_k \le b$, also $\text{size}(r_k) \le \text{size}(b)$ ist für $1 \le k \le n+1$. Wir wissen ferner, daß $\text{size}(q_k) \le \log(q_k) + 1$ ist für $1 \le k \le n$. Also benötigt der euklidische Algorithmus Zeit

$$T_1(a,b) = O(\text{size}(b)(n + \sum_{k=1}^{n} \log q_k)). \tag{2.9}$$

Nach Theorem 2.8.6 ist

$$n = O(\text{size}\, b). \tag{2.10}$$

Ferner ist

$$\begin{aligned} a = r_0 &= q_1 r_1 + r_2 \ge q_1 r_1 = q_1(q_2 r_2 + r_3) \\ &\ge q_1 q_2 r_2 > \ldots \ge q_1 q_2 \cdots q_n. \end{aligned}$$

Daraus folgt

$$\sum_{k=1}^{n} \log q_k = O(\text{size}\, a). \tag{2.11}$$

Setzt man (2.10) und (2.11) in (2.9) ein, ist die Laufzeitabschätzung für den einfachen euklidischen Algorithmus bewiesen.

Wir schätzen auch noch die Rechenzeit ab, die der erweiterte euklidische Algorithmus benötigt, um die Koeffizienten x und y zu berechnen. In der ersten Iteration wird

$$x_2 = q_1 x_1 + x_0 = 1, \quad y_2 = q_1 y_1 + y_0 = q_1$$

berechnet. Das kostet Zeit $O(\text{size}(q_1)) = O(\text{size}(a))$. Danach wird

$$x_{k+1} = q_k x_k + x_{k-1}, \quad y_{k+1} = q_k y_k + y_{k-1}$$

berechnet, und zwar für $2 \le k \le n$. Gemäß Lemma 2.10.2 ist $x_k, y_k = O(a)$ für $0 \le k \le n$. Damit ist die Laufzeit, die die Berechnung der Koeffizienten x und y braucht

$$T_2(a,b) = O(\text{size}(a)(1 + \sum_{k=2}^{n} \text{size}(q_k)) = O(\text{size}(a)(n + \sum_{k=2}^{n} \log q_k)). \tag{2.12}$$

Wie oben beweist man leicht

$$\prod_{k=2}^{n} q_k \leq b. \tag{2.13}$$

Setzt man dies in (2.12) ein, folgt die Behauptung. Damit ist das Theorem bewiesen. □

2.11 Zerlegung in Primzahlen

Ein zentraler Begriff in der elementaren Zahlentheorie ist der einer Primzahl. Primzahlen werden auch in vielen kryptographischen Verfahren benötigt. In diesem Abschnitt führen wir Primzahlen ein und beweisen, daß sich jede natürliche Zahl bis auf die Reihenfolge in eindeutiger Weise als Produkt von Primzahlen schreiben läßt.

Definition 2.11.1. *Eine natürliche Zahl $p > 1$ heißt* Primzahl*, wenn sie genau zwei positive Teiler hat, nämlich* 1 *und* p*.*

Die ersten neun Primzahlen sind $2, 3, 5, 7, 11, 13, 17, 19, 23$. Die Menge aller Primzahlen bezeichnen wir mit $\mathbb{P}$. Eine natürliche Zahl $a > 1$, die keine Primzahl ist, heißt *zusammengesetzt* . Wenn die Primzahl p die ganze Zahl a teilt, dann heißt p *Primteiler* von a.

Theorem 2.11.2. *Jede natürliche Zahl $a > 1$ hat einen Primteiler.*

Beweis. Die Zahl a besitzt einen Teiler, der größer als 1 ist, nämlich a selbst. Unter allen Teilern von a, die größer als 1 sind, sei p der kleinste. Die Zahl p muß eine Primzahl sein. Wäre sie nämlich keine Primzahl, dann besäße sie einen Teiler b, der

$$1 < b < p \leq a$$

erfüllt. Dies widerspricht der Annahme, daß p der kleinste Teiler von a ist, der größer als 1 ist. □

Das folgende Resultat ist zentral für den Beweis des Zerlegungssatzes.

Lemma 2.11.3. *Wenn eine Primzahl p ein Produkt zweier ganzer Zahlen teilt, so teilt p wenigstens einen der beiden Faktoren.*

Beweis. Angenommen, p teilt ab, aber nicht a. Da p eine Primzahl ist, muß $\gcd(a, p) = 1$ sein. Nach Korollar 2.7.7 gibt es x, y mit $1 = ax + py$. Daraus folgt

$$b = abx + pby.$$

Weil p ein Teiler von abx und pby ist, folgt aus Theorem 2.2.3, daß p auch ein Teiler von b ist. □

Korollar 2.11.4. *Wenn eine Primzahl p ein Produkt $\prod_{i=1}^{k} q_i$ von Primzahlen teilt, dann stimmt p mit einer der Primzahlen $q_1, q_2, \ldots, q_k$ überein.*

Beweis. Wir führen den Beweis durch Induktion über die Anzahl k. Ist $k = 1$, so ist p ein Teiler von q_1, der größer als 1 ist, und stimmt daher mit q_1 überein. Ist $k > 1$, dann ist p ein Teiler von $q_1(q_2 \cdots q_k)$. Nach Lemma 2.11.3 ist p ein Teiler von q_1 oder von $q_2 \cdots q_k$. Da beide Produkte weniger als k Faktoren haben, folgt die Behauptung des Korollars aus der Induktionsannahme. □

Jetzt wird der Hauptsatz der elementaren Zahlentheorie bewiesen.

Theorem 2.11.5. *Jede natürliche Zahl $a > 1$ kann als Produkt von Primzahlen geschrieben werden. Bis auf die Reihenfolge sind die Faktoren in diesem Produkt eindeutig bestimmt.*

Beweis. Wir beweisen den Satz durch Induktion über a. Für $a = 2$ stimmt der Satz. Angenommen, $a > 2$. Nach Theorem 2.11.2 hat a einen Primteiler p. Ist $a/p = 1$, so ist $a = p$ und der Satz ist bewiesen. Sei also $a/p > 1$. Da nach Induktionsvoraussetzung a/p Produkt von Primzahlen ist, kann auch a als Produkt von Primzahlen geschrieben werden. Damit ist die Existenz der Primfaktorzerlegung nachgewiesen. Es fehlt noch die Eindeutigkeit. Seien $a = p_1 \cdots p_k$ und $a = q_1 \cdots q_l$ Primfaktorzerlegungen von a. Nach Korollar 2.11.4 stimmt p_1 mit einer der Primzahlen $q_1, \ldots, q_k$ überein. Durch Umnummerierung erreicht man, daß $p_1 = q_1$ ist. Nach Induktionsannahme ist aber die Primfaktorzerlegung von $a/p_1 = a/q_1$ eindeutig. Also gilt $k = l$ und nach entsprechender Umnummerierung $q_i = p_i$ für $1 \leq i \leq k$. □

Die *Primfaktorzerlegung* einer natürlichen Zahl a ist die Darstellung der Zahl als Produkt von Primfaktoren. Effiziente Algorithmen, die die Primfaktorzerlegung einer natürlichen Zahl berechnen, sind nicht bekannt. Dies ist die Grundlage der Sicherheit des RSA-Verschlüsselungsverfahrens und auch anderer wichtiger kryptographischer Algorithmen. Es ist aber auch kein Beweis bekannt, der zeigt, daß das Faktorisierungsproblem schwer ist. Es ist daher möglich, daß es effiziente Faktorisierungsverfahren gibt, und daß die auch schon bald gefunden werden. Dann sind die entsprechenden kryptographischen Verfahren unsicher und müssen durch andere ersetzt werden.

Beispiel 2.11.6. Der französische Jurist Pierre de Fermat (1601 bis 1665) glaubte, daß die nach ihm benannten *Fermat-Zahlen*

$$F_i = 2^{2^i} + 1$$

sämtlich Primzahlen seien. Tatsächlich sind $F_0 = 3$, $F_1 = 5$, $F_2 = 17$, $F_3 = 257$ und $F_4 = 65537$ Primzahlen. Aber 1732 fand Euler heraus, daß $F_5 = 641 * 6700417$ zusammengesetzt ist. Die angegebene Faktorisierung ist auch die Primfaktorzerlegung der fünften Fermat-Zahl. Auch F_6, F_7, F_8 und

F_9 sind zusammengesetzt. Die Faktorisierung von F_6 wurde 1880 von Landry und Le Lasseur gefunden, die von F_7 erst 1970 von Brillhart und Morrison. Die Faktorisierung von F_8 wurde 1980 von Brent und Pollard gefunden und die von F_9 1990 von Lenstra, Lenstra, Manasse und Pollard. Einerseits sieht man an diesen Daten, wie schwierig das Faktorisierungsproblem ist; immerhin hat es bis 1970 gedauert, bis die 39-stellige Fermat-Zahl F_7 zerlegt war. Andererseits ist die enorme Weiterentwicklung daran zu erkennen, daß nur 20 Jahre später die 155-stellige Fermat-Zahl F_9 faktorisiert wurde.

2.12 Übungen

Übung 2.12.1. Sei α eine reelle Zahl. Zeigen Sie, daß $\lfloor \alpha \rfloor$ die eindeutig bestimmte ganze Zahl z ist mit $0 \leq \alpha - z < 1$.

Übung 2.12.2. Bestimmen Sie die Anzahl der Teiler von 2^n, $n \in \mathbb{Z}_{\geq 0}$.

Übung 2.12.3. Bestimmen Sie alle Teiler von 195.

Übung 2.12.4. Beweisen Sie folgende Modifikation der Division mit Rest: Sind a und b ganze Zahlen, $b > 0$, dann gibt es eindeutig bestimmte ganze Zahlen q und r mit der Eigenschaft, daß $a = qb + r$ und $-b/2 < r \leq b/2$ gilt. Schreiben Sie ein Programm, das den Rest r berechnet.

Übung 2.12.5. Berechnen Sie $1243 \bmod 45$ und $-1243 \bmod 45$.

Übung 2.12.6. Finden Sie eine ganze Zahl a mit $a \bmod 2 = 1$, $a \bmod 3 = 1$, und $a \bmod 5 = 1$.

Übung 2.12.7. Sei m eine natürliche Zahl und seien a, b ganze Zahlen. Zeigen Sie: Genau dann gilt $a \bmod m = b \bmod m$, wenn m die Differenz $b - a$ teilt.

Übung 2.12.8. Berechnen Sie die Binärdarstellung und die Hexadezimaldarstellung von 225.

Übung 2.12.9. Bestimmen Sie die binäre Länge der n-ten Fermat-Zahl $2^{2^n} + 1$, $n \in \mathbb{Z}_{\geq 0}$.

Übung 2.12.10. Schreiben Sie ein Programm, das für gegebenes $g \geq 2$ die g-adische Darstellung einer natürlichen Zahl n berechnet.

Übung 2.12.11. Sei $f(n) = a_d n^d + a_{d-1} n^{d-1} + \ldots + a_0$ ein Polynom mit reellen Koeffizienten, wobei $a_d > 0$ ist. Zeigen Sie, daß $f(n) = O(n^d)$ ist.

Übung 2.12.12. Sei $k \in \mathbb{N}$ und $X \subset \mathbb{N}^k$. Angenommen, $f, g, F, G : X \to \mathbb{R}_{\geq 0}$ mit $f = O(F)$ und $g = O(G)$. Zeigen Sie, daß $f \pm g = O(F + G)$ und $fg = O(FG)$ gilt.

Übung 2.12.13. Seien $a_1, \ldots, a_k$ ganze Zahlen. Beweisen Sie folgende Behauptungen.

1. Es ist $\gcd(a_1, \ldots, a_k) = \gcd(a_1, \gcd(a_2, \ldots, a_k))$.
2. Es ist $a_1\mathbb{Z} + \ldots + a_k\mathbb{Z} = \gcd(a_1, \ldots, a_k)\mathbb{Z}$.
3. Die Gleichung $x_1a_1 + \ldots + x_ka_k = n$ ist genau dann durch ganze Zahlen $x_1, \ldots, x_k$ lösbar, wenn $\gcd(a_1, \ldots, a_k)$ ein Teiler von n ist.
4. Es gibt ganze Zahlen $x_1, \ldots, x_k$ mit $a_1x_1 + \ldots + a_kx_k = \gcd(a_1, \ldots, a_k)$.
5. Der größte gemeinsame Teiler von $a_1, \ldots, a_k$ ist der eindeutig bestimmte nicht negative gemeinsame Teiler von $a_1, \ldots, a_k$, der von allen gemeinsamen Teilern von $a_1, \ldots, a_k$ geteilt wird.

Übung 2.12.14. Beweisen Sie, daß der eulidische Algorithmus auch funktioniert, wenn die Division mit Rest so modifiziert ist wie in Übung 2.12.4.

Übung 2.12.15. Berechnen Sie $\gcd(235, 124)$ samt seiner Darstellung mit dem erweiterten euklidischen Algorithmus.

Übung 2.12.16. Benutzen Sie den modifizierten euklidischen Algorithmus aus Übung 2.12.14, um $\gcd(235, 124)$ einschließlich Darstellung zu berechnen. Vergleichen Sie diese Berechnung mit der Berechnung aus Beispiel 2.12.15.

Übung 2.12.17. Beweisen Sie Lemma 2.10.4.

Übung 2.12.18. Sei $a > b > 0$. Beweisen Sie, daß der modifizierte euklidische Algorithmus aus Beispiel 2.12.14 $O(\log b)$ Iterationen braucht, um $\gcd(a, b)$ zu berechnen.

Übung 2.12.19. Seien a, b positive ganze Zahlen. Man zeige, daß die Anzahl der Iterationen und die Folge der Quotienten im euklidischen Algorithmus nur vom Quotienten a/b abhängt.

Übung 2.12.20. Finden Sie eine Folge $(a_i)_{i\geq 1}$ positiver ganzer Zahlen mit der Eigenschaft, daß der euklidische Algorithmus genau i Iterationen benötigt, um $\gcd(a_{i+1}, a_i)$ zu berechnen.

Übung 2.12.21. Zeigen Sie, daß aus $\gcd(a, m) = 1$ und $\gcd(b, m) = 1$ folgt, daß $\gcd(ab, m) = 1$ ist.

Übung 2.12.22. Berechnen Sie die Primfaktorzerlegung von 37800.

Übung 2.12.23. Zeigen Sie, daß jede zusammengesetzte Zahl $n > 1$ einen Primteiler $p \leq \sqrt{n}$ hat.

Übung 2.12.24. Das *Sieb des Eratosthenes* bestimmt alle Primzahlen unter einer gegebenen Schranke C. Es funktioniert so: Schreibe die Liste $2, 3, 4, 5, \ldots, \lfloor C \rfloor$ von ganzen Zahlen auf. Dann iteriere folgenden Prozeß für $i = 2, 3, \ldots, \lfloor \sqrt{C} \rfloor$. Wenn i noch in der Liste ist, lösche alle echten Vielfachen $2i, 3i, 4i, \ldots$ von i aus der Liste. Die Zahlen, die in der Liste bleiben, sind die gesuchten Primzahlen. Schreiben Sie ein Programm, daß diese Idee implementiert.

3. Kongruenzen und Restklassenringe

In diesem Kapitel führen wir das Rechnen in Restklassenringen und in primen Restklassengruppen ein. Diese Techniken sind von zentraler Bedeutung in kryptographischen Verfahren. Einige der behandelten Sachverhalte gelten allgemeiner in Gruppen. Daher behandeln wir in diesem Kapitel auch endliche Gruppen und ihre Eigenschaften.

Im ganzen Kapitel ist m immer eine natürliche Zahl und kleine lateinische Buchstaben bezeichnen ganze Zahlen.

3.1 Kongruenzen

Definition 3.1.1. *Wir sagen, a ist* kongruent *zu b modulo m und schreiben $a \equiv b \bmod m$, wenn m die Differenz $b - a$ teilt.*

Beispiel 3.1.2. Es gilt $-2 \equiv 19 \bmod 21$, $10 \equiv 0 \bmod 2$.

Es ist leicht zu verifizieren, daß Kongruenz modulo m eine Äquivalenzrelation auf der Menge der ganzen Zahlen ist. Das bedeutet, daß

1. jede ganze Zahl zu sich selbst kongruent ist modulo m (Reflexivität),
2. aus $a \equiv b \bmod m$ folgt, daß auch $b \equiv a \bmod m$ gilt (Symmetrie),
3. aus $a \equiv b \bmod m$ und $b \equiv c \bmod m$ folgt, daß auch $a \equiv c \bmod m$ gilt (Transitivität).

Außerdem gilt folgende Charakterisierung:

Lemma 3.1.3. *Folgende Aussagen sind äquivalent.*

1. $a \equiv b \bmod m$.
2. $a = b + km$ *mit* $k \in \mathbb{Z}$.
3. *a und b lassen bei der Division durch m denselben Rest.*

Die *Äquivalenzklasse* von a besteht aus allen ganzen Zahlen, die sich aus a durch Addition ganzzahliger Vielfacher von m ergeben, sie ist also

$$\{b : b \equiv a \bmod m\} = a + m\mathbb{Z}.$$

Man nennt sie *Restklasse* von $a \bmod m$.

Beispiel 3.1.4. Die Restklasse von 1 mod 4 ist die Menge $\{1, 1 \pm 4, 1 \pm 2 * 4, 1 \pm 3 * 4, \ldots\} = \{1, -3, 5, -7, 9, -11, 13, \ldots\}$.

Die Restklasse von 0 mod 2 ist die Menge aller geraden ganzen Zahlen. Die Restklasse von 1 mod 2 ist die Menge aller ungeraden ganzen Zahlen.

Die Restklassen mod 4 sind $0 + 4\mathbb{Z}$, $1 + 4\mathbb{Z}$, $2 + 4\mathbb{Z}$, $3 + 4\mathbb{Z}$.

Die Menge aller Restklassen mod m wird mit $\mathbb{Z}/m\mathbb{Z}$ bezeichnet. Sie hat m Elemente, weil genau die Reste $0, 1, 2, \ldots, m-1$ bei der Division durch m auftreten. Ein *Vertretersystem* für diese Äquivalenzrelation ist eine Menge ganzer Zahlen, die aus jeder Restklasse mod m genau ein Element enthält. Jedes solche Vertretersystem heißt *volles Restsystem* mod m.

Beispiel 3.1.5. Ein volles Restsystem mod 3 enthält je ein Element aus den Restklassen $3\mathbb{Z}$, $1+3\mathbb{Z}$, $2+3\mathbb{Z}$. Also sind folgende Mengen volle Restsysteme mod 3: $\{0, 1, 2\}, \{3, -2, 5\}, \{9, 16, 14\}$.

Ein volles Restsystem mod m ist z.B. die Menge $\{0, 1, \ldots, m-1\}$. Seine Elemente nennt man *kleinste nicht negative Reste* mod m. Wir bezeichnen dieses Vertretersystem mit $\mathbb{Z}_m$. Genauso ist die Menge $\{1, 2, \ldots, m\}$ ein volles Restsystem mod m. Seine Elemente heißen *kleinste positive Reste* mod m. Schließlich ist $\{n+1, n+2, \ldots, n+m\}$ mit $n = -\lceil m/2 \rceil$ ein vollständiges Restsystem mod m. Seine Elemente heißen *absolut kleinste Reste* mod m.

Beispiel 3.1.6. Es ist

$$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$$

die Menge der kleinsten nicht negativen Reste mod 13 und

$$\{-6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6\}$$

ist die Menge der absolut kleinsten Reste mod 13.

Wir brauchen einige Rechenregeln für Kongruenzen. Die erlauben es uns später, eine Ringstruktur auf der Menge der Restklassen mod m zu definieren.

Theorem 3.1.7. *Aus* $a \equiv b \bmod m$ *und* $c \equiv d \bmod m$ *folgt* $-a \equiv -b \bmod m$, $a + c \equiv b + d \bmod m$ *und* $ac \equiv bd \bmod m$.

Beweis. Weil m ein Teiler von $a-b$ ist, ist m auch ein Teiler von $-a+b$. Daher ist $-a \equiv -b \bmod m$. Weil m ein Teiler von $a-b$ und von $c-d$ ist, ist m auch ein Teiler von $a-b+c-d = (a+c)-(b+d)$. Daher ist $a+c \equiv b+d \bmod m$. Um zu zeigen, daß $ac \equiv bd \bmod m$ ist, schreiben wir $a = b + lm$ und $c = d + km$. Dann erhalten wir $ac = bd + m(ld + kb + lkm)$, wie behauptet. □

Beispiel 3.1.8. Wir wenden die Rechenregeln aus Theorem 3.1.7 an, um zu beweisen, daß die fünfte Fermat-Zahl $2^{2^5} + 1$ durch 641 teilbar ist. Zunächst gilt

$$641 = 640 + 1 = 5 * 2^7 + 1.$$

Dies zeigt

$$5 * 2^7 \equiv -1 \bmod 641.$$

Aus Theorem 3.1.7 folgt, daß diese Kongruenz bestehen bleibt, wenn man die rechte und linke Seite viermal mit sich selbst multipliziert, also zur vierten Potenz erhebt. Das machen wir und erhalten

$$5^4 * 2^{28} \equiv 1 \bmod 641. \tag{3.1}$$

Andererseits ist

$$641 = 625 + 16 = 5^4 + 2^4.$$

Daraus gewinnt man

$$5^4 \equiv -2^4 \bmod 641.$$

Wenn man diese Kongruenz in (3.1) benutzt, erhält man

$$-2^{32} \equiv 1 \bmod 641,$$

also

$$2^{32} + 1 \equiv 0 \bmod 641.$$

Dies beweist, daß 641 ein Teiler der fünften Fermat-Zahl ist.

Wir wollen zeigen, daß die Menge der Restklassen mod m einen Ring bildet. Wir wiederholen in den folgenden Abschnitten kurz einige Grundbegriffe.

3.2 Halbgruppen

Definition 3.2.1. *Ist X eine Menge, so heißt eine Abbildung $\circ : X \times X \to X$, die jedem Paar (x_1, x_2) von Elementen aus X ein Element $x_1 \circ x_2$ zuordnet, eine* innere Verknüpfung *auf X.*

Beispiel 3.2.2. Auf der Menge der reellen Zahlen kennen wir bereits die inneren Verknüpfungen Addition und Multiplikation.

Auf der Menge $\mathbb{Z}/m\mathbb{Z}$ aller Restklassen modulo m führen wir zwei innere Verknüpfungen ein, Addition und Multiplikation.

Definition 3.2.3. *Die Summe der Restklassen $a + m\mathbb{Z}$ und $b + m\mathbb{Z}$ ist $(a + m\mathbb{Z}) + (b + m\mathbb{Z}) = (a + b) + m\mathbb{Z}$. Das Produkt der Restklassen $a + m\mathbb{Z}$ und $b + m\mathbb{Z}$ ist $(a + m\mathbb{Z}) \cdot (b + m\mathbb{Z}) = (a \cdot b) + m\mathbb{Z}$.*

Man beachte, daß Summe und Produkt von Restklassen modulo m unter Verwendung von Vertretern dieser Restklassen definiert sind. Aus Theorem 3.1.7 folgt aber, daß die Definition von den Vertretern unabhängig ist. In der Praxis werden Restklassen durch feste Vertreter dargestellt und die Rechnung wird mit diesen Vertretern durchgeführt. Man erhält auf diese Weise eine Addition und eine Multiplikation auf jedem Vertretersystem.

Beispiel 3.2.4. Wir verwenden zur Darstellung von Restklassen die kleinsten nicht negativen Reste. Es ist $(3 + 5\mathbb{Z}) + (2 + 5\mathbb{Z}) = (5 + 5\mathbb{Z}) = 5\mathbb{Z}$ und $(3 + 5\mathbb{Z})(2 + 5\mathbb{Z}) = 6 + 5\mathbb{Z} = 1 + 5\mathbb{Z}$. Diese Rechnungen kann man auch als $3 + 2 \equiv 0 \bmod 5$ und $3 * 2 \equiv 1 \bmod 5$ darstellen.

Definition 3.2.5. *Sei $\circ$ eine innere Verknüpfung auf der Menge X. Sie heißt* assoziativ, *wenn $(a \circ b) \circ c = a \circ (b \circ c)$ gilt für alle $a, b, c \in X$. Sie heißt* kommutativ, *wenn $a \circ b = b \circ a$ gilt für alle $a, b \in X$.*

Beispiel 3.2.6. Addition und Multiplikation auf der Menge der reellen Zahlen sind assoziative und kommutative Verknüpfungen. Dasselbe gilt für Addition und Multiplikation auf der Menge $\mathbb{Z}/m\mathbb{Z}$ der Restklassen modulo m.

Definition 3.2.7. *Ein Paar $(H, \circ)$, bestehend aus einer nicht leeren Menge H und einer assoziativen inneren Verknüpfung $\circ$ auf H, heißt eine* Halbgruppe. *Die Halbgruppe heißt* kommutativ *oder* abelsch, *wenn die innere Verknüpfung $\circ$ kommutativ ist.*

Beispiel 3.2.8. Kommutative Halbgruppen sind $(\mathbb{Z}, +)$, $(\mathbb{Z}, \cdot)$, $(\mathbb{Z}/m\mathbb{Z}, +)$, $(\mathbb{Z}/m\mathbb{Z}, \cdot)$.

Sei $(H, \circ)$ eine Halbgruppe und bezeichne $a^1 = a$ und $a^{n+1} = a \circ a^n$ für $a \in H$ und $n \in \mathbb{N}$, dann gelten die *Potenzgesetze*

$$a^n \circ a^m = a^{n+m}, \quad (a^n)^m = a^{nm}, \quad a \in H, n, m \in \mathbb{N}. \tag{3.2}$$

Sind $a, b \in H$ und gilt $a \circ b = b \circ a$, dann folgt

$$(a \circ b)^n = a^n \circ b^n. \tag{3.3}$$

Ist die Halbgruppe also kommutativ, so gilt (3.3) immer.

Definition 3.2.9. *Ein* neutrales Element *der Halbgruppe $(H, \circ)$ ist ein Element $e \in H$, das $e \circ a = a \circ e = a$ erfüllt für alle $a \in H$. Enthält die Halbgruppe ein neutrales Element, so heißt sie* Monoid.

Eine Halbgruppe hat höchstens ein neutrales Element. (siehe Übung 3.23.3).

Definition 3.2.10. *Ist e das neutrale Element der Halbgruppe $(H, \circ)$ und ist $a \in H$, so heißt $b \in H$* Inverses *von a, wenn $a \circ b = b \circ a = e$ gilt. Besitzt a ein Inverses, so heißt a* invertierbar *in der Halbgruppe.*

In Monoiden besitzt jedes Element höchstens ein Inverses (siehe Übung 3.23.5).

Beispiel 3.2.11. 1. Die Halbgruppe $(\mathbb{Z}, +)$ besitzt das neutrale Element 0. Das Inverse von a ist $-a$.
2. Die Halbgruppe $(\mathbb{Z}, \cdot)$ besitzt das neutrale Element 1. Die einzigen invertierbaren Elemente sind 1 und -1.
3. Die Halbgruppe $(\mathbb{Z}/m\mathbb{Z}, +)$ besitzt das neutrale Element $m\mathbb{Z}$. Das Inverse von $a + m\mathbb{Z}$ ist $-a + m\mathbb{Z}$.
4. Die Halbgruppe $(\mathbb{Z}/m\mathbb{Z}, \cdot)$ besitzt das neutrale Element $1 + m\mathbb{Z}$. Die invertierbaren Elemente werden später bestimmt.

3.3 Gruppen

Definition 3.3.1. *Eine* Gruppe *ist eine Halbgruppe, die ein neutrales Element besitzt und in der jedes Element invertierbar ist. Die Gruppe heißt* kommutativ *oder* abelsch*, wenn die Halbgruppe kommutativ ist.*

Beispiel 3.3.2. 1. Die Halbgruppe $(\mathbb{Z}, +)$ ist eine abelsche Gruppe.
2. Die Halbgruppe $(\mathbb{Z}, \cdot)$ ist keine Gruppe, weil nicht jedes Element ein Inverses besitzt.
3. Die Halbgruppe $(\mathbb{Z}/m\mathbb{Z}, +)$ ist eine abelsche Gruppe.

Ist $(G, \cdot)$ eine multiplikativ geschriebene Gruppe, bezeichnet a^{-1} das Inverse eines Elementes a aus G und setzt man $a^{-n} = (a^{-1})^n$ für jede natürliche Zahl n, so gelten die Potenzgesetze (3.2) für alle ganzzahligen Exponenten. Ist die Gruppe abelsch, so gilt (3.3) für alle ganzen Zahlen n.

In einer Gruppe gelten folgende *Kürzungsregeln*, die man durch Multiplikation mit einem geeigneten Inversen beweist.

Theorem 3.3.3. *Sei $(G, \cdot)$ eine Gruppe und $a, b, c \in G$. Aus $ca = cb$ folgt $a = b$ und aus $ac = bc$ folgt $a = b$.*

Definition 3.3.4. *Die* Ordnung *einer Gruppe oder Halbgruppe ist die Anzahl ihrer Elemente.*

Beispiel 3.3.5. Die additive Gruppe $\mathbb{Z}$ hat unendliche Ordnung. Die additive Gruppe $\mathbb{Z}/m\mathbb{Z}$ hat die Ordnung m.

3.4 Restklassenringe

Definition 3.4.1. *Ein* Ring *ist ein Tripel $(R, +, \cdot)$, für das $(R, +)$ eine abelsche Gruppe und $(R, \cdot)$ eine Halbgruppe ist, und für das zusätzlich die Distributivgesetze $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$ und $(x + y) \cdot z = (x \cdot z) + (y \cdot z)$*

für alle $x, y, z \in R$ gelten. Der Ring heißt kommutativ*, wenn die Halbgruppe $(R, \cdot)$ kommutativ ist. Ein* Einselement *des Ringes ist ein neutrales Element der Halbgruppe $(R, \cdot)$.*

Beispiel 3.4.2. Das Tripel $(\mathbb{Z}, +, \cdot)$ ist ein kommutativer Ring mit Einselement 1 und daraus leitet man ab, daß $(\mathbb{Z}/m\mathbb{Z}, +, \cdot)$ ein kommutativer Ring mit Einselement $1 + m\mathbb{Z}$ ist. Der letztere Ring heißt *Restklassenring* modulo m.

In der Definition wurde festgelegt, daß Ringe Tripel sind, daß also die Verknüpfungen immer miterwähnt werden müssen. Im allgemeinen ist aber klar, welche Verknüpfungen gemeint sind. Dann lassen wir sie einfach weg und sprechen zum Beispiel von dem Restklassenring $\mathbb{Z}/m\mathbb{Z}$.

Definition 3.4.3. *Sei R ein Ring mit Einselement. Ein Element a von R heißt* invertierbar *oder* Einheit*, wenn es in der multiplikativen Halbgruppe von R invertierbar ist. Das Element a heißt* Nullteiler*, wenn es von Null verschieden ist und es ein von Null verschiedenes Element $b \in R$ gibt mit $ab = 0$ oder $ba = 0$. Enthält R keine Nullteiler, so heißt R* nullteilerfrei.

In Übung 3.23.9 wird gezeigt, daß die invertierbaren Elemente eines kommutativen Rings R mit Einselement eine Gruppe bilden. Sie heißt *Einheitengruppe* des Rings und wird mit R^* bezeichnet.

Beispiel 3.4.4. Der Ring der ganzen Zahlen ist nullteilerfrei.

Die Nullteiler im Restklassenring $\mathbb{Z}/m\mathbb{Z}$ sind die Restklassen $a + m\mathbb{Z}$, für die $1 < \gcd(a, m) < m$ ist. Ist $a + m\mathbb{Z}$ nämlich ein Nullteiler von $\mathbb{Z}/m\mathbb{Z}$, dann muß es eine ganze Zahl b geben mit $ab \equiv 0 \bmod m$, aber es gilt weder $a \equiv 0 \bmod m$ noch $b \equiv 0 \bmod m$. Also ist m ein Teiler von ab, aber weder von a noch von b. Das bedeutet, daß $1 < \gcd(a, m) < m$ gelten muß. Ist umgekehrt $1 < \gcd(a, m) < m$ und $b = m/\gcd(a, m)$, so ist $a \not\equiv 0 \bmod m$, $ab \equiv 0 \bmod m$ und $b \not\equiv 0 \bmod m$. Also ist $a + m\mathbb{Z}$ ein Nullteiler von $\mathbb{Z}/m\mathbb{Z}$.

Ist m eine Primzahl, so besitzt $\mathbb{Z}/m\mathbb{Z}$ also keine Nullteiler.

3.5 Körper

Definition 3.5.1. *Ein* Körper *ist ein kommutativer Ring mit Einselement, in dem jedes von Null verschiedene Element invertierbar ist.*

Beispiel 3.5.2. Die Menge der ganzen Zahlen ist kein Körper, weil die einzigen invertierbaren ganzen Zahlen 1 und -1 sind. Sie ist aber im Körper der rationalen Zahlen enthalten. Auch die reellen und komplexen Zahlen bilden Körper. Wie wir unten sehen werden, ist der Restklassenring $\mathbb{Z}/m\mathbb{Z}$ genau dann ein Körper, wenn m eine Primzahl ist.

3.6 Division im Restklassenring

Teilbarkeit in Ringen ist definiert wie Teilbarkeit in $\mathbb{Z}$. Sei R ein Ring und seien $a, n \in R$.

Definition 3.6.1. *Man sagt a* teilt *n, wenn es $b \in R$ gibt mit $n = ab$.*

Wenn a das Ringelement n teilt, dann heißt a *Teiler* von n und n *Vielfaches* von a und man schreibt $a|n$. Man sagt auch, n ist durch a *teilbar*. Wenn a kein Teiler von n ist, dann schreibt man $a \nmid n$.

Wir untersuchen das Problem, durch welche Elemente des Restklassenrings mod m dividiert werden darf, welche Restklasse $a + m\mathbb{Z}$ also ein multiplikatives Inverses besitzt. Zuerst stellen wir fest, daß die Restklasse $a + m\mathbb{Z}$ genau dann in $\mathbb{Z}/m\mathbb{Z}$ invertierbar ist, wenn die Kongruenz

$$ax \equiv 1 \bmod m \tag{3.4}$$

lösbar ist. Wann das der Fall ist, wird im nächsten Satz ausgesagt.

Theorem 3.6.2. *Die Restklasse $a + m\mathbb{Z}$ ist genau dann in $\mathbb{Z}/m\mathbb{Z}$ invertierbar, d.h. die Kongruenz* (3.4) *ist genau dann lösbar, wenn* $\gcd(a, m) = 1$ *gilt. Ist* $\gcd(a, m) = 1$*, dann ist das Inverse von $a + m\mathbb{Z}$ eindeutig bestimmt, d.h. die Lösung x von* (3.4) *ist eindeutig bestimmt mod m.*

Beweis. Sei $g = \gcd(a, m)$ und sei x eine Lösung von (3.4), dann ist g ein Teiler von m und damit ein Teiler von $ax-1$. Aber g ist auch ein Teiler von a. Also ist g ein Teiler von 1, d.h. $g = 1$, weil g als ggT positiv ist. Sei umgekehrt $g = 1$. Dann gibt es nach Korollar 2.7.7 Zahlen x, y mit $ax + my = 1$, d.h. $ax - 1 = -my$. Dies zeigt, daß x eine Lösung der Kongruenz (3.4) ist, und daß $x + m\mathbb{Z}$ ein Inverses von $a + m\mathbb{Z}$ in $\mathbb{Z}/m\mathbb{Z}$ ist.

Zum Beweis der Eindeutigkeit sei $v+m\mathbb{Z}$ ein weiteres Inverses von $a+m\mathbb{Z}$. Dann gilt $ax \equiv av \bmod m$. Also teilt m die Zahl $a(x-v)$. Weil $\gcd(a, m) = 1$ ist, folgt daraus, daß m ein Teiler von $x - v$ ist. Somit ist $x \equiv v \bmod m$. □

Eine Restklasse $a+m\mathbb{Z}$ mit $\gcd(a, m) = 1$ heißt *prime Restklasse* modulo m. Aus Theorem 3.6.2 folgt, daß eine Restklasse $a + m\mathbb{Z}$ mit $1 \leq a < m$ entweder ein Nullteiler oder eine prime Restklasse, d.h. eine Einheit des Restklassenrings mod m, ist.

Im Beweis von Theorem 3.6.2 wurde gezeigt, wie man die Kongruenz $ax \equiv 1 \bmod m$ mit dem erweiterten euklidischen Algorithmus (siehe Abschnitt 2.9) löst. Man muß nur die Darstellung $1 = ax + my$ berechnen. Man braucht sogar nur den Koeffizienten x. Nach Theorem 2.10.5 kann die Lösung der Kongruenz also effizient berechnet werden.

Beispiel 3.6.3. Sei $m = 12$. Die Restklasse $a + 12\mathbb{Z}$ ist genau dann invertierbar in $\mathbb{Z}/12\mathbb{Z}$, wenn $\gcd(a, 12) = 1$. Die invertierbaren Restklassen mod 12 sind also $1+12\mathbb{Z}, 5+12\mathbb{Z}, 7+12\mathbb{Z}, 11+12\mathbb{Z}$. Um das Inverse von $5+12\mathbb{Z}$ zu finden, benutzen wir den erweiterten euklidischen Algorithmus. Wir erhalten $5 * 5 \equiv 1 \bmod 12$. Entsprechend gilt $7 * 7 \equiv 1 \bmod 12$, $11 * 11 \equiv 1 \bmod 12$.

Wir führen noch den Restklassenkörper modulo einer Primzahl ein, der in der Kryptographie sehr oft benutzt wird.

Theorem 3.6.4. *Der Restklassenring $\mathbb{Z}/m\mathbb{Z}$ ist genau dann ein Körper, wenn m eine Primzahl ist.*

Beweis. Gemäß Theorem 3.6.2 ist $\mathbb{Z}/m\mathbb{Z}$ genau dann ein Körper, wenn $\gcd(k, m) = 1$ gilt für alle k mit $1 \leq k < m$. Dies ist genau dann der Fall, wenn m eine Primzahl ist. □

3.7 Rechenzeit für die Operationen im Restklassenring

In allen Verfahren der Public-Key-Kryptographie wird intensiv in Restklassenringen gerechnet. Oft müssen diese Rechnungen auf Chipkarten ausgeführt werden. Daher ist es wichtig, zu wissen, wie effizient diese Rechnungen ausgeführt werden können. Das wird in diesem Abschnitt beschrieben.

Wir gehen davon aus, daß die Elemente des Restklassenrings $\mathbb{Z}/m\mathbb{Z}$ durch ihre kleinsten nicht negativen Vertreter $\{0, 1, 2, \ldots, m-1\}$ dargestellt werden. Unter dieser Voraussetzung schätzen wir den Aufwand der Operationen im Restklassenring ab.

Seien also $a, b \in \{0, 1, \ldots, m-1\}$.

Um $(a+m\mathbb{Z})+(b+m\mathbb{Z})$ zu berechnen, müssen wir $(a+b) \bmod m$ berechnen. Wir bestimmen also zuerst $c = a + b$. Die gesuchte Summe ist $c + m\mathbb{Z}$, aber c ist vielleicht der falsche Vertreter. Es gilt nämlich $0 \leq c < 2m$. Ist $0 \leq c < m$, so ist c der richtige Vertreter. Ist $m \leq c < 2m$, so ist der richtige Vertreter $c - m$, weil $0 \leq c - m < m$ ist. In diesem Fall ersetzt man c durch $c - m$. Entsprechend berechnet man $(a + m\mathbb{Z}) - (b + m\mathbb{Z})$. Man bestimmt $c = a - b$. Dann ist $-m < c < m$. Gilt $0 \leq c < m$, so ist c der richtige Vertreter der Differenz. Ist $-m < c < 0$, so ist der richtige Vertreter $c + m$. Also muß c durch $c + m$ ersetzt werden. Aus den Ergebnissen von Abschnitt 2.5 folgt also, daß Summe und Differenz zweier Restklassen modulo m in Zeit $O(\text{size}\, m)$ berechnet werden können.

Nun wird $(a+m\mathbb{Z})(b+m\mathbb{Z})$ berechnet. Dazu wird $c = ab$ bestimmt. Dann ist $0 \leq c < m^2$. Wir dividieren c mit Rest durch m und ersetzen c durch den Rest dieser Division. Für den Quotienten q dieser Division gilt $0 \leq q < m$. Nach den Ergebnissen aus Abschnitt 2.5 kann man die Multiplikation und die Division in Zeit $O((\text{size}\, m)^2)$ durchführen. Die Restklassen können also in Zeit $O((\text{size}\, m)^2)$ multipliziert werden.

Schließlich wird die Invertierung von $a + m\mathbb{Z}$ diskutiert. Man berechnet $g = \gcd(a, m)$ und x mit $ax \equiv g \bmod m$ und $0 \leq x < m$. Hierzu benutzt man den erweiterten euklidischen Algorithmus. Nach Korollar 2.10.3 gilt $|x| \leq m/(2g)$. Der Algorithmus liefert aber möglicherweise einen negativen Koeffizienten x, den man durch $x + m$ ersetzt. Gemäß Theorem 2.10.5 erfordert diese Berechnung Zeit $O((\text{size}\, m)^2)$. Die Restklasse $a+m\mathbb{Z}$ ist genau

dann invertierbar, wenn $g = 1$ ist. In diesem Fall ist x der kleinste nicht negative Vertreter der inversen Klasse. Die gesamte Rechenzeit ist $O((\text{size}\, m)^2)$. Es folgt, daß auch die Division durch eine prime Restklasse mod m Zeit $O((\text{size}\, m)^2)$ kostet.

In allen Algorithmen müssen nur konstant viele Zahlen der Größe $O(\text{size}\, m)$ gespeichert werden. Daher brauchen die Algorithmen auch nur Speicherplatz $O(\text{size}\, m)$. Wir merken an, daß es asymptotisch effizientere Algorithmen für die Multiplikation und Division von Restklassen gibt. Sie benötigen Zeit $O(\log m(\log\log m)^2)$ (siehe [3]). Für Zahlen der Größenordnung, um die es in der Kryptographie geht, sind diese Algorithmen aber langsamer als die hier analysierten. Die $O((\text{size}\, m)^2)$-Algorithmen lassen in vielen Situationen Optimierungen zu. Einen Überblick darüber findet man in [52].

Wir haben folgenden Satz bewiesen.

Theorem 3.7.1. *Angenommen, die Restklassen modulo m werden durch ihre kleinsten nicht negativen Vertreter dargestellt. Dann erfordert die Addition und Subtraktion zweier Restklassen Zeit $O(\text{size}\, m)$ und die Multiplikation und Division zweier Restklassen kostet Zeit $O((\text{size}\, m)^2)$. Alle Operationen brauchen Speicherplatz $O(\text{size}\, m)$.*

3.8 Prime Restklassengruppen

Von fundamentaler Bedeutung in der Kryptographie ist folgendes Ergebnis.

Theorem 3.8.1. *Die Menge aller primen Restklassen modulo m bildet eine endliche abelsche Gruppe bezüglich der Multiplikation.*

Beweis. Nach Theorem 3.6.2 ist diese Menge die Einheitengruppe des Restklassenrings mod m.

Die Gruppe der primen Restklassen modulo m heißt *prime Restklassengruppe* modulo m und wird mit $(\mathbb{Z}/m\mathbb{Z})^*$ bezeichnet. Ihre Ordnung bezeichnet man mit $\varphi(m)$. Die Abbildung

$$\mathbb{N} \to \mathbb{N}, m \mapsto \varphi(m)$$

heißt *Eulersche φ-Funktion*. Man beachte, daß $\varphi(m)$ die Anzahl der Zahlen a in $\{1, 2, \ldots, m\}$ ist mit $\gcd(a, m) = 1$. Insbesondere ist $\varphi(1) = 1$.

Beispiel 3.8.2. Z.B. ist $(\mathbb{Z}/12\mathbb{Z})^* = \{1 + 12\mathbb{Z}, 5 + 12\mathbb{Z}, 7 + 12\mathbb{Z}, 11 + 12\mathbb{Z}\}$ die prime Restklassengruppe mod 12. Also ist $\varphi(12) = 4$.

Einige Werte der Eulerschen φ-Funktion findet man in Tabelle 3.1.

m	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\varphi(m)$	1	1	2	2	4	2	6	4	6	4	10	4	12	6	8

Tabelle 3.1. Werte der Eulerschen φ-Funktion

Man sieht, daß in dieser Tabelle $\varphi(p) = p - 1$ für die Primzahlen p gilt. Dies ist auch allgemein richtig, weil für eine Primzahl p alle Zahlen a zwischen 1 und $p - 1$ zu p teilerfremd sind. Also gilt der folgende Satz:

Theorem 3.8.3. *Falls p eine Primzahl ist, gilt $\varphi(p) = p - 1$.*

Die Eulersche φ-Funktion hat folgende nützliche Eigenschaft:

Theorem 3.8.4.

$$\sum_{d|m,d>0} \varphi(d) = m.$$

Beweis. Es gilt

$$\sum_{d|m,d>0} \varphi(d) = \sum_{d|m,d>0} \varphi(m/d),$$

weil die Menge der positiven Teiler von m genau $\{m/d : d|m, d > 0\}$ ist. Nun ist $\varphi(m/d)$ die Anzahl der ganzen Zahlen a in der Menge $\{1, \ldots, m/d\}$ mit $\gcd(a, m/d) = 1$. Also ist $\varphi(m/d)$ die Anzahl der ganzen Zahlen b in $\{1, 2, \ldots, m\}$ mit $\gcd(b, m) = d$. Daher ist

$$\sum_{d|m,d>0} \varphi(d) = \sum_{d|m,d>0} |\{b : 1 \leq b \leq m \text{ mit } \gcd(b, m) = d\}|.$$

Es gilt aber

$$\{1, 2, \ldots, m\} = \cup_{d|m,d>0}\{b : 1 \leq b \leq m \text{ mit } \gcd(b, m) = d\}.$$

Daraus folgt die Behauptung. □

3.9 Ordnung von Gruppenelementen

Als nächstes führen wir Elementordnungen und ihre Eigenschaften ein. Dazu sei G eine Gruppe, die multiplikativ geschrieben ist, mit neutralem Element 1.

Definition 3.9.1. *Sei $g \in G$. Wenn es eine natürliche Zahl e gibt mit $g^e = 1$, dann heißt die kleinste solche Zahl* Ordnung *von g in G. Andernfalls sagt man, daß die Ordnung von g in G unendlich ist. Die Ordnung von g in G wird mit* $\text{order}_G\, g$ *bezeichnet. Wenn es klar ist, um welche Gruppe es sich handelt, schreibt man auch* $\text{order}\, g$.

Theorem 3.9.2. *Sei $g \in G$ und $e \in \mathbb{Z}$. Dann gilt $g^e = 1$ genau dann, wenn e durch die Ordnung von g in G teilbar ist.*

Beweis. Sei $n = \operatorname{order} g$. Wenn $e = kn$ ist, dann folgt

$$g^e = g^{kn} = (g^n)^k = 1^k = 1.$$

Sei umgekehrt $g^e = 1$. Sei $e = qn + r$ mit $0 \leq r < n$. Dann folgt

$$g^r = g^{e-qn} = g^e (g^n)^{-q} = 1.$$

Weil n die kleinste natürliche Zahl ist mit $g^n = 1$, und weil $0 \leq r < n$ ist, muß $r = 0$ und damit $e = qn$ sein. Also ist n ein Teiler von e, wie behauptet. □

Korollar 3.9.3. *Sei $g \in G$ und seien k, l ganze Zahlen. Dann gilt $g^l = g^k$ genau dann, wenn $l \equiv k \bmod \operatorname{order} g$ ist.*

Beweis. Setze $e = l - k$ und wende Theorem 3.9.2 an. □

Beispiel 3.9.4. Wir bestimmen die Ordnung von $2 + 13\mathbb{Z}$ in $(\mathbb{Z}/13\mathbb{Z})^*$. Wir ziehen dazu folgende Tabelle heran.

k	0	1	2	3	4	5	6	7	8	9	10	11	12
$2^k \bmod 13$	1	2	4	8	3	6	12	11	9	5	10	7	1

Man sieht, daß die Ordnung von $2 + 13\mathbb{Z}$ den Wert 12 hat. Diese Ordnung ist gleich der Gruppenordnung von $(\mathbb{Z}/13\mathbb{Z})^*$. Dies stimmt aber nicht für jedes Gruppenelement. Zum Beispiel hat $4 + 13\mathbb{Z}$ die Ordnung 6.

Wir bestimmen noch die Ordnung von Potenzen.

Theorem 3.9.5. *Ist $g \in G$ von endlicher Ordnung e und ist n eine ganze Zahl, so ist* $\operatorname{order} g^n = e/\gcd(e, n)$.

Beweis. Es gilt

$$(g^n)^{e/\gcd(e,n)} = (g^e)^{n/\gcd(e,n)} = 1.$$

Nach Theorem 3.9.3 ist also $e/\gcd(e, n)$ ein Vielfaches der Ordnung von g^n. Gelte

$$1 = (g^n)^k = g^{nk},$$

dann folgt aus Theorem 3.9.3, daß e ein Teiler von nk ist. Daher ist $e/\gcd(e, n)$ ein Teiler von k, woraus die Behauptung folgt. □

3.10 Untergruppen

Wir führen Untergruppen ein. Mit G bezeichnen wir eine Gruppe.

Definition 3.10.1. *Eine Teilmenge U von G heißt* Untergruppe *von G, wenn U mit der Verknüpfung von G selbst eine Gruppe ist.*

Beispiel 3.10.2. Für jedes $g \in G$ bildet die Menge $\{g^k : k \in \mathbb{Z}\}$ eine Untergruppe von G. Sie heißt die von g *erzeugte Untergruppe* und wir schreiben $\langle g \rangle$ für diese Untergruppe.

Hat g endliche Ordnung e, dann ist $\langle g \rangle = \{g^k : 0 \leq k < e\}$. Ist nämlich x eine ganze Zahl, dann gilt $g^x = g^{x \bmod e}$ nach Korollar 3.9.3. Aus Korollar 3.9.3 folgt ebenfalls, daß in diesem Fall e die Ordnung von $\langle g \rangle$ ist.

Beispiel 3.10.3. Die von $2 + 13\mathbb{Z}$ erzeugte Untergruppe von $(\mathbb{Z}/13\mathbb{Z})^*$ ist gemäß Beispiel 3.9.4 die ganze Gruppe $(\mathbb{Z}/13\mathbb{Z})^*$. Die von $4 + 13\mathbb{Z}$ erzeugte Untergruppe hat die Ordnung 6. Sie ist $\{k + 13\mathbb{Z} : k = 1, 4, 3, 12, 9, 10\}$.

Definition 3.10.4. *Wenn $G = \langle g \rangle$ für ein $g \in G$ ist, so heißt G* zyklisch *und g heißt* Erzeuger *von G. Die Gruppe G ist dann die von g erzeugte Gruppe.*

Beispiel 3.10.5. Die additive Gruppe $\mathbb{Z}$ ist zyklisch. Sie hat zwei Erzeuger, nämlich 1 und -1.

Theorem 3.10.6. *Ist G endlich und zyklisch, so hat G genau $\varphi(|G|)$ Erzeuger, und die haben alle die Ordnung $|G|$.*

Beweis. Sei $g \in G$ ein Element der Ordnung e. Dann hat die von g erzeugte Gruppe die Ordnung e. Also ist ein Element von G auch Erzeuger von G, wenn es die Ordnung $|G|$ hat. Wir bestimmen die Anzahl der Elemente der Ordnung $|G|$ von G. Sei g ein Erzeuger von G. Dann ist $G = \{g^k : 0 \leq k < |G|\}$. Nach Theorem 3.9.5 hat ein Element dieser Menge genau dann die Ordnung $|G|$, wenn $\gcd(k, |G|) = 1$ ist. Das bedeutet, daß die Anzahl der Erzeuger von G genau $\varphi(|G|)$ ist. □

Beispiel 3.10.7. Weil $2+13\mathbb{Z}$ in $(\mathbb{Z}/13\mathbb{Z})^*$ die Ordnung 12 hat, ist $(\mathbb{Z}/13\mathbb{Z})^*$ zyklisch. Unten werden wir beweisen, daß $(\mathbb{Z}/p\mathbb{Z})^*$ immer zyklisch ist, wenn p eine Primzahl ist. Nach Beispiel 3.9.4 sind die Erzeuger dieser Gruppe die Restklassen $a + 13\mathbb{Z}$ mit $a \in \{2, 6, 7, 11\}$.

Um das nächste Resultat zu beweisen, brauchen wir ein paar Begriffe. Eine Abbildung $f : X \to Y$ heißt *injektiv*, falls aus $f(x) = f(y)$ immer $x = y$ folgt. Zwei verschiedene Elemente aus X können also nie die gleichen Funktionswerte haben. Die Abbildung heißt *surjektiv*, wenn es für jedes Element $y \in Y$ ein Element $x \in X$ gibt mit $f(x) = y$. Die Abbildung heißt *bijektiv*, wenn sie sowohl injektiv als auch surjektiv ist. Eine bijektive Abbildung heißt auch *Bijektion*. Wenn es eine bijektive Abbildung zwischen zwei endlichen Mengen gibt, so haben beide Mengen dieselbe Anzahl von Elementen.

Beispiel 3.10.8. Betrachte die Abbildung $f : \mathbb{N} \to \mathbb{N}$, $n \mapsto f(n) = n$. Diese Abbildung ist offensichtlich bijektiv.

Betrachte die Abbildung $f : \mathbb{N} \to \mathbb{N}$, $n \mapsto f(n) = n^2$. Da natürliche Zahlen paarweise verschiedene Quadrate haben, ist die Abbildung injektiv. Da z.B. 3 kein Quadrat einer natürlichen Zahl ist, ist die Abbildung nicht surjektiv.

Betrachte die Abbildung $f : \{1, 2, 3, 4, 5, 6\} \to \{0, 1, 2, 3, 4, 5\}$, $n \mapsto f(n) = n \bmod 6$. Da die Urbildmenge ein volles Restsystem modulo 6 ist, ist die Abbildung bijektiv.

Wir beweisen den Satz von Lagrange.

Theorem 3.10.9. *Ist G eine endliche Gruppe, so teilt die Ordnung jeder Untergruppe die Ordnung von G.*

Beweis. Sei H eine Untergruppe von G. Wir sagen, daß zwei Elemente a und b aus G äquivalent sind, wenn $a/b = ab^{-1}$ zu H gehört. Dies ist eine Äquivalenzrelation. Es ist nämlich $a/a = 1 \in H$, daher ist die Relation reflexiv. Außerdem folgt aus $a/b \in H$, daß auch das Inverse b/a zu H gehört, weil H eine Gruppe ist. Daher ist die Relation symmetrisch. Ist schließlich $a/b \in H$ und $b/c \in H$, so ist auch $a/c = (a/b)(b/c) \in H$. Also ist die Relation transitiv.

Wir zeigen, daß die Äquivalenzklassen alle die gleiche Anzahl von Elementen haben. Die Äquivalenzklasse von $a \in G$ ist $\{ha : h \in H\}$. Seien a, b zwei Elemente aus G. Betrachte die Abbildung

$$\{ha : h \in H\} \to \{hb : h \in H\}, ha \mapsto hb.$$

Die Abbildung ist injektiv, weil in G die Kürzungsregel gilt. Die Abbildung ist außerdem offensichtlich surjektiv. Daher haben beide Äquivalenzklassen gleich viele Elemente. Es ist damit gezeigt, daß alle Äquivalenzklassen die gleiche Anzahl von Elementen haben. Eine solche Äquivalenzklasse ist aber die Äquivalenzklasse von 1 und die ist H. Die Anzahl der Elemente in den Äquivalenzklassen ist somit $|H|$. Weil G aber die disjunkte Vereinigung aller Äquivalenzklassen ist, ist $|G|$ ein Vielfaches von $|H|$. □

Definition 3.10.10. *Ist H eine Untergruppe von G, so heißt die natürliche Zahl $|G|/|H|$ der* Index *von H in G.*

3.11 Der kleine Satz von Fermat

Wir formulieren den berühmten kleinen Satz von Fermat.

Theorem 3.11.1. *Wenn* $\gcd(a, m) = 1$ *ist, dann folgt* $a^{\varphi(m)} \equiv 1 \bmod m$.

Dieses Theorem eröffnet zum Beispiel eine neue Methode, prime Restklassen zu invertieren. Es impliziert nämlich, daß aus $\gcd(a, m) = 1$ die Kongruenz

$$a^{\varphi(m)-1} \cdot a \equiv 1 \bmod m$$

folgt. Das bedeutet, daß $a^{\varphi(m)-1} + m\mathbb{Z}$ die inverse Restklasse von $a + m\mathbb{Z}$ ist.

Wir beweisen den kleinen Satz von Fermat in einem allgemeineren Kontext. Dazu sei G eine endliche Gruppe der Ordnung $|G|$, die multiplikativ geschrieben ist, mit neutralem Element 1.

Theorem 3.11.2. *Die Ordnung eines Gruppenelementes teilt die Gruppenordnung.*

Beweis. Die Ordnung eines Gruppenelementes g ist die Ordnung der von g erzeugten Untergruppe. Also folgt die Behauptung aus Theorem 3.10.9. □

Aus diesem Resultat folgern wir eine allgemeine Version des kleinen Satzes von Fermat.

Korollar 3.11.3. *Es gilt $g^{|G|} = 1$ für jedes $g \in G$.*

Beweis. Die Behauptung folgt aus Theorem 3.11.2 und Theorem 3.9.3. □

Da $(\mathbb{Z}/m\mathbb{Z})^*$ eine endliche abelsche Gruppe der Ordnung $\varphi(m)$ ist, folgt Theorem 3.11.1 aus Korollar 3.11.3.

3.12 Schnelle Exponentiation

Theorem 3.11.1 zeigt, daß eine ganze Zahl x mit $x \equiv a^{\varphi(m)-1} \bmod m$ die Kongruenz (3.4) löst. Es ist also nicht nötig, diese Kongruenz durch Anwendung des erweiterten euklidischen Algorithmus zu lösen. Man kann z.B. $x = a^{\varphi(m)-1} \bmod m$ setzen. Soll die neue Methode effizient sein, dann muß man die Potenz schnell berechnen können.

Wir beschreiben jetzt ein Verfahren zur schnellen Berechnung von Potenzen in einem Monoid G. Dieses Verfahren und Varianten davon sind zentral in vielen kryptographischen Algorithmen. Sei $g \in G$ und e eine natürliche Zahl. Sei

$$e = \sum_{i=0}^{k} e_i 2^i.$$

die Binärentwicklung von e. Man beachte, daß die Koeffizienten e_i entweder 0 oder 1 sind. Dann gilt

$$g^e = g^{\sum_{i=0}^{k} e_i 2^i} = \prod_{i=0}^{k} (g^{2^i})^{e_i} = \prod_{0 \le i \le k, e_i = 1} g^{2^i}.$$

Aus dieser Formel gewinnt man die folgende Idee:

1. Berechne die sukzessiven Quadrate g^{2^i}, $0 \leq i \leq k$.
2. Bestimme g^e als Produkt derjenigen g^{2^i}, für die $e_i = 1$ ist.

Beachte dabei

$$g^{2^{i+1}} = (g^{2^i})^2.$$

Daher kann $g^{2^{i+1}}$ aus g^{2^i} mittels einer Quadrierung berechnet werden. Bevor wir das Verfahren präzise beschreiben und analysieren, erläutern wir es an einem Beispiel. Es zeigt sich, daß der Algorithmus viel effizienter ist als die naive Multiplikationsmethode.

Beispiel 3.12.1. Wir wollen 6^{73} mod 100 berechnen. Wir schreiben die Binärentwicklung des Exponenten auf:

$$73 = 1 + 2^3 + 2^6.$$

Dann bestimmen wir die sukzessiven Quadrate 6, $6^2 = 36$, $6^{2^2} = 36^2 \equiv -4 \bmod 100$, $6^{2^3} \equiv 16 \bmod 100$, $6^{2^4} \equiv 16^2 \equiv 56 \bmod 100$, $6^{2^5} \equiv 56^2 \equiv 36 \bmod 100$, $6^{2^6} \equiv -4 \bmod 100$. Also ist $6^{73} \equiv 6 * 6^{2^3} * 6^{2^6} \equiv 6 * 16 * (-4) \equiv 16 \bmod 100$. Wir haben nur 6 Quadrierungen und zwei Multiplikationen in $(\mathbb{Z}/m\mathbb{Z})^*$ verwendet, um das Resultat zu berechnen. Hätten wir 6^{73} mod 100 nur durch Multiplikation berechnet, dann hätten wir dazu 72 Multiplikationen modulo 100 gebraucht.

In Abbildung 3.1 findet man eine Implementierung der schnellen Exponentiation.

```
pow(goupElement base, int exponent, groupElement result)
begin
  result = 1
  while (exponent > 0)
    if (isEven(exponent) == false)
      result = result * base
    base = base*base
    exponent = exponent/2
  end while
end
}
```

Abb. 3.1. Schnelle Exponentiation

Die Implementierung arbeitet so: In der Variablen `result` ist das Resultat gespeichert, soweit es bisher bestimmt ist. In der Variablen `base` sind die sukzessiven Quadrate gespeichert. Die Quadrate werden eins nach dem anderen ausgerechnet und mit dem Resultat multipliziert, wenn das entsprechende Bit 1 ist.

Die Komplexität des Algorithmus gibt folgender Satz an, der leicht verifiziert werden kann.

Theorem 3.12.2. *pow berechnet* $\texttt{base}^{\texttt{exponent}}$ *und benötigt dazu höchstens* size *exponent* $- 1$ *Quadrierungen und Multiplikationen. pow muß nur eine konstante Anzahl von Gruppenelementen speichern.*

Aus Theorem 3.12.2 und Theorem 3.7.1 erhalten wir eine Abschätzung für die Zeit, die die Berechnung von Potenzen in der primen Restklassengruppe mod m benötigt.

Korollar 3.12.3. *Ist* e *eine ganze Zahl und* $a \in \{0, \ldots, m-1\}$*, so erfordert die Berechnung von* $a^e \bmod m$ *Zeit* $O((\text{size}\, e)(\text{size}\, m)^2)$ *und Platz* $O(\text{size}\, e + \text{size}\, m)$.

Exponentiation in der primen Restklassengruppe ist also in polynomieller Zeit möglich. Es gibt Varianten des schnellen Exponentiationsalgorithmus, die in [35] und [57] beschrieben sind. Sie sind in unterschiedlichen Situationen effizienter als die Basisvariante.

3.13 Schnelle Auswertung von Potenzprodukten

Angenommen, G ist eine endliche abelsche Gruppe, $g_1, \ldots, g_k$ sind Elemente von G und $e_1, \ldots, e_k$ sind nicht negative ganze Zahlen. Wir wollen das Potenzprodukt

$$A = \prod_{i=1}^{k} g_i^{e_i}$$

berechnen. Dazu brauchen wir die Binärentwicklung der Exponenten e_i. Sie werden auf gleiche Länge normiert. Die Binärentwicklung von e_i sei

$$b_{i,n-1} b_{i,n-2} \ldots b_{i,0}, \quad 1 \le i \le k.$$

Für wenigstens ein i sei $b_{i,n-1}$ ungleich Null. Für $1 \le i \le k$ und $0 \le j < n$ sei $e_{i,j}$ die ganze Zahl mit Binärentwicklung $b_{i,n-1} b_{i,n-2} \ldots b_{i,j}$. Ferner sei $e_{i,n} = 0$ für $1 \le i \le k$. Dann ist $e_i = e_{i,0}$ für $1 \le i \le k$. Zuletzt setze

$$A_j = \prod_{i=1}^{k} g_i^{e_{i,j}}.$$

Dann ist $A_0 = A$ das gewünschte Potenzprodukt. Wir berechnen iterativ $A_n, A_{n-1}, \ldots, A_0 = A$. Dazu beachten wir, daß

$$e_{i,j} = 2 * e_{i,j+1} + b_{i,j}, \quad 1 \le i \le k, 0 \le j < n$$

ist. Daher ist

$$A_j = A_{j+1}^2 \prod_{i=1}^{k} g_i^{b_{i,j}}, \quad 0 \le j < n.$$

Für alle $\mathbf{b} = (b_1, \ldots, b_k) \in \{0,1\}^k$ wird

$$G_{\mathbf{b}} = \prod_{i=1}^{k} g_i^{b_i}$$

bestimmt. Dann gilt

$$A_j = A_{j+1}^2 G_{(b_{1,j},\ldots,b_{k,j})}, \quad 0 \le j < n.$$

Wir analysieren dieses Verfahren. Die Berechnung aller $G_{\mathbf{b}}$, $\mathbf{b} \in \{0,1\}^k$ erfordert $2^k - 2$ Multiplikationen in G. Sind diese ausgeführt, so werden noch $n-1$ Quadrierungen und Multiplikationen in G benötigt, um A zu berechnen. Damit ist folgendes Resultat bewiesen:

Theorem 3.13.1. *Sei $k \in \mathbb{N}$, $g_i \in G$, $e_i \in \mathbb{Z}_{\ge 0}$, $1 \le i \le k$ und sei n die maximale binäre Länge der e_i. Dann kann man das Potenzprodukt $\prod_{i=1}^{k} g_i^{e_i}$ mittels $2^k + n - 3$ Multiplikationen und $n-1$ Quadrierungen bestimmen.*

Das beschriebene Verfahren ist für den Fall $k = 1$ eine andere Methode der schnellen Exponentiation. Während in der Methode aus Abschnitt 3.12 die Binärentwicklung des Exponenten von rechts nach links abgearbeitet wird, geht man in dieser Methode die Binärentwicklung von links nach rechts durch.

3.14 Berechnung von Elementordnungen

In kryptographischen Anwendungen braucht man häufig Gruppenelemente großer Ordnung. Wir diskutieren das Problem, die Ordnung eines Elementes g einer endlichen Gruppe G zu berechnen bzw. zu überprüfen, ob eine vorgelegte natürliche Zahl die Ordnung von g ist.

Der folgende Satz zeigt, wie die Ordnung von g berechnet werden kann, wenn die Primfaktorzerlegung

$$|G| = \prod_{p||G|} p^{e(p)}$$

der Ordnung von G bekannt ist. Wenn diese Primfaktorzerlegung unbekannt ist, kann man die Ordnung nicht so leicht finden. In der Public-Key-Kryptographie kennt man aber die Gruppenordnung und ihre Faktorisierung oft.

Theorem 3.14.1. *Für jeden Primteiler p von $|G|$ sei $f(p)$ die größte ganze Zahl derart, daß $g^{|G|/p^{f(p)}} = 1$ ist. Dann ist*

$$\operatorname{order} g = \prod_{p||G|} p^{e(p)-f(p)}. \tag{3.5}$$

Beweis. Übung 3.23.22. □

Theorem 3.14.1 kann man unmittelbar in einen Algorithmus verwandeln, der die Ordnung eines Elementes g berechnet.

Beispiel 3.14.2. Sei G die prime Restklassengruppe modulo 101. Ihre Ordnung ist $100 = 2^2 * 5^2$. Also ist

$$e(2) = e(5) = 2.$$

Wir berechnen die Ordnung von $2 + 101\mathbb{Z}$. Dazu berechnen wir zuerst die Zahlen $f(p)$ aus Theorem 3.14.1. Es ist

$$2^{2*5^2} \equiv 2^{50} \equiv -1 \bmod 101.$$

Also ist $f(2) = 0$. Weiter ist

$$2^{2^2*5} \equiv 2^{20} \equiv -6 \bmod 101.$$

Also ist $f(5) = 0$. Insgesamt ist also 100 die Ordnung von $2 + 101\mathbb{Z}$. Das bedeutet, daß $\mathbb{Z}/101\mathbb{Z}$ zyklisch ist und $2+101\mathbb{Z}$ ein Erzeuger dieser Gruppe ist.

Der Algorithmus bestimmt die Zahlen $f(p)$ für alle Primteiler p von $|G|$. Daraus wird dann die Elementordnung berechnet. Die Implementierung wird dem Leser überlassen.

Als nächstes stellen wir das Problem, zu verifizieren, daß eine vorgelegte Zahl die Ordnung eines Elementes g ist. Das braucht man zum Beispiel, wenn man beweisen will, daß ein vorgelegtes Element die Gruppe erzeugt. Folgendes Resultat ist die Grundlage des Algorithmus. Es ist eine unmittelbare Folge von Theorem 3.14.1.

Korollar 3.14.3. *Sei $n \in \mathbb{N}$, und gelte $g^n = 1$ und $g^{n/p} \neq 1$ für jeden Primteiler p von n. Dann ist n die Ordnung von g.*

Ist die Faktorisierung der Ordnung der Gruppe oder sogar der Elementordnung bekannt, so kann man die Elementordnung in Polynomzeit finden bzw. verifizieren. Ist aber keine dieser Faktorisierungen bekannt, so sind diese Aufgaben im allgemeinen wesentlich schwieriger.

Beispiel 3.14.4. Wir behaupten, daß 25 die Ordnung der Restklasse $5+101\mathbb{Z}$ in der primen Restklassengruppe modulo 101 ist. Tatsächlich ist $5^{25} \equiv 1 \bmod 101$ und $5^5 \equiv -6 \bmod 101$. Also folgt die Behauptung aus Korollar 3.14.3.

3.15 Der Chinesische Restsatz

Seien $m_1, \ldots, m_n$ natürliche Zahlen, die paarweise teilerfremd sind, und seien $a_1, \ldots, a_n$ ganze Zahlen. Wir erläutern eine Lösungsmethode für folgende *simultane Kongruenz*

$$x \equiv a_1 \bmod m_1, \quad x \equiv a_2 \bmod m_2, \quad \ldots, \quad x \equiv a_n \bmod m_n. \tag{3.6}$$

Setze

$$m = \prod_{i=1}^{n} m_i, \quad M_i = m/m_i, \quad 1 \leq i \leq n.$$

Wir werden sehen, daß die Lösung der Kongruenz (3.6) modulo m eindeutig ist. Es gilt

$$\gcd(m_i, M_i) = 1, \quad 1 \leq i \leq n,$$

weil die m_i paarweise teilerfremd sind. Wir benutzen den erweiterten euklidischen Algorithmus, um Zahlen $y_i \in \mathbb{Z}$, $1 \leq i \leq n$, zu berechnen mit

$$y_i M_i \equiv 1 \bmod m_i, 1 \leq i \leq n. \tag{3.7}$$

Dann setzen wir

$$x = (\sum_{i=1}^{n} a_i y_i M_i) \bmod m. \tag{3.8}$$

Wir zeigen, daß x eine Lösung der simultanen Kongruenz (3.6) ist. Aus (3.7) folgt

$$a_i y_i M_i \equiv a_i \bmod m_i, \quad 1 \leq i \leq n, \tag{3.9}$$

und weil für $j \neq i$ die Zahl m_i ein Teiler von M_j ist, gilt

$$a_j y_j M_j \equiv 0 \bmod m_i, \quad 1 \leq i, j \leq n, i \neq j. \tag{3.10}$$

Aus (3.8), (3.9) und (3.10) folgt

$$x \equiv a_i y_i M_i + \sum_{j=1, j\neq i}^{n} a_j y_j M_j \equiv a_i \bmod m_i, \quad 1 \leq i \leq n.$$

Also löst x die Kongruenz (3.6).

Beispiel 3.15.1. Wir wollen die simultane Kongruenz

$$x \equiv 2 \bmod 4, \quad x \equiv 1 \bmod 3, \quad x \equiv 0 \bmod 5$$

lösen. Also ist $m_1 = 4$, $m_2 = 3$, $m_3 = 5$, $a_1 = 2$, $a_2 = 1$, $a_3 = 0$. Dann ist $m = 60$, $M_1 = 60/4 = 15$, $M_2 = 60/3 = 20$, $M_3 = 60/5 = 12$. Wir lösen $y_1 M_1 \equiv 1 \bmod m_1$, d.h. $-y_1 \equiv 1 \bmod 4$. Die absolut kleinste Lösung

ist $y_1 = -1$. Wir lösen $y_2 M_2 \equiv 1 \bmod m_2$, d.h. $-y_2 \equiv 1 \bmod 3$. Die absolut kleinste Lösung ist $y_2 = -1$. Schließlich lösen wir $y_3 M_3 \equiv 1 \bmod m_3$, d.h. $2y_3 \equiv 1 \bmod 5$. Die kleinste nicht negative Lösung ist $y_3 = 3$. Daher erhalten wir $x \equiv -2 * 15 - 20 \equiv 10 \bmod 60$. Eine Lösung der simultanen Kongruenz ist $x = 10$.

Man beachte, daß in dem beschriebenen Algorithmus die Zahlen y_i und M_i nicht von den Zahlen a_i abhängen. Sind also die Werte y_i und M_i berechnet, dann kann man (3.8) benutzen, um (3.6) für jede Auswahl von Werten a_i zu lösen. Eine Implementierung findet man in Abbildung 3.2.

```
crtPrecomp(int moduli[], int numberOfModuli, int modulus,
                         int multipliers[])
begin
  int i, m, M, inverse, gcd, y
  modulus = 1;
  for(i = 0; i < numberOfModuli; i=i+1)
    modulus = modulus*moduli[i]
  end for
  for(i = 0; i < numberOfModuli; i=i+1)
    m = moduli[i];
    M = modulus/m;
    xeuclid(M,m,gcd,inverse,y);
    multipliers[i] = inverse*M%modulus;
  end for
  return modulus;
end

crt(int moduli[], int x[], int numberOfModuli, int result)
begin
  int multipliers[number]
  int result = 0
  int modulus, i
  crtPrecomp(moduli, numberOfModuli, modulus, multipliers)
  for(i = 0; i < numberOfModuli; i=i+1)
    result = (result + multipliers[i]*x[i])%modulus;
  end for
end
```

Abb. 3.2. Der chinesische Restalgorithmus

Jetzt wird der *Chinesische Restsatz* formuliert.

Theorem 3.15.2. *Seien $m_1, \ldots, m_n$ paarweise teilerfremde natürliche Zahlen und seien $a_1, \ldots, a_n$ ganze Zahlen. Dann hat die simultane Kongruenz* (3.6) *eine Lösung x, die eindeutig ist mod $m = \prod_{i=1}^{n} m_i$.*

Beweis. Die Existenz wurde schon bewiesen. Also muß noch die Eindeutigkeit gezeigt werden. Zu diesem Zweck seien x und x' zwei solche Lösungen. Dann gilt $x \equiv x' \bmod m_i$, $1 \leq i \leq n$. Weil die Zahlen m_i paarweise teilerfremd sind, folgt $x \equiv x' \bmod m$. □

Der folgende Satz schätzt den Aufwand zur Konstruktion einer Lösung einer simultanen Kongruenz ab.

Theorem 3.15.3. *Das Verfahren zur Lösung der simultanen Kongruenz* (3.6) *kostet Zeit* $O((\operatorname{size} m)^2)$ *und Platz* $O(\operatorname{size} m)$.

Beweis. Die Berechnung von m erfordert nach den Ergebnissen aus Abschnitt 2.5 Zeit $O(\operatorname{size} m \sum_{i=1}^{n} \operatorname{size} m_i) = O((\operatorname{size} m)^2)$. Die Berechnung aller M_i und y_i und des Wertes x kostet dieselbe Zeit. Das folgt ebenfalls aus den Ergebnissen von Abschnitt 2.5 und aus Theorem 2.10.5. Die Platzschranke ist leicht zu verifizieren. □

3.16 Zerlegung des Restklassenrings

Wir benutzen den Chinesischen Restsatz, um den Restklassenring $\mathbb{Z}/m\mathbb{Z}$ zu zerlegen. Diese Zerlegung erlaubt es, anstatt in einem großen Restklassenring $\mathbb{Z}/m\mathbb{Z}$ in vielen kleinen Restklassenringen $\mathbb{Z}/m_i\mathbb{Z}$ zu rechnen. Das ist oft effizienter. Man kann diese Methode zum Beispiel verwenden, um die Entschlüsselung im RSA-Verfahren zu beschleunigen.

Wir definieren das *Produkt von Ringen.*

Definition 3.16.1. *Seien* $R_1, R_2, \ldots, R_n$ *Ringe. Dann ist ihr* direktes Produkt $\prod_{i=1}^{n} R_i$ *definiert als die Menge aller Tupel* $(r_1, r_2, \ldots, r_n) \in R_1 \times \cdots \times R_n$ *zusammen mit komponentenweiser Addition und Multiplikation.*

Man kann leicht verifizieren, daß $R = \prod_{i=1}^{n} R_i$ aus Definition 3.16.1 ein Ring ist. Wenn die R_i kommutative Ringe mit Einselementen e_i, $1 \leq i \leq n$, sind, dann ist R ein kommutativer Ring mit Einselement $(e_1, \ldots, e_n)$.

Das direkte Produkt von Gruppen ist entsprechend definiert.

Beispiel 3.16.2. Sei $R_1 = \mathbb{Z}/2\mathbb{Z}$ und $R_2 = \mathbb{Z}/9\mathbb{Z}$. Dann besteht $R = R_1 \times R_2$ aus allen Paaren $(a+2\mathbb{Z}, b+9\mathbb{Z})$, $0 \leq a < 2, 0 \leq b < 9$. Also hat $R = R_1 \times R_2$ genau 18 Elemente. Das Einselement in R ist $(1 + 2\mathbb{Z}, 1 + 9\mathbb{Z})$.

Wir brauchen auch noch den Begriff des Homomorphismus und des Isomorphismus.

Definition 3.16.3. *Seien* $(X, \perp_1, \ldots, \perp_n)$ *und* $(Y, \top_1, \ldots, \top_n)$ *Mengen mit jeweils* n *inneren Verknüpfungen. Eine Abbildung* $f : X \to Y$ *heißt* Homomorphismus *dieser Strukturen, wenn* $f(a \perp_i b) = f(a) \top_i f(b)$ *gilt für alle* $a, b \in X$ *und* $1 \leq i \leq n$. *Ist die Abbildung bijektiv, so heißt sie* Isomorphismus *dieser Strukturen.*

Wenn man einen Isomorphismus zwischen zwei Ringen kennt, den man in beiden Richtungen leicht berechnen kann, dann lassen sich alle Aufgaben in dem einen Ring auch in dem anderen Ring lösen. Dies bringt oft Effizienzvorteile.

Beispiel 3.16.4. Ist m eine natürliche Zahl, so ist die Abbildung $\mathbb{Z} \to \mathbb{Z}/m\mathbb{Z}$, $a \mapsto a + m\mathbb{Z}$ ein Ringhomomorphismus.

Ist G eine zyklische Gruppe der Ordnung n mit Erzeuger g, so ist $\mathbb{Z}/n\mathbb{Z} \to G$, $e + n\mathbb{Z} \mapsto g^e$ ein Isomorphismus von Gruppen (siehe Übung 3.23.24).

Theorem 3.16.5. *Seien $m_1, \ldots, m_n$ paarweise teilerfremde ganze Zahlen und sei $m = m_1 m_2 \cdots m_n$. Dann ist die Abbildung*

$$\mathbb{Z}/m\mathbb{Z} \to \prod_{i=1}^{n} \mathbb{Z}/m_i\mathbb{Z}, \quad a + m\mathbb{Z} \mapsto (a + m_1\mathbb{Z}, \ldots, a + m_n\mathbb{Z}) \tag{3.11}$$

ein Isomorphismus von Ringen.

Beweis. Beachte zuerst, daß (3.11) wohldefiniert ist. Ist nämlich $a \equiv b \bmod m$, dann folgt $a \equiv b \bmod m_i$ für $1 \leq i \leq n$. Es ist auch leicht, zu verifizieren, daß (3.11) ein Homomorphismus von Ringen ist. Um die Surjektivität zu beweisen, sei $(a_1 + m_1\mathbb{Z}, \ldots, a_n + m_n\mathbb{Z}) \in \prod_{i=1}^{n} \mathbb{Z}/m_i\mathbb{Z}$. Dann folgt aus Theorem 3.15.2, daß dieses Tupel ein Urbild unter (3.11) hat. Die Injektivität folgt aus der Eindeutigkeit in Theorem 3.15.2. □

Theorem 3.16.5 zeigt, daß man Berechnungen in $\mathbb{Z}/m\mathbb{Z}$ auf Berechnungen in $\prod_{i=1}^{n} \mathbb{Z}/m_i\mathbb{Z}$ zurückführen kann. Man verwandelt dazu eine Restklasse mod m in ein Tupel von Restklassen mod m_i, führt die Berechnung auf dem Tupel aus und benutzt den chinesischen Restsatz, um das Ergebnis wieder in eine Restklasse mod m zu verwandeln. Dies ist z.B. effizienter, wenn die Berechnung mod m_i unter Benutzung von Maschinenzahlen ausgeführt werden kann, während die Berechnungen mod m die Verwendung einer Multiprecision-Arithmetik erfordern würden.

3.17 Bestimmung der Eulerschen φ-Funktion

Jetzt leiten wir eine Formel für die Eulersche φ-Funktion her.

Theorem 3.17.1. *Seien $m_1, \ldots, m_n$ paarweise teilerfremde natürliche Zahlen und $m = \prod_{i=1}^{n} m_i$. Dann gilt $\varphi(m) = \varphi(m_1)\varphi(m_2)\cdots\varphi(m_n)$.*

Beweis. Es folgt aus Theorem 3.16.5, daß die Abbildung

$$(\mathbb{Z}/m\mathbb{Z})^* \to \prod_{i=1}^{n} (\mathbb{Z}/m_i\mathbb{Z})^*, a + m\mathbb{Z} \mapsto (a + m_1\mathbb{Z}, \ldots, a + m_n\mathbb{Z}) \tag{3.12}$$

ein Isomorphismus von Gruppen ist. Insbesondere ist die Abbildung bijektiv. Daher ist die Anzahl $\varphi(m)$ der Elemente von $(\mathbb{Z}/m\mathbb{Z})^*$ gleich der Anzahl $\prod_{i=1}^{n} \varphi(m_i)$ der Elemente von $\prod_{i=1}^{n}(\mathbb{Z}/m_i\mathbb{Z})^*$. □

Theorem 3.17.2. *Sei m eine natürliche Zahl und $m = \prod_{p|m} p^{e(p)}$ ihre Primfaktorzerlegung. Dann gilt*

$$\varphi(m) = \prod_{p|m}(p-1)p^{e(p)-1} = m\prod_{p|m}\frac{p-1}{p}.$$

Beweis. Nach Theorem 3.17.1 gilt

$$\varphi(m) = \prod_{p|m}\varphi(p^{e(p)}).$$

Also braucht nur $\varphi(p^e)$ berechnet zu werden, und zwar für eine Primzahl p und eine natürliche Zahl e. Nach Theorem 2.3.3 hat jedes a in der Menge $\{0, 1, 2, \ldots, p^e - 1\}$ eine eindeutige Darstellung

$$a = a_e + a_{e-1}p + a_{e-2}p^2 + \ldots + a_1p^{e-1}$$

mit $a_i \in \{0, 1, \ldots, p-1\}$, $1 \leq i \leq e$. Außerdem gilt genau dann $\gcd(a, p^e) = 1$, wenn $a_e \neq 0$ ist. Dies impliziert, daß

$$\varphi(p^e) = (p-1)p^{e-1} = p^e(1 - \frac{1}{p}).$$

Also ist die Behauptung bewiesen. □

Beispiel 3.17.3. Es gilt $\varphi(2^m) = 2^{m-1}$, $\varphi(100) = \varphi(2^2 * 5^2) = 2 * 4 * 5 = 40$.

Wenn die Faktorisierung von m bekannt ist, kann $\varphi(m)$ gemäß Theorem 3.17.2 in Zeit $O((\text{size}\, m)^2)$ berechnet werden.

3.18 Polynome

Wir wollen in diesem Kapitel noch beweisen, daß für jede Primzahl p die prime Restklassengruppe $(\mathbb{Z}/p\mathbb{Z})^*$ zyklisch von der Ordnung $p-1$ ist. Dazu brauchen wir Polynome, die wir in diesem Abschnitt kurz einführen. Polynome brauchen wir später auch noch, um endliche Körper einzuführen.

Es sei R ein kommutativer Ring mit Einselement $1 \neq 0$. Ein *Polynom* in einer Variablen über R ist ein Ausdruck

$$f(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$$

wobei x die Variable ist und die *Koeffizienten* $a_0, \ldots, a_n$ zu R gehören. Die Menge aller Polynome über R in der Variablen x wird mit $R[x]$ bezeichnet.

Sei $a_n \neq 0$. Dann heißt n der *Grad* des Polynoms. Man schreibt $n = \deg f$. Außerdem heißt a_n der *Leitkoeffizient* oder *führender Koeffizient* von f. Sind alle Koeffizienten außer dem führenden Koeffizienten 0, so heißt f *Monom*.

Beispiel 3.18.1. Die Polynome $2x^3 + x + 1$, x, 1 liegen in $\mathbb{Z}[x]$. Das erste Polynom hat den Grad 3, das zweite den Grad 1 und das dritte den Grad 0.

Ist $r \in R$, so heißt

$$f(r) = a_n r^n + \cdots + a_0$$

der *Wert* von f an der Stelle r. Ist $f(r) = 0$, so heißt r *Nullstelle* von f.

Beispiel 3.18.2. Der Wert des Polynoms $2x^3 + x + 1 \in \mathbb{Z}[x]$ an der Stelle -1 ist -2.

Beispiel 3.18.3. Bezeichne die Elemente von $\mathbb{Z}/2\mathbb{Z}$ mit 0 und 1. Dann ist $x^2 + 1 \in (\mathbb{Z}/2\mathbb{Z})[x]$. Dieses Polynom hat die Nullstelle 1.

Sei

$$g(x) = b_m x^m + \cdots + b_0$$

ein anderes Polynom über R und gelte $n \geq m$. Indem man die fehlenden Koeffizienten auf Null setzt, kann man

$$g(x) = b_n x^n + \cdots + b_0$$

schreiben. Die *Summe* der Polynome f und g ist

$$(f + g)(x) = (a_n + b_n)x^n + \cdots + (a_0 + b_0).$$

Dies ist wieder ein Polynom.

Beispiel 3.18.4. Ist $g(x) = x^2 + x + 1 \in \mathbb{Z}[x]$ und $f(x) = x^3 + 2x^2 + x + 2 \in \mathbb{Z}[x]$, so ist $(f + g)(x) = x^3 + 3x^2 + 2x + 3$.

Die Addition von f und g benötigt $O(\max\{\deg f, \deg g\} + 1)$ Additionen in R.

Das *Produkt* der Polynome f und g ist

$$(fg)(x) = c_{n+m} x^{n+m} + \cdots + c_0$$

wobei

$$c_k = \sum_{i=0}^{k} a_i b_{k-i}, \quad 0 \leq k \leq n + m$$

ist. Auch hierin sind die nicht definierten Koeffizienten a_i und b_i auf 0 gesetzt.

Beispiel 3.18.5. Sei $f(x) = x^2 + x + 1 \in \mathbb{Z}[x]$ und $g(x) = x^3 + 2x^2 + x + 2 \in \mathbb{Z}[x]$. Dann ist $(fg)(x) = (x^2 + x + 1)(x^3 + 2x^2 + x + 2) = x^5 + (2+1)x^4 + (1+2+1)x^3 + (2+1+2)x^2 + (2+1)x + 2 = x^5 + 3x^4 + 4x^3 + 5x^2 + 3x + 2$.

Wir schätzen die Anzahl der Operationen ab, die zur Multiplikation von f und g verwandt werden. Es werden alle Produkte $a_i b_j$, $0 \le i \le \deg f$, $0 \le j \le \deg g$ gebildet. Dies sind $(\deg f + 1)(\deg g + 1)$ Multiplikationen. Dann werden diejenigen Produkte $a_i b_j$ addiert, für die $i + j$ den gleichen Wert hat. Diese Summe bildet den Koeffizienten von x^{i+j}. Da jedes Produkt in genau einer Summe vorkommt, sind dies höchstens $(\deg f + 1)(\deg g + 1)$ Additionen. Insgesamt braucht man also zur Multiplikation von f und g $O((\deg f + 1)(\deg g + 1))$ Additionen und Multiplikationen in R. Schnellere Polynomoperationen mit Hilfe der schnellen Fouriertransformation werden in [3] beschrieben. Siehe auch [42].

Man sieht leicht ein, daß $(R[x], +, \cdot)$ ein kommutativer Ring mit Einselement 1 ist.

3.19 Polynome über Körpern

Sei K ein Körper. Dann ist der Polynomring $K[x]$ nullteilerfrei. Folgende Regel kann man leicht verifizieren.

Lemma 3.19.1. *Sind $f, g \in \mathbb{K}[x]$, $f, g \neq 0$, dann gilt* $\deg(fg) = \deg f + \deg g$.

Wie im Ring der ganzen Zahlen ist im Polynomring $K[x]$ die Division mit Rest möglich.

Theorem 3.19.2. *Seien $f, g \in K[x]$, $g \neq 0$. Dann gibt es eindeutig bestimmte Polynome $q, r \in K[x]$ mit $f = qg + r$ und $r = 0$ oder* $\deg r < \deg g$.

Beweis. Ist $f = 0$, so setze $q = r = 0$. Sei also $f \neq 0$. Ist $\deg g > \deg f$, so setze $q = 0$ und $r = f$. Wir nehmen also weiter an, daß $\deg g \le \deg f$ ist.

Wir beweisen die Existenz von q und r durch Induktion über den Grad von f.

Ist $\deg f = 0$, dann ist $\deg g = 0$. Also sind $f, g \in K$ und man kann $q = f/g$ und $r = 0$ setzen.

Sei $\deg f = n > 0$, $\deg g = m$, $n \ge m$ und

$$f(x) = a_n x^n + \cdots + a_0, \quad g(x) = b_m x^m + \cdots + b_0.$$

Setze

$$f_1 = f - a_n / b_m x^{n-m} g.$$

Entweder ist $f_1 = 0$ oder $\deg f_1 < \deg f$. Nach Induktionsvoraussetzung gibt es Polynome q_1 und r mit $f_1 = q_1 g + r$ und $r = 0$ oder $\deg r < \deg g$. Daraus folgt

$$f = (a_n/b_m x^{n-m} + q_1)g + r.$$

Die Polynome $q = a_n/b_m x^{n-m} + q_1$ und r von oben erfüllen die Behauptung.

Jetzt beweisen wir noch die Eindeutigkeit. Seien $f = qg + r = q'g + r'$ zwei Darstellungen wie im Satz. Dann ist $(q - q')g = r' - r$. Ist $r = r'$, so ist $q = q'$, weil $g \neq 0$ und $K[x]$ nullteilerfrei ist. Ist $r \neq r'$, so ist $q - q' \neq 0$ und wegen $\deg g > \deg r$ und $\deg g > \deg r'$ gilt nach Lemma 3.19.1 auch $\deg(q - q')g > \deg(r' - r)$. Dies kann aber nicht sein, weil $(q - q')g = r' - r$ ist. □

In der Situation von Theorem 3.19.2 nennt man q den *Quotienten* und r den *Rest* der Division von f durch g und man schreibt $r = f \bmod g$.

Aus dem Beweis von Theorem 3.19.2 erhält man einen Algorithmus, der es ermöglicht, ein Polynom f durch ein anderes Polynom g mit Rest zu dividieren. Man setzt zuerst $r = f$ und $q = 0$. Solange $r \neq 0$ und $\deg r \geq \deg g$ ist, setzt man $h(x) = (a/b)x^{\deg r - \deg g}$, wobei a der höchste Koeffizient von r und b der höchste Koeffizient von g ist. Dann ersetzt man r durch $r - hg$ und q durch $q + h$. Sobald $r = 0$ oder $\deg r < \deg g$ ist, gibt man den Quotienten q und den Rest r aus. Dies wird im folgenden Beispiel illustriert.

Beispiel 3.19.3. Sei $K = \mathbb{Z}/2\mathbb{Z}$ der Restklassenring mod 2. Dieser Ring ist ein Körper. Die Elemente werden durch ihre kleinsten nicht negativen Vertreter dargestellt. Wir schreiben also $\mathbb{Z}/2\mathbb{Z} = \{0, 1\}$.

Sei

$$f(x) = x^3 + x + 1, \quad g(x) = x^2 + x.$$

Wir dividieren f mit Rest durch g. Wir setzen also zuerst $r = f$ und $q = 0$. Dann eliminieren wir x^3 in r. Wir setzen $h(x) = x$ und ersetzen r durch $r - hg = x^3 + x + 1 - x(x^2 + x) = x^2 + x + 1$ und q durch $q + h = x$. Danach ist $\deg r = \deg g$. Der Algorithmus benötigt also noch eine Iteration. Wieder eliminieren wir den höchsten Koeffizienten in r. Dazu setzen wir $h(x) = 1$ und ersetzen r durch $r - hg = 1$ und q durch $q + h = x + 1$. Da nun $0 = \deg r < \deg g = 2$ ist, sind wir fertig und haben den Quotienten $q = x + 1$ und den Rest $r = 1$ berechnet.

Wir schätzen die Anzahl der Operationen in K ab, die man für die Division mit Rest von f durch g braucht. Die Berechnung eines Monoms h erfordert eine Operation in K. Die Anzahl der Monome h ist höchstens $\deg q + 1$, weil deren Grade streng monoton fallen und ihre Summe gerade q ist. Jedesmal, wenn h berechnet ist, muß man $r - hg$ berechnen. Die Berechnung von hg erfordert $\deg g + 1$ Multiplikationen in K. Der Grad der Polynome r und hg ist derselbe und die Anzahl der von Null verschiedenen Koeffizienten in hg ist höchstens $\deg g + 1$. Daher erfordert die Berechnung von $r - hg$ höchstens $\deg g + 1$ Additionen in K. Insgesamt erfordert die Division mit Rest höchstens $O((\deg g + 1)(\deg q + 1))$ Operationen in K.

Theorem 3.19.4. *Sind $f, g \in K[x]$ mit $g \neq 0$, so kann man f mit Rest durch g unter Verwendung von $O((\deg g + 1)(\deg q + 1))$ Operationen in K dividieren, wobei q der Quotient der Division ist.*

Aus Theorem 3.19.2 erhält man folgende Konsequenzen.

Korollar 3.19.5. *Ist f ein von Null verschiedenes Polynom in $K[x]$ und ist a eine Nullstelle von f, dann ist $f = (x - a)q$ mit $q \in K[x]$, d.h. f ist durch das Polynom $x - a$ teilbar.*

Beweis. Nach Theorem 3.19.2 gibt es Polynome $q, r \in K[x]$ mit $f = (x - a)q + r$ und $r = 0$ oder $\deg r < 1$. Daraus folgt $0 = f(a) = r$, also $f = (x-a)q$. □

Beispiel 3.19.6. Das Polynom $x^2 + 1 \in (\mathbb{Z}/2\mathbb{Z})[x]$ hat die Nullstelle 1 und es gilt $x^2 + 1 = (x - 1)^2$.

Korollar 3.19.7. *Ein Polynom $f \in K[x]$, $f \neq 0$, hat höchstens* $\deg f$ *viele Nullstellen.*

Beweis. Wir beweisen die Behauptung durch Induktion über $n = \deg f$. Für $n = 0$ ist die Behauptung wahr, weil $f \in K$ und $f \neq 0$ ist. Sei $n > 0$. Wenn f keine Nullstelle hat, dann ist die Behauptung wahr. Wenn f aber eine Nullstelle a hat, dann gilt nach Korollar 3.19.5 $f = (x-a)q$ und $\deg q = n-1$. Nach Induktionsvoraussetzung hat q höchstens $n - 1$ Nullstellen und darum hat f höchstens n Nullstellen. □

Wir zeigen in folgendem Beispiel, daß Korollar 3.19.7 wirklich nur eine obere Schranke liefert, die keineswegs immer angenommen wird.

Beispiel 3.19.8. Das Polynom $x^2 + x \in (\mathbb{Z}/2\mathbb{Z})[x]$ hat die Nullstellen 0 und 1 in $\mathbb{Z}/2\mathbb{Z}$. Mehr Nullstellen kann es auch nach Korollar 3.19.7 nicht haben.

Das Polynom $x^2 + 1 \in (\mathbb{Z}/2\mathbb{Z})[x]$ hat die einzige Nullstelle 1 in $\mathbb{Z}/2\mathbb{Z}$. Nach Korollar 3.19.7 könnte es aber 2 Nullstellen haben.

Das Polynom $x^2 + x + 1 \in (\mathbb{Z}/2\mathbb{Z})[x]$ hat keine Nullstellen in $\mathbb{Z}/2\mathbb{Z}$. Nach Korollar 3.19.7 könnte es aber 2 Nullstellen haben.

3.20 Konstruktion endlicher Körper

In diesem Abschnitt beschreiben wir, wie man zu jeder Primzahl p und jeder natürlichen Zahl n einen endlichen Körper mit p^n Elementen konstruieren kann. Dieser Körper ist bis auf Isomorphie eindeutig bestimmt und wird mit $\mathrm{GF}(p^n)$ bezeichnet. Die Abkürzung GF steht für *galois field*. Das ist die englische Bezeichnung für endliche Körper. Aus Theorem 3.6.4 wissen wir bereits, daß $\mathbb{Z}/p\mathbb{Z}$ ein Körper mit p Elementen ist. Er wird mit $\mathrm{GF}(p)$ bezeichnet. Die Primzahl p heißt *Charakteristik* des Körpers $\mathrm{GF}(p^n)$. Der Körper $\mathrm{GF}(p)$

heißt *Primkörper*. Die Konstruktion ist mit der Konstruktion des Körpers $\mathbb{Z}/p\mathbb{Z}$ für eine Primzahl p eng verwandt. Wir werden die Konstruktion nur skizzieren. Details und Beweise findet man z.B. in [56].

Sei p eine Primzahl, sei n eine natürliche Zahl und sei f ein Polynom mit Koeffizienten in $\mathbb{Z}/p\mathbb{Z}$ vom Grad n. Das Polynom muß *irreduzibel* sein, d.h. es darf nicht als Produkt $f = gh$ geschrieben werden können, wobei g und h Polynome in $(\mathbb{Z}/p\mathbb{Z})[X]$ sind, deren Grad größer als Null ist. Polynome, die nicht irreduzibel sind heißen *reduzibel*.

Beispiel 3.20.1. Sei $p = 2$.

Das Polynom $f(X) = X^2 + X + 1$ ist irreduzibel in $(\mathbb{Z}/2\mathbb{Z})[X]$. Wäre f reduzibel, müßte f nach Lemma 3.19.1 Produkt von zwei Polynomen vom Grad eins aus $(\mathbb{Z}/2\mathbb{Z})[X]$ sein. Dann hätte f also eine Nullstelle in $\mathbb{Z}/2\mathbb{Z}$. Es ist aber $f(0) \equiv f(1) \equiv 1 \bmod 2$. Also ist f irreduzibel.

Das Polynom $f(X) = X^2 + 1$ ist reduzibel in $(\mathbb{Z}/2\mathbb{Z})[X]$, denn es gilt $X^2 + 1 \equiv (X + 1)^2 \bmod 2$.

Die Elemente des endlichen Körpers, der nun konstruiert wird, sind Restklassen mod f. Die Konstruktion dieser Restklassen entspricht der Konstruktion von Restklassen in $\mathbb{Z}$. Die Restklasse des Polynoms $g \in (\mathbb{Z}/p\mathbb{Z})[X]$ besteht aus allen Polynomen h in $(\mathbb{Z}/p\mathbb{Z})[X]$, die sich von g nur durch ein Vielfaches von f unterscheiden, für die also $g - h$ durch f teilbar ist. Wir schreiben $g + f(\mathbb{Z}/p\mathbb{Z})[X]$ für diese Restklasse, denn es gilt

$$g + f(\mathbb{Z}/p\mathbb{Z})[X] = \{g + hf : h \in (\mathbb{Z}/p\mathbb{Z})[X]\}.$$

Nach Theorem 3.19.2 gibt es in jeder Restklasse mod f einen eindeutig bestimmten Vertreter, der entweder Null ist oder dessen Grad kleiner als der Grad von f ist. Diesen Vertreter kann man durch Division mit Rest bestimmen. Will man also feststellen, ob die Restklassen zweier Polynome gleich sind, so kann man jeweils diesen Vertreter berechnen und vergleichen. Sind sie gleich, so sind die Restklassen gleich. Sind sie verschieden, so sind die Restklassen verschieden.

Die Anzahl der verschiedenen Restklassen mod f ist p^n. Das liegt daran, daß die Restklassen aller Polynome, deren Grad kleiner n ist, paarweise verschieden sind und daß jede Restklasse mod f einen Vertreter enthält, dessen Grad kleiner als n ist.

Beispiel 3.20.2. Die Restklassen in $(\mathbb{Z}/2\mathbb{Z})[X]$ mod $f(X) = X^2 + X + 1$ sind $f(\mathbb{Z}/2\mathbb{Z})$, $1 + f(\mathbb{Z}/2\mathbb{Z})$, $X + f(\mathbb{Z}/2\mathbb{Z})$, $X + 1 + f(\mathbb{Z}/2\mathbb{Z})$.

Sind $g, h \in (\mathbb{Z}/p\mathbb{Z})[X]$, dann ist die Summe der Restklassen von g und h mod f definiert als die Restklasse von $g + h$. Das Produkt der Restklassen von g und h ist die Restklasse des Produkts von g und h. Mit dieser Addition und Multiplikation ist die Menge der Restklassen mod f ein kommutativer Ring mit Einselement $1 + f(\mathbb{Z}/p\mathbb{Z})[X]$.

Beispiel 3.20.3. Sei $p = 2$ und $f(X) = X^2 + X + 1$.

Die Restklassen mod f sind die Restklassen der Polynome 0, 1, X und $X + 1$ mod f. In Tabelle 3.2 und Tabelle 3.3 geben wir die Additions- und Multiplikationstabelle dieser Restklassen an. Dabei bezeichnet α die Restklasse $X + f(\mathbb{Z}/2\mathbb{Z})[X]$. Man beachte, daß α eine Nullstelle von f in GF(4) ist, also $\alpha^2 + \alpha + 1 = 0$ gilt.

$+$	0	1	α	$\alpha+1$
0	0	1	α	$\alpha+1$
1	1	0	$\alpha+1$	α
α	α	$\alpha+1$	0	1
$\alpha+1$	$\alpha+1$	α	1	0

Tabelle 3.2. Addition in GF(4)

$*$	1	α	$\alpha+1$
1	1	α	$\alpha+1$
α	α	$\alpha+1$	1
$\alpha+1$	$\alpha+1$	1	α

Tabelle 3.3. Multiplikation in GF(4)

Weil f irreduzibel ist, ist der Restklassenring mod f sogar ein Körper. In Beispiel 3.20.3 sieht man, daß alle von Null verschiedenen Restklassen mod f ein multiplikatives Inverses besitzen. Das ist auch allgemein richtig. Soll die Restklasse eines Polynoms $g \in (\mathbb{Z}/p\mathbb{Z})[X]$ invertiert werden, verwendet man ein Analogon des erweiterten euklidischen Algorithmus, um ein Polynom $r \in (\mathbb{Z}/p\mathbb{Z})[X]$ zu bestimmen, das $gr + fs = 1$ für ein Polynom $s \in (\mathbb{Z}/p\mathbb{Z})[X]$ erfüllt. Dann ist die Restklasse von r das Inverse der Restklasse von g. Das geht also genauso, wie Invertieren in $\mathbb{Z}/p\mathbb{Z}$. Ist f nicht irreduzibel, so kann man nicht alle von Null verschiedenen Restklassen invertieren. Man erhält dann durch die beschriebene Konstruktion einen Ring, der i.a. nicht nullteilerfrei ist.

Beispiel 3.20.4. Sei $p = 2$ und sei $f(X) = x^8 + x^4 + x^3 + x + 1$. Dieses Polynom ist irrduzibel in $(\mathbb{Z}/2\mathbb{Z})[X]$ (siehe Übung 3.23.26). Sei α die Restklasse von X mod f. Wir bestimmen das Inverse von $\alpha + 1$. Hierzu wenden wir den erweiterten euklidischen Algorithmus an. Es gilt

$$f(X) = (X + 1)q(X) + 1$$

mit

$$q(X) = X^7 + X^6 + X^5 + X^4 + X^2 + X.$$

Wie in Beispiel 2.9.2 bekommt man folgende Tabelle

k	0	1	2	3
r_k	f	$X+1$	1	0
q_k		$q(X)$	$X+1$	
x_k	1	0	1	$X^8 + X^4 + X^3$
y_k	0	1	$q(X)$	$X \cdot q(X)$

Es gilt also

$$f(X) - q(X)(X+1) = 1.$$

Daher ist die Restklasse von $q(X)$, also $\alpha^7+\alpha^6+\alpha^5+\alpha^4+\alpha^2+\alpha$, das Inverse von $\alpha + 1$.

Konstruiert man auf diese Weise Körper für verschiedene Polynome vom Grad n, so sind diese Körper isomorph, also nicht wirklich verschieden.

Da es für jede natürliche Zahl n ein irreduzibles Polynom in $(\mathbb{Z}/p\mathbb{Z})[X]$ vom Grad n gibt, existiert auch der Körper $\mathrm{GF}(p^n)$ für alle p und n.

3.21 Struktur der Einheitengruppe endlicher Körper

Wir untersuchen jetzt die Einheitengruppe K^* eines endlichen Körpers K, also eines Körpers mit endlich vielen Elementen. Wir beweisen, daß diese Gruppe immer zyklisch ist. Daher ist sie für die Kryptographie besonders interessant, weil dort Gruppen mit Elementen hoher Ordnung benötigt werden. Wir kennen bereits die endlichen Körper $\mathbb{Z}/p\mathbb{Z}$ für Primzahlen p. Ihre Einheitengruppe hat die Ordnung $p-1$. Später werden wir noch andere endliche Körper kennenlernen.

Allgemein hat die Einheitengruppe K^* eines Körpers K mit q Elementen die Ordnung $q-1$, weil alle Elemente außer der Null Einheiten sind.

Theorem 3.21.1. *Sei K ein endlicher Körper mit q Elementen. Dann gibt es für jeden Teiler d von $q-1$ genau $\varphi(d)$ Elemente der Ordnung d in der Einheitengruppe K^*.*

Beweis. Sei d ein Teiler von $q-1$. Bezeichne mit $\psi(d)$ die Anzahl der Elemente der Ordnung d in K^*.

Angenommen, $\psi(d) > 0$. Wir zeigen, daß unter dieser Voraussetzung $\psi(d) = \varphi(d)$ gilt. Später beweisen wir dann, daß tatsächlich $\psi(d) > 0$ gilt. Sei a ein Element der Ordnung d in K^*. Die Potenzen a^e, $0 \le e < d$, sind paarweise verschieden und alle Nullstellen des Polynoms $x^d - 1$. Im Körper K gibt es nach Korollar 3.19.7 höchstens d Nullstellen dieses Polynoms. Das Polynom besitzt also genau d Nullstellen und sie sind die Potenzen von a.

Nun ist aber jedes Element von K der Ordnung d eine Nullstelle von $x^d - 1$ und daher eine Potenz von a. Aus Theorem 3.9.5 folgt, daß a^e genau dann die Ordnung d hat, wenn $\gcd(d, e) = 1$ ist. Also haben wir bewiesen, daß aus $\psi(d) > 0$ folgt, daß $\psi(d) = \varphi(d)$ ist.

Wir zeigen nun, daß $\psi(d) > 0$ ist. Angenommen, $\psi(d) = 0$ für einen Teiler d von $q - 1$. Dann gilt

$$q - 1 = \sum_{d|q-1} \psi(d) < \sum_{d|q-1} \varphi(d).$$

Dies widerspricht Theorem 3.8.4. □

Beispiel 3.21.2. Betrachte den Körper $\mathbb{Z}/13\mathbb{Z}$. Seine Einheitengruppe hat die Ordnung 12. In dieser Gruppe gibt es ein Element der Ordnung 1, ein Element der Ordnung 2, zwei Elemente der Ordnung 3, zwei Elemente der Ordnung 4, zwei Elemente der Ordnung 6 und vier Elemente der Ordnung 12. Insbesondere ist diese Gruppe also zyklisch und hat vier Erzeuger.

Ist K ein endlicher Körper mit q Elementen, so gibt es nach Theorem 3.21.1 genau $\varphi(q - 1)$ Elemente der Ordnung $q - 1$. Daraus ergibt sich folgendes:

Korollar 3.21.3. *Ist K ein endlicher Körper mit q Elementen, so ist die Einheitengruppe K^* zyklisch von der Ordnung $q - 1$. Sie hat genau $\varphi(q - 1)$ Erzeuger.*

3.22 Struktur der primen Restklassengruppe nach einer Primzahl

Sei p eine Primzahl. In Korollar 3.21.3 haben wir folgendes Resultat bewiesen:

Korollar 3.22.1. *Die prime Restklassengruppe mod p ist zyklisch von der Ordnung $p - 1$.*

Eine ganze Zahl a, für die die Restklasse $a + p\mathbb{Z}$ die prime Restklassengruppe $(\mathbb{Z}/p\mathbb{Z})^*$ erzeugt, heißt *Primitivwurzel* mod p .

Beispiel 3.22.2. Für $p = 13$ ist $p - 1 = 12$. Aus Theorem 3.17.2 folgt, daß $\varphi(12) = 4$. Also gibt es vier Primitivwurzeln mod 13, nämlich $2, 6, 7$ und 11.

Wir diskutieren die Berechnung von Primitivwurzeln modulo einer Primzahl p. Wir haben in Theorem 3.21.3 gesehen, daß es $\varphi(p-1)$ Primitivwurzeln mod p gibt. Nun gilt

$$\varphi(n) \geq n/(6 \ln \ln n)$$

für jede natürliche Zahl $n \geq 5$ (siehe [66]). Der Beweis dieser Ungleichung sprengt den Rahmen dieses Buches. Also ist die Anzahl der Erzeuger einer

zyklischen Gruppe der Ordnung n wenigstens $\lceil n/(6 \ln \ln n) \rceil$. Wenn $n = 2 * q$ mit einer Primzahl q ist, dann ist die Anzahl der Erzeuger sogar $q - 1$. Fast die Hälfte aller Gruppenelemente erzeugen also die Gruppe. Wenn man also zufällig eine natürliche Zahl g mit $1 \leq g \leq p - 1$ wählt, hat man eine gute Chance, eine Primitivwurzel mod p zu finden. Das Problem ist nur, zu verifizieren, daß man tatsächlich eine solche Primitivwurzel gefunden hat. Aus Korollar 3.14.3 kennen wir ein effizientes Verfahren, um zu überprüfen, ob g eine Primitivwurzel mod p ist, wenn wir $p - 1$ faktorisieren können. In dem besonders einfachen Fall $p - 1 = 2q$ mit einer Primzahl q brauchen wir nur zu testen, ob $g^2 \equiv 1 \bmod p$ oder $g^q \equiv 1 \bmod p$ ist. Wenn diese beiden Kongruenzen nicht erfüllt sind, ist g eine Primitivwurzel mod p.

Beispiel 3.22.3. Sei $p = 23$. Dann ist $p - 1 = 22 = 11 * 2$. Um zu prüfen, ob eine ganze Zahl g eine Primitivwurzel modulo 23 ist, muß man verifizieren, daß $g^2 \bmod 23 \neq 1$ ist und daß $g^{11} \bmod 23 \neq 1$ ist. Hier ist eine Tabelle mit den entsprechenden Resten für die Primzahlen zwischen 2 und 17.

g	2	3	5	7	11	13	17
$g^2 \bmod 23$	4	9	2	3	6	8	13
$g^{11} \bmod 23$	1	1	-1	-1	-1	1	-1

Es zeigt sich, daß 5, 7, 11, 17 Primitivwurzeln mod 23 sind und daß 2, 3, 13 keine Primitivwurzeln mod 23 sind.

3.23 Übungen

Übung 3.23.1. Beweisen Sie die Potenzgesetze für Halbgruppen und Gruppen.

Übung 3.23.2. Bestimmen Sie alle Halbgruppen, die man durch Definition einer Operation auf $\{0, 1\}$ erhält.

Übung 3.23.3. Zeigen Sie, daß es in einer Halbgruppe höchstens ein neutrales Element geben kann.

Übung 3.23.4. Welche der Halbgruppen aus Übung 3.23.2 sind Monoide? Welche sind Gruppen?

Übung 3.23.5. Zeigen Sie, daß in einem Monoid jedes Element höchstens ein Inverses haben kann.

Übung 3.23.6. Sei n ein positiver Teiler einer positiven Zahl m. Beweisen Sie, daß die Abbildung $\mathbb{Z}/m\mathbb{Z} \to \mathbb{Z}/n\mathbb{Z}$, $a + m\mathbb{Z} \mapsto a + n\mathbb{Z}$ ein surjektiver Ringhomomorphismus ist.

Übung 3.23.7. Zeigen Sie an einem Beispiel, daß die Kürzungsregel in der Halbgruppe $(\mathbb{Z}/m\mathbb{Z}, \cdot)$ im allgemeinen nicht gilt.

Übung 3.23.8. Bestimmen Sie die Einheitengruppe und die Nullteiler des Rings $\mathbb{Z}/16\mathbb{Z}$.

Übung 3.23.9. Zeigen Sie, daß die invertierbaren Elemente eines kommutativen Rings mit Einselement eine Gruppe bilden.

Übung 3.23.10. Lösen Sie $122x \equiv 1 \bmod 343$.

Übung 3.23.11. Beweisen Sie, daß die Kongruenz $ax \equiv b \bmod m$ genau dann lösbar ist, wenn $\gcd(a, m)$ ein Teiler von b ist. Im Falle der Lösbarkeit bestimmen Sie alle Lösungen.

Übung 3.23.12. Sei $d_1 d_2 \dots d_k$ die Dezimalentwicklung einer positiven ganzen Zahl d. Beweisen Sie, daß d genau dann durch 11 teilbar ist, wenn $\sum_{i=1}^{k} (-1)^{k-i}$ durch 11 teilbar ist.

Übung 3.23.13. Bestimmen Sie alle invertierbaren Restklassen modulo 25 und berechnen Sie alle Inverse.

Übung 3.23.14. Das kleinste gemeinsame Vielfache zweier von Null verschiedener ganzer Zahlen a, b ist die kleinste natürliche Zahl k, die sowohl ein Vielfaches von a als auch ein Vielfaches von b ist. Es wird mit $\operatorname{lcm}(a, b)$ bezeichnet. Dabei steht lcm für least common multiple.

1. Beweisen Sie Existenz und Eindeutigkeit von $\operatorname{lcm}(a, b)$.
2. Wie kann $\operatorname{lcm}(a, b)$ mit dem euklidischen Algorithmus berechnet werden?

Übung 3.23.15. Seien X und Y endliche Mengen und $f : X \to Y$ eine Bijektion. Zeigen Sie, daß X und Y gleich viele Elemente besitzen.

Übung 3.23.16. Berechnen Sie die von $2 + 17\mathbb{Z}$ in $(\mathbb{Z}/17\mathbb{Z})^*$ erzeugte Untergruppe.

Übung 3.23.17. Berechnen Sie die Ordnung von 2 mod 1237.

Übung 3.23.18. Bestimmen Sie die Ordnung aller Elemente in $(\mathbb{Z}/15\mathbb{Z})^*$.

Übung 3.23.19. Berechnen Sie $2^{20} \bmod 7$.

Übung 3.23.20. Sei G eine endliche zyklische Gruppe. Zeigen Sie, daß es für jeden Teiler d von $|G|$ genau eine Untergruppe von G der Ordnung d gibt.

Übung 3.23.21. Sei p eine Primzahl, $p \equiv 3 \bmod 4$. Sei a eine ganze Zahl, die ein Quadrat mod p ist (d.h., die Kongruenz $a \equiv b^2 \bmod p$ hat eine Lösung). Zeigen Sie, daß $a^{(p+1)/4}$ eine Quadratwurzel von a mod p ist.

Übung 3.23.22. Beweisen Sie Theorem 3.14.1.

Übung 3.23.23. Konstruieren Sie ein Element der Ordnung 103 in der primen Restklassengruppe mod 1237.

Übung 3.23.24. Sei G eine zyklische Gruppe der Ordnung n mit Erzeuger g. Zeigen Sie, daß $\mathbb{Z}/n\mathbb{Z} \to G$, $e+n\mathbb{Z} \mapsto g^e$ ein Isomorphismus von Gruppen ist.

Übung 3.23.25. Lösen Sie die simultane Kongruenz $x \equiv 1 \bmod p$ für alle $p \in \{2, 3, 5, 7\}$.

Übung 3.23.26. Zeigen Sie, daß das Polynom $f(X) = x^8 + x^4 + x^3 + x + 1$ in $(\mathbb{Z}/2\mathbb{Z})[X]$ irreduzibel ist.

Übung 3.23.27. Bestimmen Sie für $g = 2, 3, 5, 7, 11$ jeweils eine Primzahl $p > g$ mit der Eigenschaft, daß g eine Primitivwurzel mod p ist.

Übung 3.23.28. Finden Sie alle primen Restklassengruppen mit vier Elementen.

4. Verschlüsselung

Der klassische Gegenstand der Kryptographie sind Verschlüsselungsverfahren. Solche Verfahren braucht man, wenn man Nachrichten oder gespeicherte Daten geheimhalten will. In diesem Kapitel führen wir Grundbegriffe ein, die die Beschreibung von Verschlüsselungsverfahren ermöglichen. Als erstes Beispiel besprechen wir affin lineare Chiffren und ihre Kryptoanalyse.

4.1 Verschlüsselungsverfahren

Wir definieren Verschlüsselungsverfahren.

Definition 4.1.1. *Ein* Verschlüsselungsverfahren *oder* Kryptosystem *ist ein Fünftupel* $(\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D})$ *mit folgenden Eigenschaften:*

1. $\mathcal{P}$ *ist eine Menge. Sie heißt* Klartextraum. *Ihre Elemente heißen* Klartexte.
2. $\mathcal{C}$ *ist eine Menge. Sie heißt* Chiffretextraum. *Ihre Elemente heißen* Chiffretexte *oder* Schlüsseltexte.
3. $\mathcal{K}$ *ist eine Menge. Sie heißt* Schlüsselraum. *Ihre Elemente heißen* Schlüssel.
4. $\mathcal{E} = \{E_k : k \in \mathcal{K}\}$ *ist eine Familie von Funktionen* $E_k : \mathcal{P} \to \mathcal{C}$. *Ihre Elemente heißen* Verschlüsselungsfunktionen.
5. $\mathcal{D} = \{D_k : k \in \mathcal{K}\}$ *ist eine Familie von Funktionen* $D_k : \mathcal{C} \to \mathcal{P}$. *Ihre Elemente heißen* Entschlüsselungsfunktionen.
6. *Für jedes* $e \in \mathcal{K}$ *gibt es ein* $d \in \mathcal{K}$, *so daß für alle* $p \in \mathcal{P}$ *die Gleichung* $D_d(E_e(p)) = p$ *gilt.*

Die gewählten Bezeichnungen entsprechen den entsprechenden englischen Wörtern: $\mathcal{P}$ wie plaintext, $\mathcal{C}$ wie ciphertext, $\mathcal{K}$ wie key, $\mathcal{E}$ wie encryption, $\mathcal{D}$ wie decryption.

Alice kann ein Verschlüsslungsverfahren benutzen, um eine Nachricht m vertraulich an Bob zu schicken. Sie benötigt dazu einen Verschlüsselungsschlüssel e. Bob braucht den entsprechenden Entschlüsslungsschlüssel d. Alice berechnet den Schlüsseltext $c = E_e(m)$ und schickt ihn an Bob. Bob kann dann den Klartext $m = D_d(c)$ rekonstruieren. Der Entschlüsselungsschlüssel muß natürlich geheim gehalten werden.

Als erstes Beispiel für ein Verschlüsselungsverfahren beschreiben wir die *Verschiebungschiffre*.

Der Klartext-, Chiffretext- und Schlüsselraum ist $\Sigma = \{A, B, \ldots, Z\}$. Wir identifizieren die Buchstaben $A, B, \ldots, Z$ gemäß Tabelle 4.1 mit den Zahlen $0, 1, \ldots, 25$:

A	B	C	D	E	F	G	H	I	J	K	L	M
0	1	2	3	4	5	6	7	8	9	10	11	12
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
13	14	15	16	17	18	19	20	21	22	23	24	25

Tabelle 4.1. Entsprechung von Buchstaben und Zahlen

Diese Identifikation ermöglicht es, mit Buchstaben wie mit Zahlen zu rechnen. Für $e \in \mathbb{Z}_{26}$ ist die Verschlüsselungsfunktion E_e definiert als

$$E_e : \Sigma \to \Sigma, \quad x \mapsto (x + e) \bmod 26.$$

Entsprechend ist für $d \in \mathbb{Z}_{26}$

$$D_d : \Sigma \times \Sigma \to \Sigma, \quad x \mapsto (x - d) \bmod 26.$$

Der Entschlüsselungsschlüssel zum Verschlüsselungsschlüssel e ist also $d = e$. Das ist aber nicht immer so.

Aus der Verschiebungschiffre kann man leicht ein Verschlüsselungsverfahren machen, dessen Klartext- und Chiffretextraum die Menge aller Folgen $\mathbf{w} = (w_1, w_2, \ldots, w_k)$ mit Gliedern w_i aus Σ, $1 \leq i \leq k$, ist. Der Schlüsselraum ist wieder $\mathbb{Z}_{26}$. Die Verschlüsselungsfunktion E_e ersetzt jeden Buchstaben w_i durch $w_i + e + n \bmod 26$, $1 \leq i \leq k$. Dies nennt man ebenfalls Verschiebungschiffre.

Beispiel 4.1.2. Wendet man die Verschiebungschiffre mit Schlüssel 5 auf das Wort KRYPTOGRAPHIE an, so erhält man PWDUYTLWFUMNJ.

In der Verschiebungschiffre gibt es nur 26 Schlüssel. Das sind sehr wenige mögliche Schlüssel. Wurde ein Text mit der Verschiebungschiffre verschlüsselt, so kann man aus dem Chiffretext den Klartext gewinnen, indem man alle möglichen Schlüssel ausprobiert und prüft, welcher Schlüssel einen sinnvollen Text ergibt. Man erhält auf diese Weise nicht nur den Klartext aus dem Chiffretext, sondern auch den verwendeten Schlüssel.

4.2 Private-Key-Verfahren und Public-Key-Verfahren

Wir erläutern kurz den Unterschied zwischen symmetrischen und asymmetrischen Kryptosystemen.

Wenn Alice an Bob Nachrichten schicken will, die mit einem Kryptosystem verschlüsselt sind, dann benötigt Alice einen Schlüssel e zum Verschlüsseln und Bob braucht den zugehörigen Schlüssel d zum Entschlüsseln.

Wenn in dem Kryptosystem der Verschlüsselungsschlüssel e immer mit dem entsprechenden Entschlüsselungsschlüssel d übereinstimmt oder d aus e wenigstens leicht zu berechnen ist, spricht man von einem *symmetrischen Verschlüsselungsverfahren* oder einem *Private-Key-Verfahren*. Bei Verwendung eines solchen Verfahrens müssen Alice und Bob zu Beginn der Kommunikation den Schlüssel e über eine sichere Leitung austauschen und dann geheimhalten. Der Entschlüsselungsschlüssel d kann ja aus e berechnet werden. Die Verschiebungschiffre ist z.B. ein symmetrisches Kryptosystem. Die Schlüssel zum Ver- und Entschlüsseln sind nämlich immer gleich.

In *asymmetrischen Kryptosystemen* sind d und e verschieden und d ist aus e auch nicht mit vertretbarem Aufwand zu berechnen. Will Bob verschlüsselte Nachrichten empfangen, so veröffentlicht er den Verschlüsselungsschlüssel e, hält aber den Entschlüsselungsschlüssel d geheim. Jeder kann e benutzen, um geheime Nachrichten an Bob zu schicken. Daher heißt e auch *öffentlicher Schlüssel (public key)* und d heißt *privater Schlüssel (private key)*. Solche Kryptosysteme heißen auch *Public-Key-Verfahren*.

In Public-Key-Verfahren ist es oft nützlich, zwei Schlüsselräume einzuführen, weil Verschlüsselungsschlüssel und Entschlüsselungsschlüssel eine verschiedene Form haben. Beim RSA-Verfahren (siehe Abschnitt 9.3) ist z.B. der öffentliche Schlüssel ein Paar (n, e) natürlicher Zahlen, während der private Schlüssel eine einzige natürliche Zahl d ist. Dadurch ändert sich die Definition von Kryptosystemen aber nicht wesentlich.

In diesem Kapitel werden nur symmetrische Verschlüsselungsverfahren besprochen. Public-Key-Verfahren werden in Kapitel 9 besprochen.

4.3 Sicherheit

4.3.1 Typen von Attacken

Um Angriffe auf Verschlüsselungsverfahren zu erschweren, kann man das verwendete Verfahren geheimhalten. Die Sicherheit, die man daraus gewinnen kann, ist aber sehr zweifelhaft. Ein Angreifer hat nämlich viele Möglichkeiten, zu erfahren, welches Kryptosystem verwendet wird. Er kann verschlüsselte Nachrichten abfangen und daraus Rückschlüsse ziehen. Er kann beobachten, welche technischen Hilfsmittel zur Verschlüsselung benutzt werden. Jemand, der früher mit dem Verfahren gearbeitet hat, gibt die Information preis. Es ist also völlig unklar, ob es gelingen kann, das verwendete Kryptosystem wirklich geheimzuhalten. Daher werden in öffentlichen Anwendungen, wie zum Beispiel im Internet öffentlich bekannte Verschlüsselungsverfahren benutzt. Das Militär und die Geheimdienste verwenden aber oft geheime Verschlüsselungsverfahren.

Bei der Dikussion der Kryptoanalyse gehen wir davon aus, daß bekannt ist, welches Kryptosystem verwendet wird. Nur der verwendete Schlüssel und die verschlüsselten Klartexte werden als geheim angenommen.

Will man die Sicherheit eines Verschlüsselungsverfahrens bestimmen, muß man wissen, welche Ziele ein Angreifer verfolgen kann und welche Möglichkeiten er hat.

Das Ziel eines Angreifers, der einen Schlüsseltext kennt, ist es, möglichst viel über den entsprechenden Klartext zu erfahren. Er kann zum Beispiel versuchen, den geheimen Entschlüsselungsschlüssel zu erfahren. Wenn er ihn kennt, kann er den Schlüsseltext entschlüsseln. Er kann dann sogar alle mit dem entsprechenden Verschlüsselungsschlüssel verschlüsselten Nachrichten entschlüsseln. Der Angreifer kann sich aber auch darauf beschränken, eine einzelne Nachricht zu entschlüsseln ohne den entsprechenden Entschlüsselungsschlüssel zu ermitteln. Oft genügt es dem Angreifer sogar, nicht die gesamte Nachricht zu entschlüsseln, sondern nur eine spezielle Information über die Nachricht in Erfahrung zu bringen. Dies wird an folgendem Beispiel deutlich. Ein Bauunternehmen macht ein Angebot für die Errichtung eines Bürogebäudes. Das Angebot soll vor der Konkurrenz geheimgehalten werden. Die wichtigste Information für die Konkurrenz ist der Preis, der im Angebot genannt wird. Die Details des Angebots sind weniger wichtig. Ein Angreifer kann sich also darauf beschränken, den Preis zu erfahren.

Angreifer haben verschiedene Kenntnisse und Fähigkeiten, die sie verwenden können, um ihr Ziel zu verfolgen. Man unterscheidet folgende Angriffsarten.

Ciphertext-Only-Attacke: Der Angreifer kennt nur den Chiffretext. Dies ist der schwächste Angriff.

Eine einfache Ciphertext-Only-Attacke besteht darin, den Schlüsseltext mit allen Schlüsseln aus dem Schlüsselraum zu entschlüsseln. Unter den wenigen sinnvollen Texten, die sich dabei ergeben, befindet sich der gesuchte Klartext. Wegen der Geschwindigkeit heutiger Computer führt diese Methode bei vielen Kryptosystemen zum Erfolg.

Andere Ciphertext-Only-Angriffe nutzen statistische Eigenschaften der Klartext-Sprache aus. Wird zum Beispiel eine Verschiebungschiffre zum Verschlüsseln benutzt, dann wird bei festem Schlüssel jedes Klartextzeichen durch das gleiche Schlüsseltextzeichen ersetzt. Das häufigste Klartextzeichen entspricht also dem häufigsten Schlüsseltextzeichen, das zweithäufigste Klartextzeichen entspricht dem zweithäufigsten Schlüsseltextzeichen usw. Genauso wiederholt sich die Häufigkeit von Zeichenpaaren, Tripeln usw. Diese statistischen Eigenschaften können ausgenutzt werden, um Chiffretexte zu entschlüsseln und den Schlüssel zu finden. Weitere Beispiele für solche statistischen Angriffe finden sich in [37], [79] und in [6].

Known-Plaintext-Attacke: Der Angreifer kennt andere Klartexte und die zugehörigen Chiffretexte. Ein Beispiel: Briefe enden häufig mit bekannten Formeln, zum Beispiel mit “Beste Grüße”. Kennt der Angreifer den Chiffre-

text dieser Grußformel, dann kann er eine Known-Plaintext-Attacke anwenden. Wir werden im Abschnitt 4.14 sehen, wie Known-Plaintext-Attacken affin lineare Chiffren brechen.

Ein Beispiel aus dem Bereich der symmetrischen Chiffren: Im zweiten Weltkrieg kannten die Alliierten die Nachrichten, die von den deutschen Schiffen gesendet wurden, wenn der Abwurf einer Wasserbombe beobachtet wurde. Die Nachrichten enthielten den Zeitpunkt und den Abwurfort. Um die jeweils aktuellen Schlüssel zu finden, warfen die Alliierten also Wasserbomben ab, ohne deutsche Schiffe zu treffen. Sie fingen die Chiffretexte ab und hatten so Chiffretexte zu selbst gewählten Nachrichten, weil sie ja den Abwurfort und den Abwurfzeitpunkt bestimmen konnten. Diese Kombinationen aus Klar- und Chiffretexten ermöglichten es den Alliierten, den geheimen Schlüssel zu finden.

Chosen-Plaintext-Attacke: Der Angreifer kann kann Chiffretexte zu selbst gewählten Klartexten erzeugen. Dieser Angriff ist bei Public-Key-Verschlüsselungsverfahren immer möglich, weil jeder den öffentlichen Verschlüsselungsschlüssel kennt. Jeder kann also beliebige Klartexte verschlüsseln. Ein Beispiel. Der Angreifer fängt einen Schlüsseltext ab. Er weiß, daß der zugehörige Klartext entweder “ja” oder “nein” ist. Um den richtigen Klartext herauszufinden, verschlüsselt er “ja” und “nein” und vergleicht. Dieses problem von Public-Key-Verschlüsselungsverfahren wird durch randomisierung gelöst (siehe Abschnitt 4.3.2).

Chosen-Ciphertext-Attacke: Der Angreifer kann selbst gewählte Schlüsseltexte entschlüsseln ohne den Entschlüsselungsschlüssel zu kennen. Ein solcher Angriff ist zum Beispiel möglich, wenn Verschlüsselungsverfahren zur Identifikation benutzt werden. Das funktioniert zum Beispiel so. Um herauszufinden, ob ein Webserver mit Bob verbunden ist, schickt der Webserver eine verschlüsselte Zufallszahl an Bob. Der verwendete Entschlüsselungsschlüssel ist nur Bob bekannt. Bob entschlüsselt die Zahl und schickt sie zurück. Jetzt weiß der Webserver, daß er mit Bob verbunden ist. Ein Angreifer, der sich als Webserver ausgibt, kann Bob selbst gewählte Nachrichten entschlüsseln lassen.

Man unterscheidet zwischen *passiven und aktiven Angreifern.*

Ein passiver Angreifer kann nur *Ciphertext-Only-Attacken* anwenden.

Kann ein Angreifer dagegen Chosen-Plaintext oder Chosen-Ciphertext-Attacken anwenden, so handelt es sich um einen aktiven Angreifer. Aktive Angreifer können auch den Chiffretext mit dem Ziel verändern, den Sinn des Klartextes in ihrem Sinn zu beeinflussen.

4.3.2 Randomisierte Verschlüsselung

Werden bei Verwendung eines Private-Key-Verfahrens viele Nachrichten mit demselben Schlüssel verschlüsselt, so muß die Verschlüsselung randomisiert (d.h. zufällig gestaltet) werden. Andernfalls kann zum Beispiel ein Known-Plaintext-Angriff erfolgreich sein. Ist nämlich bekannt, daß ein Chiffretext

einen bestimmten Klartext verschlüsselt, so kennt ein Angreifer den Klartext, sobald er diesen Chiffretext sieht.

Ein Beispiel: Ein Bankkunde handelt mit Aktien. Er sendet der Bank eine von drei Anweisungen: "Kaufen" oder "Halten" oder "Verkaufen". Diese Anweisungen verschlüsselt er. Ein Angreifer braucht nur einmal zu erfahren, daß der Kunde gekauft hat. Dann weiß er, wie der Chiffretext zu "Kaufen" aussieht und kann diese Anweisung jedesmal entschlüsseln. Entsprechendes gilt für die anderen Anweisungen. Ist die Chiffre randomisiert, sieht der Chiffretext zu einem festen Klartext mit an Sicherheit grenzender Wahrscheinlichkeit bei jeder Verschlüsselung anders aus. Wenn der Angreifer also einmal den Chiffretext zu "Kaufen" beobachtet, nützt das nichts.

Noch einfacher ist der Angriff bei Public-Key-Verfahren. Angenommen der Aktienkunde verwendet ein solches Verfahren. Dann verschlüsselt er die Anweisungen "Kaufen", "Halten" und "Verkaufen" mit dem öffentlichen Schlüssel der Bank. Ein Angreifer, der wissen will, welche Anweisung gesendet wurde, verschlüsselt die drei Anweisungen selbst. Dann kennt er die Schlüsseltexte, die zu den drei Klartexten gehören. Er weiß, welche Anweisung gesendet wurde, sobald er den Schlüsseltext des Kunden sieht. Darum muß die Verschlüsselung randomisiert werden.

Wird eine Private-Key-Blockchiffre im CBC-, CFB- oder OFB-Mode benutzt (siehe Abschnitt 4.8), so erfolgt die Randomisierung der Chiffre durch die Auswahl des Initialisierungsvektors. Es handelt sich dabei um einen Startwert, der nicht geheimgehalten werden muss. Die Randomisierung von Public-Key-Verfahren wird im Kapitel 9 erläutert.

4.3.3 Mathematische Modellierung

Soll die Sicherheit eines Verschlüsselungsverfahrens mathematisch nachgewiesen werden, wird zunächst ein mathematisches Sicherheitsmodell benötigt. Mathematische Sicherheitsmodelle liefert die Komplexitätstheorie. Das wird in diesem Buch nicht behandelt. Eine Einführung ist zum Beispiel das Buch von Goldreich [33].

4.4 Alphabete und Wörter

Um Texte aufzuschreiben, braucht man Zeichen aus einem Alphabet. Unter einem *Alphabet* verstehen wir eine endliche, nicht leere Menge Σ. Die *Länge* von Σ ist die Anzahl der Elemente in Σ. Die Elemente von Σ heißen *Zeichen*, *Buchstaben* oder *Symbole* von Σ.

Beispiel 4.4.1. Ein bekanntes Alphabet ist

$$\Sigma = \{\text{A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z}\}.$$

Es hat die Länge 26.

Beispiel 4.4.2. In der Datenverarbeitung wird das Alphabet $\{0, 1\}$ verwendet. Es hat die Länge 2.

Beispiel 4.4.3. Ein häufig benutztes Alphabet ist der der ASCII-Zeichensatz. Dieser Zeichensatz samt seiner Kodierung durch die Zahlen von 0 bis 127 findet sich in Tabelle 4.2.

0	NUL	1	SOH	2	STX	3	ETX
4	EOT	5	ENQ	6	ACK	7	BEL
8	BS	9	HT	10	NL	11	VT
12	NP	13	CR	14	SO	15	SI
16	DLE	17	DC1	18	DC2	19	DC3
20	DC4	21	NAK	22	SYN	23	ETB
24	CAN	25	EM	26	SUB	27	ESC
28	FS	29	GS	30	RS	31	US
32	SP	33	!	34	“	35	#
36	$	37	%	38	&	39	’
40	(	41	)	42	*	43	+
44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3
52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;
60	¡	61	=	62	¿	63	?
64	@	65	A	66	B	67	C
68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K
76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S
84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[
92		93	]	94	ˆ	95	_
96	‘	97	a	98	b	99	c
100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k
108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s
116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{
124	—	125	}	126	˜	127	DEL

Tabelle 4.2. Der ASCII-Zeichensatz

Da Alphabete endliche Mengen sind, kann man ihre Zeichen mit nicht negativen ganzen Zahlen identifizieren. Hat ein Alphabet die Länge m, so identifiziert man seine Zeichen mit den Zahlen in $\mathbb{Z}_m = \{0, 1, \ldots, m-1\}$. Für das Alphabet $\{A, B, \ldots, Z\}$ und den ASCII-Zeichensatz haben wir das in den Tabellen 4.1 und 4.2 dargestellt. Das Alphabet $\{0, 1\}$ sieht schon richtig

aus. Wir werden daher meistens das Alphabet $\mathbb{Z}_m$ verwenden, wobei m eine natürliche Zahl ist.

In der folgenden Definition brauchen wir eine endliche Folge, an die wir kurz erinnern. Ein Beispiel für eine endliche Folge ist

$$(2, 3, 1, 2, 3).$$

Sie hat fünf Folgenglieder. Das erste ist 2, das zweite 3 usw. Manchmal schreibt man die Folge auch als

$$23123.$$

Aus formalen Gründen braucht man auch die *leere Folge* (). Sie hat null Folgenglieder.

Definition 4.4.4. *Sei Σ ein Alphabet.*

1. *Als* Wort *oder* String *über Σ bezeichnet man eine endliche Folge von Zeichen aus Σ einschließlich der leeren Folge, die mit ε bezeichnet und* leeres Wort *genannt wird.*
2. *Die* Länge *eines Wortes $\mathbf{w}$ über Σ ist die Anzahl seiner Zeichen. Sie wird mit $|\mathbf{w}|$ bezeichnet. Das leere Wort hat die Länge* 0.
3. *Die Menge aller Wörter über Σ einschließlich des leeren wird mit Σ^* bezeichnet.*
4. *Sind $\mathbf{v}, \mathbf{w} \in \Sigma^*$, dann ist der String $\mathbf{vw} = \mathbf{v} \circ \mathbf{w}$, den man durch Hintereinanderschreiben von $\mathbf{v}$ und $\mathbf{w}$ erhält, die* Konkatenation *von $\mathbf{v}$ und $\mathbf{w}$. Insbesondere ist $\mathbf{v} \circ \varepsilon = \varepsilon \circ \mathbf{v} = \mathbf{v}$.*
5. *Ist n eine nicht negative ganze Zahl, dann bezeichnet Σ^n die Menge aller Wörter der Länge n über Σ.*

Wie in Übung 4.16.5 gezeigt wird, ist $(\Sigma^*, \circ)$ eine Halbgruppe. Deren neutrales Element ist das leere Wort.

Beispiel 4.4.5. Ein Wort über dem Alphabet aus Beispiel 4.4.1 ist COLA. Es hat die Länge vier. Ein anderes Wort über Σ ist COCA. Die Konkatenation von COCA und COLA ist COCACOLA.

4.5 Permutationen

Um eine sehr allgemeine Klasse von Verschlüsselungsverfahren, die Blockchiffren (siehe Abschnitt 4.6), zu charakterisieren, wird der Begriff der Permutation benötigt.

Definition 4.5.1. *Sei X eine Menge. Eine* Permutation *von X ist eine bijektive Abbildung $f : X \to X$. Die Menge aller Permutationen von X wird mit $S(X)$ bezeichnet.*

Beispiel 4.5.2. Sei $X = \{0, 1, \ldots, 5\}$. Man erhält eine Permutation von X, wenn man jedem Element von X in der oberen Zeile der folgenden Matrix die Ziffer in der unteren Zeile zuordnet.

$$\begin{pmatrix} 0\,1\,2\,3\,4\,5 \\ 1\,2\,4\,3\,5\,0 \end{pmatrix}$$

Auf diese Weise lassen sich Permutationen endlicher Mengen immer darstellen.

Die Menge $S(X)$ aller Permutationen von X zusammen mit der Hintereinanderausführung bildet eine Gruppe, die im allgemeinen nicht kommutativ ist.

Ist n eine natürliche Zahl, dann bezeichnet man mit S_n die Gruppe der Permutationen der Menge $\{1, 2, \ldots, n\}$.

Beispiel 4.5.3. Die Gruppe S_2 hat die Elemente $\begin{pmatrix} 1\,2 \\ 1\,2 \end{pmatrix}, \begin{pmatrix} 1\,2 \\ 2\,1 \end{pmatrix}$.

Theorem 4.5.4. *Die Gruppe S_n hat genau $n! = 1 * 2 * 3 * \cdots * n$ Elemente.*

Beweis. Wir beweisen die Behauptung durch Induktion über n. Offensichtlich hat S_1 nur ein Element. Angenommen, S_{n-1} hat $(n-1)!$ viele Elemente. Betrachte jetzt Permutationen der Menge $\{1, \ldots, n\}$. Wir zählen die Anzahl dieser Permutationen, die 1 auf ein festes Element x abbilden. Bei einer solchen Permutation werden die Zahlen $2, \ldots, n$ bijektiv auf die Zahlen $1, 2, \ldots, x-1, x+1, \ldots, n$ abgebildet. Nach Induktionsannahme gibt es $(n-1)!$ solche Bijektionen. Da es aber n Möglichkeiten gibt, 1 abzubilden, ist $n(n-1)! = n!$ die Gesamtzahl der Permutationen in S_n. □

Sei $X = \{0,1\}^n$ die Menge aller Bitstrings der Länge n. Eine Permutation von X, in der nur die Positionen der Bits vertauscht werden, heißt *Bitpermutation* . Um eine solche Bitpermutation formal zu beschreiben, wählt man $\pi \in S_n$. Dann setzt man

$$f : \{0,1\}^n \to \{0,1\}^n, b_1 \ldots b_n \mapsto b_{\pi(1)} \ldots b_{\pi(n)}.$$

Dies ist tatsächlich eine Bitpermutation und jede Bitpermutation läßt sich in eindeutiger Weise so schreiben. Es gibt also $n!$ Bitpermutationen von Bitstrings der Länge n.

Spezielle Bitpermutationen sind z.B. *zirkuläre Links- oder Rechtsshifts.* Ein zirkulärer Linksshift um i Stellen macht aus dem Bitstring $(b_0, b_1, \ldots, b_{n-1})$ den String $(b_{i \bmod n}, b_{(i+1) \bmod n}, \ldots, b_{(i+n-1) \bmod n})$. Zirkuläre Rechtsshifts sind entsprechend definiert.

4.6 Blockchiffren

Blockchiffren sind Verschlüsselungsverfahren, die Blöcke fester Länge auf Blöcke derselben Länge abbilden. Wie wir in Abschnitt 4.8 sehen werden, kann man sie in unterschiedlicher Weise benutzen, um beliebig lange Texte zu verschlüsseln.

Definition 4.6.1. *Unter einer* Blockchiffre *versteht man ein Verschlüsselungsverfahren, in dem Klartext- und Schlüsseltextraum die Menge Σ^n aller Wörter der Länge n über einem Alphabet Σ sind. Die* Blocklänge n *ist eine natürliche Zahl.*

Blockchiffren der Länge 1 heißen *Substitutionschiffren.*
Wir beweisen folgende Eigenschaft von Blockchiffren:

Theorem 4.6.2. *Die Verschlüsselungsfunktionen einer Blockchiffre sind Permutationen.*

Beweis. Weil zu jeder Verschlüsselungsfunktion eine passende Entschlüsselungsfunktion existiert, sind Verschlüsselungsfunktionen injektiv. Eine injektive Abbildung $\Sigma^n \to \Sigma^n$ ist bijektiv. □

Nach Theorem 4.6.2 kann man die allgemeinste Blockchiffre folgendermaßen beschreiben. Man fixiert die Blocklänge n und ein Alphabet Σ. Als Klartext- und Schlüsseltextraum verwendet man $\mathcal{P} = \mathcal{C} = \Sigma^n$. Der Schlüsselraum ist die Menge $S(\Sigma^n)$ aller Permutationen von Σ^n. Zu einem Schlüssel $\pi \in S(\Sigma^n)$ gehört die Verschlüsselungsfunktion

$$E_\pi : \Sigma^n \to \Sigma^n, \mathbf{v} \mapsto \pi(\mathbf{v}).$$

Die entsprechende Entschlüsselungsfunktion ist

$$D_\pi : \Sigma^n \to \Sigma^n, \mathbf{v} \mapsto \pi^{-1}(\mathbf{v}).$$

Der Schlüsselraum dieses Verfahrens ist sehr groß. Er enthält $(|\Sigma|^n)!$ viele Elemente. Das Verfahren ist aber nicht besonders praktikabel, weil man den Schlüssel, also die Permutation π, benötigt. Stellt man die Permutation dar, indem man zu jedem Klartext $\mathbf{w} \in \Sigma^n$ den Wert $\pi(\mathbf{w})$ notiert, dann erhält man eine Tabelle von $|\Sigma|^n$ Werten, und die ist sehr groß. Es ist daher vernünftig, als Ver- und Entschlüsselungsfunktionen nur einen Teil aller möglichen Permutationen von Σ^n zu verwenden. Diese Permutationen sollten durch geeignete Schlüssel leicht erzeugbar sein.

Ein Beispiel ist die *Permutationschiffre* . Sie verwendet nur solche Permutationen, die durch Vertauschen der Positionen der Zeichen entstehen. Falls $\Sigma = \{0, 1\}$ ist, sind das die Bitpermutationen. Der Schlüsselraum ist die Permutationsgruppe S_n. Für $\pi \in S_n$ setzt man

$$E_\pi : \Sigma^n \to \Sigma^n, \quad (v_1, \ldots, v_n) \mapsto (v_{\pi(1)}, \ldots v_{\pi(n)}).$$

Die zugehörige Entschlüsselungsfunktion ist

$$D_\pi : \Sigma^n \to \Sigma^n, \quad (x_1, \ldots, x_n) \mapsto (x_{\pi^{-1}(1)}, \ldots x_{\pi^{-1}(n)}).$$

Der Schlüsselraum hat $n!$ viele Elemente. Jeder einzelne Schlüssel läßt sich als eine Folge von n Zahlen kodieren.

Eine Methode, Blockchiffren auf ihre Unsicherheit zu untersuchen, besteht darin, ihre algebraischen Eigenschaften zu studieren. Jede Verschlüsselungsfunktion ist ja eine Permutation. Ist die Ordnung dieser Permutation klein, so kann man sie durch iterierte Anwendung entschlüsseln.

4.7 Mehrfachverschlüsselung

Will man die Sicherheit einer Blockchiffre steigern, so kann man sie mehrmals hintereinander mit verschiedenen Schlüsseln verwenden. Gebräuchlich ist die E-D-E-Dreifach-Verschlüsselung (Triple Encryption) . Einen Klartext p verschlüsselt man in

$$c = E_{k_1}(D_{k_2}(E_{k_3}(p))).$$

Dabei sind $k_i, 1 \leq i \leq 3$, drei Schlüssel, E_{k_i} ist die Verschlüsselungsfunktion und D_{k_i} ist die Entschlüsselungsfunktion zum Schlüssel k_i, $1 \leq i \leq 3$. Man erreicht auf diese Weise eine erhebliche Vergrößerung des Schlüsselraums. Will man die Schlüssellänge nur verdoppeln, benutzt man $k_1 = k_3$.

4.8 Verschlüsselungsmodi

Bevor weitere klassische Beispiele für Verschlüsselungsverfahren besprochen werden, behandeln wir erst Verwendungsmöglichkeiten von Blockchiffren zur Verschlüsselung von längeren Texten.

4.8.1 ECB-Mode

In diesem Abschnitt verwenden wir eine Blockchiffre mit Alphabet Σ und Blocklänge n. Der Schlüsselraum sei $\mathcal{K}$. Die Verschlüsselungsfunktionen seien E_k und die Entschlüsselungsfunktionen D_k, $k \in \mathcal{K}$.

Eine naheliegende Art aus dieser Blockchiffre ein Verschlüsselungsverfahren für beliebig lange Texte zu machen, ist ihre Verwendung im *Electronic Codebook Mode* (ECB-Mode). Ein beliebig langer Klartext wird in Blöcke der Länge n aufgeteilt. Gegebenenfalls muß man den Klartext so ergänzen, daß seine Länge durch n teilbar ist. Diese Ergänzung erfolgt zum Beispiel durch Anhängen von zufälligen Zeichen. Bei Verwendung des Schlüssels e

wird dann jeder Block der Länge n mit Hilfe der Funktion E_e verschlüsselt. Der Schlüsseltext ist die Folge der Schlüsseltextblöcke. Die Entschlüsselung erfolgt durch Anwendung der Entschlüsselungsfunktion D_d mit dem zu e korrespondierenden Schlüssel.

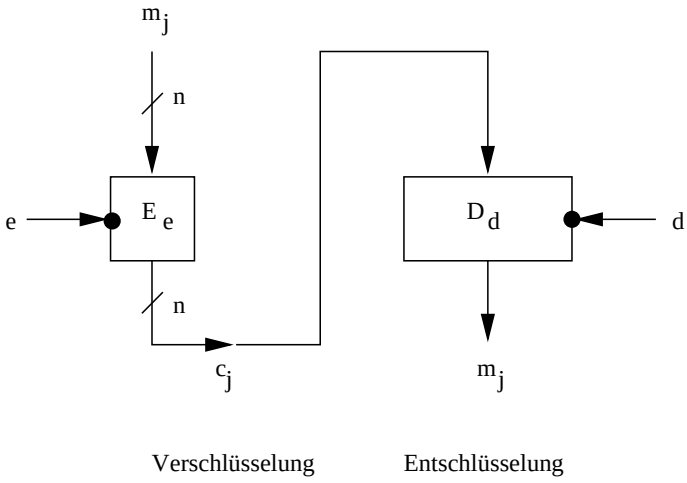


Abb. 4.1. ECB-Mode

Beispiel 4.8.1. Wir betrachten die Blockchiffre, die auf Bitvektoren der Länge vier Bitpermutationen durchführt, also die Permutationschiffre mit Alphabet $\Sigma = \{0, 1\}$ und Blocklänge 4. Es ist $\mathcal{K} = S_4$ und für $\pi \in S_4$ ist

$$E_\pi : \{0,1\}^4 \to \{0,1\}^4, \quad b_1b_2b_3b_4 \mapsto b_{\pi(1)}b_{\pi(2)}b_{\pi(3)}b_{\pi(4)}.$$

Der Klartext m sei

$$m = 101100010100101.$$

Dieser Klartext wird in Blöcke der Länge vier aufgeteilt. Der letzte Block hat dann nur die Länge drei. Er wird auf die Länge vier ergänzt, indem eine Null angehängt wird. Man erhält

$$m = 1011\,0001\,0100\,1010,$$

also die Blöcke

$$m_1 = 1011, m_2 = 0001, m_3 = 0100, m_4 = 1010.$$

Wir verwenden den Schlüssel

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \end{pmatrix}.$$

Die Blöcke werden nun einzeln verschlüsselt. Wir erhalten $c_1 = E_\pi(m_1) = 0111, c_2 = E_\pi(m_2) = 0010, c_3 = E_\pi(m_3) = 1000, c_4 = E_\pi(m_4) = 0101$. Der Chiffretext ist

$$c = 0111001010000101.$$

Der ECB-Mode kann auch mit Verschlüsselungsverfahren angewandt werden, die Blöcke der Länge n in Blöcke der größeren Länge m verschlüsseln.

Bei der Verwendung des ECB-Mode werden gleiche Klartextblöcke in gleiche Chiffretextblöcke verschlüsselt. Regelmäßigkeiten des Klartextes führen zu Regelmäßigkeiten des Schlüsseltextes. Man erhält also Informationen über den Klartext aus dem Schlüsseltext, auch wenn man ihn nicht entschlüsseln kann. Solche Informationen können die Kryptoanalyse erleichtern. Ein anderer Nachteil des ECB-Mode besteht darin, daß ein Angreifer die Nachricht ändern kann, indem er Chiffretext einfügt, der mit dem gleichen Schlüssel verschlüsselt worden ist. Genauso kann ein Angreifer unbemerkt die Reihenfolge der Schlüsseltextblöcke verändern. Aus diesen Gründen ist der ECB-Mode zur Verschlüsselung langer Nachrichten ungeeignet.

Man kann die Sicherheit des ECB-Mode steigern, wenn man die Blöcke nur teilweise aus dem Klartext und teilweise durch zufällige Zeichen bildet.

4.8.2 CBC-Mode

Um die Nachteile des ECB-Mode zu beseitigen, wurde der *Cipherblock Chaining Mode* (CBC-Mode) erfunden. In diesem Mode hängt die Verschlüsselung eines Klartextblocks nicht nur von diesem Block und dem Schlüssel, sondern auch von den vorhergegangenen Blöcken ab. Die Verschlüsselung ist also kontextabhängig. Gleiche Muster in unterschiedlichem Kontext werden verschieden verschlüsselt. Nachträgliche Veränderung des Schlüsseltextes kann man daran erkennen, daß die Entschlüsselung nicht mehr funktioniert. Der CBC-Mode wird nun im Detail beschrieben. Es wird eine Blockchiffre mit Alphabet $\Sigma = \{0, 1\}$, Blocklänge n, Schlüsselraum $\mathcal{K}$, Verschlüsselungsfunktionen E_k und Entschlüsselungsfunktionen D_k, $k \in \mathcal{K}$, verwendet.

Wir brauchen noch eine kleine Definition:

Definition 4.8.2. *Die Verknüpfung*

$$\oplus : \{0, 1\}^2 \to \{0, 1\}, (b, c) \mapsto b \oplus c$$

ist durch folgende Tabelle definiert:

b	c	$b \oplus c$
0	0	0
1	0	1
0	1	1
1	1	0

Diese Verknüpfung heißt exklusives Oder *von zwei Bits.*

Für $k \in \mathbb{N}$ und $b = (b_1, b_2, \ldots, b_k)$, $c = (c_1, c_2, \ldots, c_k) \in \{0,1\}^k$ setzt man $b \oplus c = (b_1 \oplus c_1, b_2 \oplus c_2, \ldots, b_k \oplus c_k)$.

Werden die Restklassen in $\mathbb{Z}/2\mathbb{Z}$ durch ihre kleinsten nicht negativen Vertreter 0 und 1 dargestellt, so entspricht das exklusive Oder zweier Elemente von $\mathbb{Z}/2\mathbb{Z}$ der Addition in $\mathbb{Z}/2\mathbb{Z}$.

Beispiel 4.8.3. Ist $b = 0100$ und $c = 1101$ dann ist $b \oplus c = 1001$.

Der CBC-Mode benötigt einen festen *Initialisierungsvektor*

$$IV \in \Sigma^n.$$

Wie im ECB-Mode wird der Klartext in Blöcke der Länge n aufgeteilt. Will man eine Folge $m_1, \ldots, m_t$ von Klartextblöcken der Länge n mit dem Schlüssel e verschlüsseln, so setzt man

$$c_0 = IV, \quad c_j = E_e(c_{j-1} \oplus m_j), \quad 1 \le j \le t.$$

Man erhält die Schlüsseltextblöcke

$$c_1, \ldots, c_t.$$

Um sie zu entschlüsseln, benötigt man den Schlüssel d, der $D_d(E_e(w)) = w$ für alle Blöcke w erfüllt. Dann setzt man

$$c_0 = IV, \quad m_j = c_{j-1} \oplus D_d(c_j), \quad 1 \le j \le t. \tag{4.1}$$

Tatsächlich ist $c_0 \oplus D_d(c_1) = c_0 \oplus c_0 \oplus m_1 = m_1$. Entsprechend verifiziert man das ganze Verfahren.

Beispiel 4.8.4. Wir verwenden dieselbe Blockchiffre, denselben Klartext und denselben Schlüssel wie in Beispiel 4.8.1. Die Klartextblöcke sind

$$m_1 = 1011, m_2 = 0001, m_3 = 0100, m_4 = 1010.$$

Der Schlüssel ist

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \end{pmatrix}.$$

Als Initialisierungsvektor verwenden wir

$$IV = 1010.$$

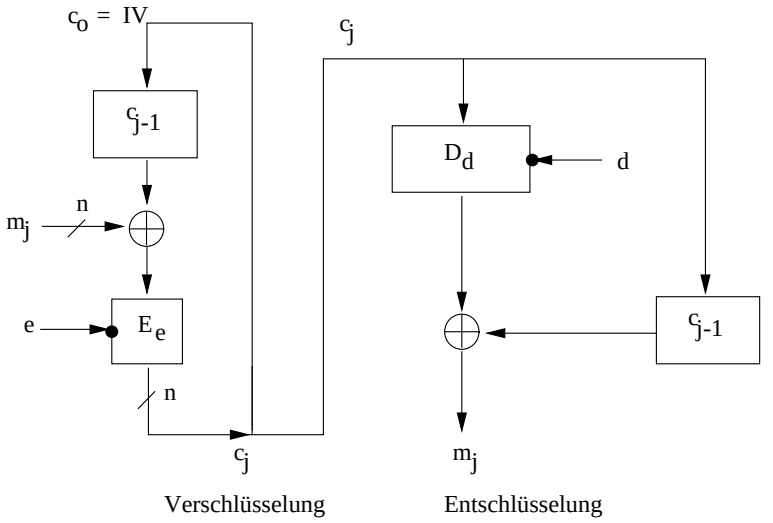


Abb. 4.2. CBC-Mode

Damit ist also $c_0 = 1010$, $c_1 = E_\pi(c_0 \oplus m_1) = E_\pi(0001) = 0010$, $c_2 = E_\pi(c_1 \oplus m_2) = E_\pi(0011) = 0110$, $c_3 = E_\pi(c_2 \oplus m_3) = E_\pi(0010) = 0100$, $c_4 = E_\pi(c_3 \oplus m_4) = E_\pi(1110) = 1101$. Also ist der Schlüsseltext

$$c = 0010011001001101.$$

Wir entschlüsseln diesen Schlüsseltext wieder und erhalten $m_1 = c_0 \oplus E_\pi^{-1}(c_1) = 1010 \oplus 0001 = 1011$, $m_2 = c_1 \oplus E_\pi^{-1}(c_2) = 0010 \oplus 0011 = 0001$, $m_3 = c_2 \oplus E_\pi^{-1}(c_3) = 0110 \oplus 0010 = 0100$, $m_4 = c_3 \oplus E_\pi^{-1}(c_4) = 0100 \oplus 1110 = 1010$.

Gleiche Texte werden im CBC-Mode verschieden verschlüsselt, wenn man den Initialisierungsvektor ändert. Außerdem hängt die Verschlüsselung eines Blocks von den vorhergegangenen Blöcken ab. Gleiche Klartextblöcke in unterschiedlichem Kontext werden im allgemeinen verschieden verschlüsselt. Verändert man die Reihenfolge der Schlüsseltextblöcke oder ändert man einen Schlüsseltextblock, so läßt sich der Schlüsseltext nicht mehr entschlüsseln. Dies ist ein Vorteil gegenüber dem ECB-Mode.

Wir untersuchen, welche Auswirkungen Übertragungsfehler haben können. Bei solchen Übertragungsfehlern enthalten einige Blöcke des Schlüsseltextes, den der Adressat empfängt, Fehler. Man sieht aber an (4.1), daß ein Übertragungsfehler im Schlüsseltextwort c_j nur bewirken kann, daß m_j und m_{j+1} falsch berechnet werden. Die Berechnung von $m_{j+2}, m_{j+3}, \ldots$ ist dann wieder korrekt, weil sie nicht von c_j abhängt. Das bedeutet auch, daß Sender und Empfänger nicht einmal denselben Initialisierungsvektor brauchen. Haben sie verschiedene Initialisierungsvektoren gewählt, so kann der Empfänger

zwar vielleicht nicht den ersten Block, aber dann alle weiteren Blöcke korrekt entschlüsseln.

4.8.3 CFB-Mode

Der CBC-Mode ist zum Verschlüsseln langer Nachrichten gut geeignet. Es kann jedoch Effizienzprobleme geben, weil der Empfänger immer abwarten muß, bis der Sender einen ganzen Schlüsseltextblock erzeugt und diesen versandt hat, bevor er mit der Entschlüsselung des Blocks beginnen kann. Solche Blöcke können ziemlich lang sein. Beim *Cipher Feedback Mode* (CFB-Mode) ist das anders. Wir verwenden dieselbe Blockchiffre wie beim CBC-Mode, um diesen Modus zu erklären.

Im CFB-Modus werden Blöcke, die eine kürzere Länge als n haben können, nicht direkt durch die Blockverschlüsselungsfunktion E_k, sondern durch Addition mod 2 entsprechender Schlüsselblöcke verschlüsselt. Diese Schlüsselblöcke können mit Hilfe der Blockchiffre auf Sender- und Empfängerseite fast simultan berechnet werden. Im einzelnen funktioniert das folgendermaßen:

Man benötigt einen Initialisierungsvektor $IV \in \{0,1\}^n$ und eine natürliche Zahl r, $1 \leq r \leq n$. Der Klartext wird in Blöcke der Länge r aufgeteilt. Wenn man die Blockfolge $m_1, \ldots, m_u$ verschlüsseln möchte, setzt man

$$I_1 = IV$$

und dann für $1 \leq j \leq u$

1. $O_j = E_k(I_j)$,
2. t_j auf den String, der aus den ersten r Bits von O_j gebildet wird,
3. $c_j = m_j \oplus t_j$,
4. $I_{j+1} = 2^r I_j + c_j \bmod 2^n$. I_{j+1} entsteht also, indem in I_j die ersten r Bits gelöscht werden und c_j hinten angehängt wird.

Der Schlüsseltext ist die Folge $c_1, c_2, \ldots, c_u$.

Die Entschlüsselung funktioniert ähnlich wie die Verschlüsselung. Man setzt

$$I_1 = IV$$

und dann für $1 \leq j \leq u$

1. $O_j = E_k(I_j)$,
2. t_j auf den String, der aus den ersten r Bits von O_j gebildet wird,
3. $m_j = c_j \oplus t_j$,
4. $I_{j+1} = 2^r I_j + c_j \bmod 2^n$.

Man erkennt, daß sowohl Sender als auch Empfänger den String t_{j+1} bestimmen können, sobald sie c_j kennen. Der Schlüssel t_1 kann also von Sender und Empfänger gleichzeitig berechnet werden. Dann erzeugt der Sender $c_1 = m_1 \oplus t_1$ und versendet c_1. Die Berechnung von c_1 geht sehr schnell. Danach können Sender und Empfänger simultan t_2 berechnen usw.

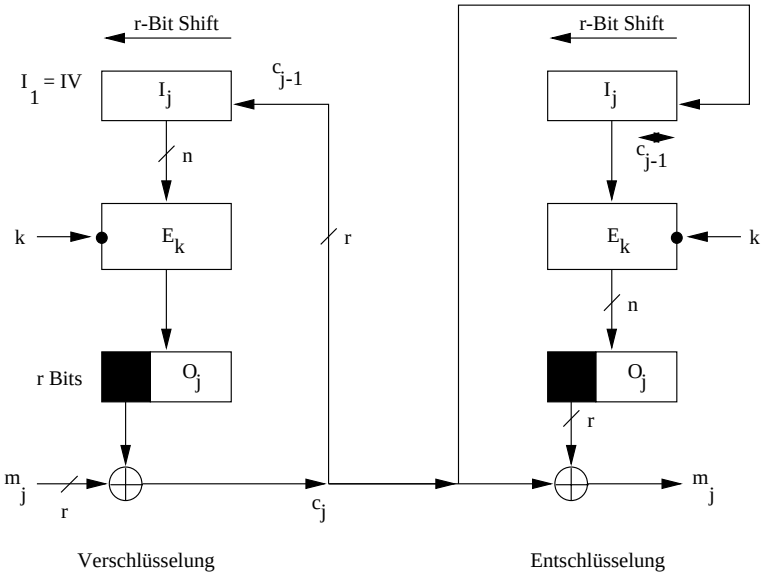


Abb. 4.3. CFB-Mode

Beispiel 4.8.5. Wir verwenden Blockchiffre, Klartext und Schlüssel aus Beispiel 4.8.1. Außerdem verwenden wir als verkürzte Blocklänge $r = 3$. Die Klartextblöcke sind dann

$$m_1 = 101, m_2 = 100, m_3 = 010, m_4 = 100, m_5 = 101.$$

Der Schlüssel ist

$$\pi = \begin{pmatrix} 1\,2\,3\,4 \\ 2\,3\,4\,1 \end{pmatrix}.$$

Als Initialisierungsvektor verwenden wir

$$IV = 1010.$$

Die Verschlüsselung erfolgt dann gemäß folgender Tabelle.

j	I_j	O_j	t_j	m_j	c_j
1	1010	0101	010	101	111
2	0111	1110	111	100	011
3	1011	0111	011	010	001
4	1001	0011	001	100	101
5	1101	1011	101	101	000

Im CFB-Mode beeinflussen Übertragungsfehler das Ergebnis der Entschlüsselung solange, bis der fehlerhafte Ciphertextblock aus dem Vektor I_j

herausgeschoben wurde. Wie lange das dauert, hängt auch von der Größe von r ab.

Der CFB-Mode kann bei Public-Key-Verfahren nicht angewendet werden, weil beide Kommunikationspartner denselben Schlüssel k kennen müßten.

4.8.4 OFB-Mode

Der *Output Feedback Mode* (OFB-Mode) ist dem CFB-Mode ähnlich. Die Voraussetzungen und die Initialisierung sind genauso wie im CFB-Mode. Dann setzt man für $1 \leq j \leq u$

1. $O_j = E_k(I_j)$,
2. t_j auf den String, der aus den ersten r Bits von O_j gebildet wird,
3. $c_j = m_j \oplus t_j$,
4. $I_{j+1} = O_j$.

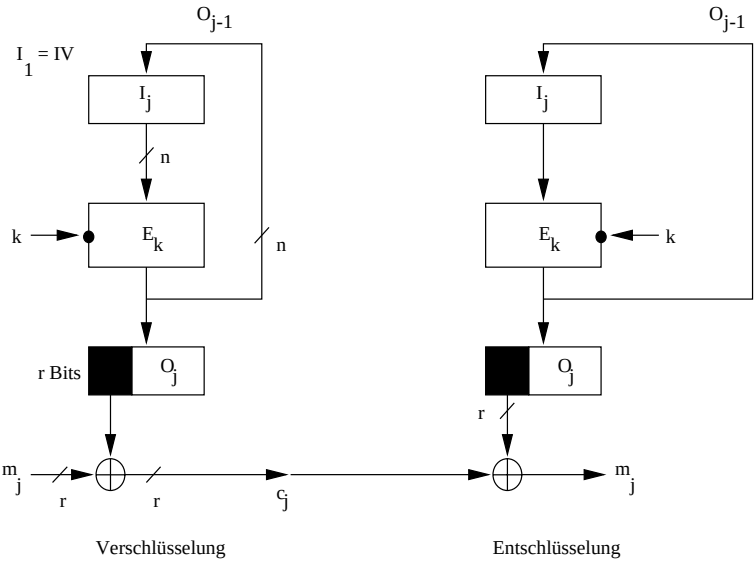


Abb. 4.4. OFB-Mode

Wieder funktioniert die Entschlüsselung wie die Verschlüsselung, wobei der dritte Schritt durch die Vorschrift $m_j = c_j \oplus t_j$ ersetzt wird.

Werden Bits bei der Übertragung eines Schlüsseltextwortes verändert, so ensteht bei der Entschlüsselung nur ein Fehler an genau derselben Position, aber sonst nirgends.

Die Schlüsselstrings t_j hängen nur vom Initialisierungsvektor I_1 und vom Schlüssel k ab. Sie können also von Sender und Empfänger parallel berechnet

werden. Die Verschlüsselung der Klartextblöcke hängt nicht von den vorherigen Klartextblöcken, sondern nur von der Position ab. Daher können im OFB-Mode verschlüsselte Texte leichter manipuliert werden als Texte, die im CBC-Mode verschlüsselt wurden.

Beispiel 4.8.6. Wir verwenden Blockchiffre, Klartext und Schlüssel aus Beispiel 4.8.1. Außerdem verwenden wir als verkürzte Blocklänge $r = 3$. Die Klartextblöcke sind dann

$$m_1 = 101, m_2 = 100, m_3 = 010, m_4 = 100, m_5 = 101.$$

Der Schlüssel ist

$$\pi = \begin{pmatrix} 1\,2\,3\,4 \\ 2\,3\,4\,1 \end{pmatrix}.$$

Als Initialisierungsvektor verwenden wir

$$IV = 1010.$$

Die Verschlüsselung erfolgt dann gemäß folgender Tabelle.

j	I_j	O_j	t_j	m_j	c_j
1	1010	0101	010	101	111
2	0101	1010	101	100	001
3	1010	0101	010	010	000
4	0101	1010	101	100	001
5	1010	0101	010	101	111

Wenn derselbe Schlüssel k mehrmals verwendet werden soll, muß der Initialisierungsvektor IV verändert werden. Sonst erhält man nämlich dieselbe Folge von Schlüsselstrings t_j. Hat man dann zwei Schlüsseltextblöcke $c_j = m_j \oplus t_j$ und $c'_j = m'_j \oplus t_j$, dann erhält man $c_j \oplus c'_j = m_j \oplus m'_j$. Hieraus kann man m'_j ermitteln, wenn m_j bekannt ist.

4.9 Stromchiffren

Wir haben bereits erläutert, wie man mit Hilfe von Blockchiffren Verschlüsselungsverfahren konstruiert, die Klartextblöcke kontextabhängig verschlüsseln.

Dieses Prinzip wird in Stromchiffren verallgemeinert.

Eine bekannte Stromchiffre funktioniert folgendermaßen: Man verwendet das Alphabet $\Sigma = \{0, 1\}$. Der Klartext- und Schlüsseltextraum ist Σ^*. Die Schlüsselmenge ist Σ^n für eine natürliche Zahl n. Wörter in Σ^* werden Zeichen für Zeichen verschlüsselt. Das funktioniert so: Sei $k = (k_1, \ldots, k_n)$ ein Schlüssel und $w = \sigma_1 \ldots \sigma_m$ ein Wort der Länge m in Σ^*. Man erzeugt einen Schlüsselstrom $z_1, z_2, \ldots, z_m$. Dazu setzt man

$$z_i = k_i, \quad 1 \leq i \leq n \tag{4.2}$$

und wenn $m > n$ ist

$$z_i = \sum_{j=1}^{n} c_j z_{i-j} \bmod 2, \quad n < i \leq m, \tag{4.3}$$

wobei $c_1, \ldots, c_n$ für das Verfahren fest gewählte Bits sind. Diese Gleichung nennt man eine *lineare Rekursion* vom Grad n. Die Verschlüsselungsfunktion E_k und die Entschlüsselungsfunktion D_k sind dann gegeben durch die Vorschriften

$$E_k(w) = \sigma_1 \oplus z_1, \ldots, \sigma_m \oplus z_m, D_k(w) = \sigma_1 \oplus z_1, \ldots, \sigma_m \oplus z_m.$$

Beispiel 4.9.1. Sei $n = 4$. Der Schlüsselstrom wird generiert gemäß der Rekursion

$$z_{i+4} = z_i + z_{i+1} \bmod 2.$$

Dann ist also $c_1 = c_2 = 0$, $c_3 = c_4 = 1$. Der Schlüssel sei $k = (1, 0, 0, 0)$. Dann erhält man den Schlüsselstrom

$$1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, \cdots$$

Der Schlüsselstrom ist periodisch und hat die Periodenlänge 15.

Die beschriebene Stromchiffre läßt sich durch ein sogenanntes lineares Schieberegister effizient als Hardwarebaustein realisieren. In Abbildung 4.5 ist ein solches Schieberegister dargestellt. In den Registern befinden sich die letzten vier Werte des Schlüsselstroms. In jedem Schritt wird der Schlüssel aus dem ersten Register zur Verschlüsselung verwendet. Dann wird der zweite, dritte und vierte um eins nach links geshiftet und der neue vierte Schlüssel entsteht durch Addition derjenigen Schlüssel mod 2, für die der Koeffizient c_i gleich 1 ist. Wir verfolgen hier das Thema Stromchiffren nicht weiter, sondern verweisen auf [68].

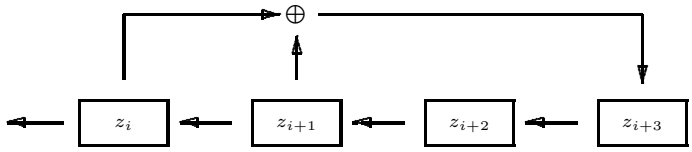


Abb. 4.5. Lineares Schieberegister

4.10 Die affine Chiffre

Sei m eine natürliche Zahl. Die *affine Chiffre* mit Klartextalphabet $\mathbb{Z}_m$ ist eine Blockchiffre der Blocklänge $n = 1$. Der Schlüsselraum besteht aus allen Paaren $(a, b) \in \mathbb{Z}_m^2$ mit zu m primen a. Die Verschlüsselungsfunktion E_e zum Schlüssel $e = (a, b)$ ist

$$E_e : \Sigma \to \Sigma, \quad x \mapsto ax + b \bmod m.$$

Die Entschlüsselungsfunktion zum Schlüssel $d = (a', b)$ ist

$$D_d : \Sigma \to \Sigma, \quad x \mapsto a'(x - b) \bmod m.$$

Um den zu (a, b) passenden Entschlüsselungsschlüssel zu berechnen, löst man $aa' \equiv 1 \bmod m$ mit dem erweiterten euklidischen Algorithmus. Der zu (a, b) passende Schlüssel ist (a', b).

Beispiel 4.10.1. Wählt man $(a, b) = (7, 3)$, verwendet das Alphabet $\mathbb{Z}_{26}$ und verschlüsselt man das Wort BALD mit der affinen Chiffre (m = 26) im ECB-Mode, so ergibt sich:

B	A	L	D
1	0	11	3
10	3	2	24
K	D	C	Y

Zur Berechnung der entsprechenden Entschlüsselungsfunktion bestimmt man a' mit $7a' \equiv 1 \bmod 26$. Der erweiterte euklidische Algorithmus liefert $a' = 15$. Die Entschlüsselungsfunktion bildet also einen Buchstaben σ auf $15(\sigma - 3) \bmod 26$ ab. Tatsächlich erhält man

K	D	C	Y
10	3	2	24
1	0	11	3
B	A	L	D

Der Schlüsselraum der affinen Chiffre mit $m = 26$ hat $\varphi(26) * 26 = 312$ Elemente. Also kann man die affine Chiffre bei einer Ciphertext-Only-Attacke im ECB-Mode entschlüsseln, indem man alle diese Schlüssel durchsucht.

Bei einer Known-Plaintext Attacke, in der man zwei Zeichen und ihre Verschlüsselung kennt, kann man den Schlüssel mit linearer Algebra ermitteln. Dies wird im folgenden Beispiel vorgeführt.

Beispiel 4.10.2. Das Alphabet $\{A, B, \ldots, Z\}$ wird mit $\mathbb{Z}_{26}$ identifiziert. Wenn man weiß, daß bei Anwendung der affinen Chiffre mit Schlüssel (a, b) der Buchstabe E in R und S in H verschlüsselt wird, dann gelten die Kongruenzen

$$4a + b \equiv 17 \bmod 26 \quad 18a + b \equiv 7 \bmod 26.$$

Aus der ersten Kongruenz ergibt sich $b \equiv 17 - 4a \bmod 26$. Setzt man dies in die zweite Kongruenz ein, so erhält man $18a + 17 - 4a \equiv 7 \bmod 26$ und damit $14a \equiv 16 \bmod 26$. Daraus folgt $7a \equiv 8 \bmod 13$. Dies multipliziert man mit dem Inversen 2 von 7 modulo 13 und erhält $a \equiv 3 \bmod 13$. Hieraus schließt man $a = 3$ und $b = 5$.

4.11 Matrizen und lineare Abbildungen

Wir wollen affine Chiffren verallgemeinern. Dazu führen wir einige grundlegende Ergebnisse der linearen Algebra über Ringen auf, ohne sie zu beweisen. Für Details verweisen wir auf [59]. Es sei R ein kommutativer Ring mit Einselement 1. Z.B. kann $R = \mathbb{Z}/m\mathbb{Z}$ sein mit einer natürlichen Zahl m.

4.11.1 Matrizen über Ringen

Eine $k \times n$-*Matrix* über R ist ein rechteckiges Schema

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \cdots & \vdots \\ a_{k,1} & a_{k,2} & \cdots & a_{k,n} \end{pmatrix}$$

mit k Zeilen und n Spalten. Wir schreiben auch

$$A = (a_{i,j}).$$

Ist $n = k$, so heißt die Matrix *quadratisch*. Die i-te Zeile von A ist der Vektor $(a_{i,1}, \ldots, a_{i,n})$, $1 \leq i \leq k$. Die j-te Spalte von A ist der Vektor $(a_{1,j}, \ldots, a_{k,j})$, $1 \leq j \leq n$. Der Eintrag in der i-ten Zeile und j-ten Spalte von A ist $a_{i,j}$. Die Menge aller $k \times n$-Matrizen über R wird mit $R^{(k,n)}$ bezeichnet.

Beispiel 4.11.1. Sei $R = \mathbb{Z}$. Eine Matrix aus $\mathbb{Z}^{(2,3)}$ ist z.B.

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}.$$

Sie hat zwei Zeilen, nämlich $(1, 2, 3)$ und $(4, 5, 6)$ und drei Spalten, nämlich $(1, 4)$, $(2, 5)$ und $(3, 6)$.

4.11.2 Produkt von Matrizen mit Vektoren

Ist $A = (a_{i,j}) \in R^{(k,n)}$ und $\mathbf{v} = (v_1, \ldots, v_n) \in R^n$, dann ist das Produkt $A\mathbf{v}$ definiert als der Vektor $\mathbf{w} = (w_1, w_2, \ldots, w_k)$ mit

$$w_i = \sum_{j=1}^{n} a_{i,j} v_j, \quad 1 \leq i \leq k.$$

Beispiel 4.11.2. Sei $A = \begin{pmatrix} 1 & 2 \\ 2 & 3 \end{pmatrix}$, $\mathbf{v} = (1, 2)$. Dann ist $A\mathbf{v} = (5, 8)$.

4.11.3 Summe und Produkt von Matrizen

Sei $n \in \mathbb{N}$ und seien $A, B \in R^{(n,n)}$, $A = (a_{i,j})$, $B = (b_{i,j})$. Die *Summe* von A und B ist

$$A + B = (a_{i,j} + b_{i,j}).$$

Das *Produkt* von A und B ist $A \cdot B = AB = (c_{i,j})$ mit

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}.$$

Beispiel 4.11.3. Sei $A = \begin{pmatrix} 1 & 2 \\ 2 & 3 \end{pmatrix}$, $B = \begin{pmatrix} 4 & 5 \\ 6 & 7 \end{pmatrix}$. Dann ist $A + B = \begin{pmatrix} 5 & 7 \\ 8 & 10 \end{pmatrix}$, $AB = \begin{pmatrix} 16 & 19 \\ 26 & 31 \end{pmatrix}$, $BA = \begin{pmatrix} 14 & 23 \\ 20 & 33 \end{pmatrix}$. Man sieht daran, daß die Multiplikation von Matrizen im allgemeinen nicht kommutativ ist.

4.11.4 Der Matrizenring

Die $n \times n$-Einheitsmatrix (über R) ist $E_n = (e_{i,j})$ mit

$$e_{i,j} = \begin{cases} 1 \text{ für } i = j, \\ 0 \text{ für } i \neq j. \end{cases}$$

Die $n \times n$-Nullmatrix (über R) ist die $n \times n$-Matrix, deren sämtliche Einträge Null sind. Wir schreiben dafür (0).

Beispiel 4.11.4. Die 2×2-Einheitsmatrix über $\mathbb{Z}$ ist $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$. Die 2×2-Nullmatrix über $\mathbb{Z}$ ist $\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$.

Zusammen mit Addition und Multiplikation ist $R^{(n,n)}$ ein Ring mit Einselement E_n, der aber im allgemeinen nicht kommutativ ist. Das neutrale Element bezüglich der Addition ist die Nullmatrix.

4.11.5 Determinante

Die *Determinante* $\det A$ einer Matrix $A \in R^{(n,n)}$ kann rekursiv definiert werden. Ist $n = 1$, $A = (a)$, dann ist $\det A = a$. Sei $n > 1$. Für $i, j \in \{1, 2, \ldots n\}$ bezeichne mit $A_{i,j}$ die Matrix, die man aus A erhält, wenn man in A die i-te Zeile und die j-te Spalte streicht. Fixiere $i \in \{1, 2, \ldots, n\}$. Dann ist die Determinante von A

$$\det A = \sum_{j=1}^{n} (-1)^{i+j} a_{i,j} \det A_{i,j}$$

Dieser Wert ist unabhängig von der Auswahl von i und es gilt für alle $j \in \{1, 2, \ldots, n\}$

$$\det A = \sum_{i=1}^{n} (-1)^{i+j} a_{i,j} \det A_{i,j}.$$

Beispiel 4.11.5. Sei $A = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix}$. Dann ist $A_{1,1} = (a_{2,2})$, $A_{1,2} = (a_{2,1})$, $A_{2,1} = (a_{1,2})$, $A_{2,2} = (a_{1,1})$. Daher ist $\det A = a_{1,1}a_{2,2} - a_{1,2}a_{2,1}$.

4.11.6 Inverse von Matrizen

Eine Matrix A aus $R^{(n,n)}$ besitzt genau dann ein multiplikatives Inverses, wenn $\det A$ eine Einheit in R ist. Wir geben eine Formel für das Inverse an. Falls $n = 1$ ist, so ist $(a_{1,1}^{-1})$ das Inverse von A. Sei $n > 1$ und $A_{i,j}$ wie oben definiert. Die *Adjunkte* von A ist eine $n \times n$-Matrix. Sie ist definiert als

$$\operatorname{adj} A = ((-1)^{i+j} \det A_{j,i}).$$

Die Inverse von A ist

$$A^{-1} = (\det A)^{-1} \operatorname{adj} A.$$

Beispiel 4.11.6. Sei $A = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix}$. Dann ist $\operatorname{adj} A = \begin{pmatrix} a_{2,2} & -a_{1,2} \\ -a_{2,1} & a_{1,1} \end{pmatrix}$.

Sind $A = (a_{i,j}), B = (b_{i,j}) \in \mathbb{Z}^{(n,n)}$ und ist $m \in \mathbb{N}$, so schreiben wir

$$A \equiv B \bmod m$$

wenn $a_{i,j} \equiv b_{i,j} \bmod m$ ist für $1 \leq i, j \leq n$.

Als Anwendung der Ergebnisse dieses Abschnitts beschreiben wir, wann die Kongruenz

$$AA' \equiv E_n \bmod m \tag{4.4}$$

eine Lösung $A' \in \mathbb{Z}^{(n,n)}$ hat und wie man sie findet. Wir geben zuerst ein Beispiel.

Beispiel 4.11.7. Sei $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$. Wir lösen die Kongruenz

$$AA' \equiv E_2 \bmod 11 \tag{4.5}$$

mit $A' \in \mathbb{Z}^{(2,2)}$. Bezeichnet man mit $\overline{A}$ die Matrix, die man erhält, indem man die Einträge von A durch ihre Restklassen mod m ersetzt, dann ist also das Inverse dieser Matrix gesucht. Es existiert, wenn die Determinante von $\overline{A}$ eine Einheit in $\mathbb{Z}/11\mathbb{Z}$ ist. Das ist genau dann der Fall, wenn $\det A$ zu 11 teilerfremd ist. Nun ist $\det A = -2$, also teilerfremd zu 11. Außerdem ist $(-2)(-6) \equiv 1 \bmod 11$. Setzt man also

$$A' = (-6) * \operatorname{adj} A \bmod 11 = 5 * \begin{pmatrix} 4 & -2 \\ -3 & 1 \end{pmatrix} \bmod 11 = \begin{pmatrix} 9 & 1 \\ 7 & 5 \end{pmatrix},$$

so hat man eine Lösung der Kongruenz (4.5) gefunden.

Wir verallgemeinern das Ergebnis des vorigen Beispiels. Sei $A \in \mathbb{Z}^{n,n}$ und $m > 1$. Dann hat die Kongruenz (4.4) genau dann eine Lösung, wenn $\det A$ teilerfremd zu m ist. Ist dies der Fall und ist a eine ganze Zahl mit $a \det A \equiv 1 \bmod m$, dann ist

$$A' = a \operatorname{adj} A \bmod m$$

eine Lösung der Kongruenz (4.4). Diese Lösung ist eindeutig mod m. Man erkennt, daß die Matrix A' in Polynomzeit berechnet werden kann.

4.11.7 Affin lineare Funktionen

Wir definieren affin lineare Abbildungen. Man kann sie verwenden, um einfache Blockchiffren zu konstruieren.

Definition 4.11.8. *Eine Funktion $f : R^n \to R^l$ heißt* affin linear*, wenn es eine Matrix $A \in R^{(l,n)}$ und einen Vektor $\mathbf{b} \in R^l$ gibt, so daß für alle $\mathbf{v} \in R^n$*

$$f(\mathbf{v}) = A\mathbf{v} + \mathbf{b}$$

gilt. Ist $\mathbf{b} = 0$, so heißt die Abbildung linear*.*

Affin lineare Abbildungen $\mathbb{Z}_m^n \to \mathbb{Z}_m^l$ sind analog definiert.

Definition 4.11.9. *Eine Funktion $f : \mathbb{Z}_m^n \to \mathbb{Z}_m^l$ heißt* affin linear*, wenn es eine Matrix $A \in \mathbb{Z}_m^{(l,n)}$ und einen Vektor $\mathbf{b} \in \mathbb{Z}_m^l$ gibt, so daß für alle $\mathbf{v} \in \mathbb{Z}_m^n$*

$$f(\mathbf{v}) = (A\mathbf{v} + \mathbf{b}) \bmod m$$

gilt. Ist $\mathbf{b} \equiv 0 \bmod m$, so heißt die Abbildung linear*.*

Theorem 4.11.10. *Die affin lineare Abbildung aus Definition 4.11.8 ist genau dann bijektiv, wenn $l = n$ und* $\det A$ *eine Einheit aus R ist.*

Entsprechend ist die Abbildung aus Definition 4.11.8 genau dann bijektiv, wenn $l = n$ und $\det A$ teilerfremd zu m ist.

Beispiel 4.11.11. Betrachte die Abbildung $f : \{0,1\}^2 \to \{0,1\}^2$, die definiert ist durch

$$f(0,0) = (0,0), f(1,0) = (1,1), f(0,1) = (1,0), f(1,1) = (0,1).$$

Diese Abbildung ist linear, weil $f(\mathbf{v}) = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \mathbf{v}$ für alle $\mathbf{v} \in \{0,1\}^2$ gilt.

Wir charakterisieren lineare und affin lineare Abbildungen.

Theorem 4.11.12. *Eine Abbildung $f : R^n \to R^l$ ist genau dann linear, wenn für alle $\mathbf{v}, \mathbf{w} \in R^n$ und alle $a, b \in R$*

$$f(a\mathbf{v} + b\mathbf{w}) = af(\mathbf{v}) + bf(\mathbf{w})$$

gilt. Sie ist genau dann affin linear, wenn die Abbildung $R^n \to R^l, \mathbf{v} \mapsto f(\mathbf{v}) - f(\mathbf{0})$ linear ist.

4.12 Affin lineare Blockchiffren

Wir definieren *affin lineare Blockchiffren*. Sie sind Verallgemeinerungen der affinen Chiffre und wurden früher oft verwendet. Wir beschreiben diese Chiffren hier einerseits aus historischen Gründen. Andererseits beschreiben wir, wie leicht solche Chiffren mit Known-Plaintext-Attacken angegriffen werden können. Das zeigt, daß man beim Design von Blockchiffren vermeiden muß, daß sie affin linear sind.

Um affin lineare Blockchiffren zu definieren, brauchen wir eine natürliche Zahl n, die Blocklänge und eine natürliche Zahl m, $m > 1$.

Definition 4.12.1. *Eine Blockchiffre mit Blocklänge n und Klartext- und Schlüsseltextraum $\mathbb{Z}_m^n$ heißt* affin linear*, wenn ihre sämtlichen Verschlüsselungsfunktionen affin linear sind. Sie heißt linear, wenn alle Verschlüsselungsfunktionen linear sind.*

Wir beschreiben affin lineare Blockchiffren explizit. Die Verschlüsselungsfunktionen sind affin linear, also von der Form

$$E : \mathbb{Z}_m^n \to \mathbb{Z}_m^n, \quad \mathbf{v} \mapsto A\mathbf{v} + \mathbf{b} \bmod m$$

mit $A \in \mathbb{Z}^{(n,n)}$ und $b \in \mathbb{Z}^n$. Außerdem ist E nach Theorem 4.6.2 bijektiv, also ist nach Theorem 4.11.10 $\det A$ teilerfremd zu m. Die Verschlüsselungsfunktion E ist durch das Paar $(A, \mathbf{b})$ eindeutig bestimmt. Dieses Paar kann

als Schlüssel genommen werden. Die entsprechende Entschlüsselungsfunktion ist nach den Ergebnissen von Abschnitt 4.11.6

$$D : \mathbb{Z}_m^n \to \mathbb{Z}_m^n, \quad \mathbf{v} \mapsto A'(\mathbf{v} - \mathbf{b}) \bmod m,$$

wobei $A' = (a' \operatorname{adj} A) \bmod m$ und a' das Inverse von $\det A \bmod m$ ist.

4.13 Vigenère-, Hill- und Permutationschiffre

Wir geben zwei Beispiele für affin lineare Chiffren.

Die Vigenère Chiffre ist nach Blaise de Vigenère benannt, der im 16. Jahrhundert lebte. Der Schlüsselraum ist $\mathcal{K} = \mathbb{Z}_m^n$. Ist $\mathbf{k} \in \mathbb{Z}_m^n$, dann ist

$$E_{\mathbf{k}} : \mathbb{Z}_m^n \to \mathbb{Z}_m^n, \quad \mathbf{v} \mapsto \mathbf{v} + \mathbf{k} \bmod m$$

und

$$D_{\mathbf{k}} : \mathbb{Z}_m^n \to \mathbb{Z}_m^n, \quad \mathbf{v} \mapsto \mathbf{v} - \mathbf{k} \bmod m.$$

Die Abbildungen sind offensichtlich affin linear. Der Schlüsselraum hat m^n Elemente.

Eine anderes klassisches Verschlüsselungsverfahren ist die *Hill-Chiffre*, die 1929 von Lester S. Hill erfunden wurde. Der Schlüsselraum $\mathcal{K}$ ist die Menge aller Matrizen $A \in \mathbb{Z}_m^{(n,n)}$ mit $\gcd(\det A, m) = 1$. Für $A \in \mathcal{K}$ ist

$$E_A : \mathbb{Z}_m^n \to \mathbb{Z}_m^n, \quad \mathbf{v} \mapsto A\mathbf{v} \bmod m. \tag{4.6}$$

Die Hill-Chiffre ist also die allgemeinste lineare Blockchiffre.

Zuletzt zeigen wir noch, daß die Permutationschiffre linear ist. Sei $\pi \in S_n$ und seien $\mathbf{e}_i$, $1 \le i \le n$, die Einheitsvektoren der Länge n, also die Zeilenvektoren der Einheitsmatrix. Sei ferner E_π die $n \times n$-Matrix, deren i-te Zeile $\mathbf{e}_{\pi(i)}$ ist, $1 \le i \le n$. Diese Matrix erhält man aus der Einheitsmatrix, indem man ihre Zeilen gemäß der Permutation π vertauscht. In der j-ten Spalte von E_π steht also der Einheitsvektor $\mathbf{e}_{\pi(j)}$. Dann gilt für jeden Vektor $\mathbf{v} = (v_1, \ldots, v_n) \in \Sigma^n$

$$(v_{\pi(1)}, \ldots, v_{\pi(n)}) = E_\pi \mathbf{v}.$$

Die Permutationschiffre ist eine lineare Chiffre, also ein Spezialfall der Hill-Chiffre.

4.14 Kryptoanalyse affin linearer Blockchiffren

Wir zeigen, wie eine affin lineare Blockchiffre mit Alphabet $\mathbb{Z}_m$ und Blocklänge n mittels einer Known-Plaintext-Attacke gebrochen werden kann.

Die Ausgangssituation: Ein Schlüssel ist fixiert worden. Die zugehörige Verschlüsselungsfunktion ist nach den Ergebnissen von Abschnitt 4.12 von der Form

$$E : \mathbb{Z}_m^n \to \mathbb{Z}_m^n, \quad \mathbf{v} \mapsto A\mathbf{v} + \mathbf{b} \bmod m$$

mit $A \in \mathbb{Z}^{(n,n)}$ und $\mathbf{b} \in \mathbb{Z}^n$. Der Angreifer will den Schlüssel $(A, \mathbf{b})$ bestimmen.

Dazu verwendet er $n+1$ Klartexte $\mathbf{w}_i$, $0 \le i \le n$, und die zugehörigen Schlüsseltexte $\mathbf{c}_i = A\mathbf{w}_i + \mathbf{b} \bmod m$, $0 \le i \le n$. Dann ist

$$\mathbf{c}_i - \mathbf{c}_0 \equiv A(\mathbf{w}_i - \mathbf{w}_0) \bmod m$$

Ist W die Matrix

$$W = (\mathbf{w}_1 - \mathbf{w}_0, \ldots, \mathbf{w}_n - \mathbf{w}_0) \bmod m$$

deren Spalten die Differenzen $(\mathbf{w}_i - \mathbf{w}_0) \bmod m$, $1 \le i \le n$, sind, und ist C die Matrix

$$C = (\mathbf{c}_1 - \mathbf{c}_0, \ldots, \mathbf{c}_n - \mathbf{c}_0) \bmod m,$$

deren Spalten die Differenzen $(\mathbf{c}_i - \mathbf{c}_0) \bmod m$, $1 \le i \le n$, sind, dann gilt

$$AW \equiv C \bmod m.$$

Ist $\det W$ teilerfremd zu m, so ist

$$A \equiv C(w' \operatorname{adj} W) \bmod m,$$

wobei w' das Inverse von $\det W \bmod m$ ist. Weiter ist

$$\mathbf{b} = \mathbf{c}_0 - A\mathbf{w}_0.$$

Damit ist der Schlüssel aus $n+1$ Paaren von Klar- und Schlüsseltexten bestimmt worden. Ist die Chiffre sogar linear, so kann man $\mathbf{w}_0 = \mathbf{c}_0 = \mathbf{0}$ setzen, und es ist $\mathbf{b} = 0$.

Beispiel 4.14.1. Wir zeigen, wie eine Hill-Chiffre mit Blocklänge 2 gebrochen werden kann. Angenommen, man weiß, daß HAND in FOOT verschlüsselt wird. Damit wird $\mathbf{w}_1 = (7, 0)$ in $\mathbf{c}_1 = (5, 14)$ und $\mathbf{w}_2 = (13, 3)$ in $\mathbf{c}_2 = (14, 19)$ verschlüsselt. Wir erhalten also $W = \begin{pmatrix} 7 & 13 \\ 0 & 3 \end{pmatrix}$ und $C = \begin{pmatrix} 5 & 14 \\ 14 & 19 \end{pmatrix}$. Es ist $\det W = 21$ teilerfremd zu 26. Das Inverse von 21 mod 26 ist 5. Also ist

$$A = 5C(\operatorname{adj} W) \bmod 26 = 5 * \begin{pmatrix} 5 & 14 \\ 14 & 19 \end{pmatrix} \begin{pmatrix} 3 & 13 \\ 0 & 7 \end{pmatrix} \bmod 26 = \begin{pmatrix} 23 & 9 \\ 2 & 15 \end{pmatrix}.$$

Tatsächlich ist $AW = C$.

4.15 Sichere Blockchiffren

Die Konstruktion sicherer und effizienter Blockchiffren ist eine komplizierte Aufgabe.

4.15.1 Konfusion und Diffusion

Wichtige Konstruktionsprinzipien sind *Konfusion* und *Diffusion*. Sie wurden von Shannon vorgeschlagen, der die kryptographische Sicherheit im Rahmen seiner *Informationstheorie* untersuchte (siehe [73]). In Kapitel 5 geben wir eine Einführung in die Shannonsche Theorie.

Die Konfusion einer Blockchiffre ist groß, wenn die statistische Verteilung der Chiffretexte in einer so komplizierten Weise von der Verteilung der Klartexte abhängt, daß ein Angreifer diese Abhängigkeit nicht ausnutzen kann. Die Verschiebungschiffre hat zum Beispiel viel zu geringe Konfusion. Die Wahrscheinlichkeitsverteilung der Klartextzeichen überträgt sich unmittelbar auf die Chiffretextzeichen.

Die Diffusion einer Blockchiffre ist groß, wenn jedes einzelne Bit des Klartextes und jedes einzelne Bit des Schlüssel möglichst viele Bits des Chiffretextes beeinflußt.

Man sieht, daß Konfusion und Diffusion intuitive aber nicht mathematisch formalisierte Begriffe sind. Sie zeigen aber, wie sichere Blockchiffren konstruiert werden könnten.

Neben den Prinzipien Konfusion und Diffusion postuliert Shannon, daß eine sichere Blockchiffre gegen alle bekannten Angriffe resistent sein muß. Wie wir im Abschnitt 4.14 gesehen haben, dürfen sichere Blockchiffren zum Beispiel nicht affin linear sein. Aber das genügt nicht. Wir geben einen Überblick über bekannte Angriffe. Mehr Details findet man in [41].

4.15.2 Exhaustive Key Search

Der einfachste Ciphertext-Only-Angriff auf eine Blockchiffre ist die *vollständige Suche* oder englisch *exhaustive search*. Dieser Angriff wurde bereits in Abschnitt 4.3.1 beschrieben. Bei gegebenen Chiffretext probiert der Angreifer alle Schlüsselx aus bis ein sinnvoller Chiffretext gefunden wurde. Das funktioniert, wenn der Schlüsselraum klein genug ist. Es ist heute zum Beispiel möglich, den langjährigen Verschlüsselungsstandard DES (siehe Kapitel DES) mit diesem Angriff zu brechen. Der Schlüsselraum dieser Blockchiffre hat 2^{56} Elemente. In [82] stellt M. Wiener einen Spezialcomputer vor, der im Durchschnitt eine halbe Stunde braucht, um einen DES-Schlüssel durch eine vollständige Suche zu finden. Noch einfacher ist die vollständige Suche, wenn ein Klartext mit dem zugehörigen Schlüsseltext bekannt ist. Dann braucht dieser Klartext nur mit allen möglichen Schlüsseln verschlüsselt zu werden, bis der Chiffretext gefunden ist.

4.15.3 Time-Memory Trade-Off

Ist ein Klartext x mit dem zugehörigen Chiffretext c bekannt, dann kann die Rechenzeit für die vollständige Suche beschleunigt werden, wenn entsprechend mehr Speicherplatz verwendet wird. Hat der Schlüsselraum N Elemente dann benötigt der Time-Memory-Trade-Off-Algorithmus von M. Hellman [36] Zeit und Platz $\mathrm{O}(N^{2/3})$ um den geheimen Schlüssel, der bei der Verschlüsselung von x verwendet wurde, zu finden. Der Algorithmus erfordert eine Vorberechnung, die Zeit $\mathrm{O}(N)$ braucht. Eine Verallgemeinerung dieser Strategie findet man in [30].

Wir beschreiben den Algorithmus. Der Schlüsselraum $\mathcal{K}$ der Blockchiffre, die angegriffen wird, hat N Elemente. Die Blockchiffre hat Blocklänge n. Das verwendete Alphabet ist Σ. Wir nehmen an, dass ein Klartext x und der zugehörige schlüsseltext c bekannt sind. Gesucht ist der verwendete Schlüssel. Sei

$$m = \lceil N^{1/3} \rceil.$$

Wähle m Funktionen

$$g_k : \Sigma^n \to \mathcal{K}, \quad 1 \le k \le m$$

zufällig. Die Funktionen g_k machen aus Klartexten oder Chiffretexten Schlüssel.

Wähle m Schlüssel K_i, $1 \le i \le m$, zufällig und setze

$$K_k(i,0) = K_i, 1 \le i,k \le m.$$

Damit können jetzt folgende Schlüsseltabellen berechnet werden:

$$K_k(i,j) = g_k(E_{K(i,j-1}(x)), \quad 1 \le i,j,k \le m. \tag{4.7}$$

Die k-te Schlüsseltabelle hat also die in Tabelle 4.3 dargestellte Form.

$K_k(1,0)$	$\overset{g_k}{\to}$	$K_k(1,1)$	$\overset{g_k}{\to}$	$\cdots$	$\overset{g_k}{\to}$	$K_k(1,m)$
$K_k(2,0)$	$\overset{g_k}{\to}$	$K_k(2,1)$	$\overset{g_k}{\to}$	$\cdots$	$\overset{g_k}{\to}$	$K_k(2,m)$
$\cdots$						$\cdots$
$K_k(m,0)$	$\overset{g_k}{\to}$	$K_k(m,1)$	$\overset{g_k}{\to}$	$\cdots$	$\overset{g_k}{\to}$	$K_k(m,m)$

Tabelle 4.3. Die k-te Schlüsseltaxbelle im Time-Memory-Trade-Off

Die Anzahl der Tabelleneinträge ist ungefähr N.

Um den geheimen Schlüssel zu finden berechnen wir

$$g_k(c), 1 \le k \le m.$$

Wir prüfen, ob

$$K_k(i,j) = g_k(c)$$

gilt für Indizes $i, j, k \in \{1, \ldots, m\}$. Da

$$K_k(i,j) = g_k(E_{K_k(i,j-1)}(x))$$

gilt, ist es gut möglich, dass $K_k(i, j-1)$ der gesuchte Schlüssel ist. Wir überprüfen also, ob

$$c = E_{K_k(i,j-1}(x)$$

gilt. Wenn ja, ist der gesuchte Schlüssel $K(i, j-1)$.

So wie das Verfahren bis jetzt beschrieben worden ist, erfordert es Berechnung von ungefähr $N^{1/3}$ Schlüsseln aber die Speicherung von N Schlüsseln. Damit ist der Speicherplatzbedarf zu groß. Statt dessen werden tatsächlich nur die letzten Spalten der Tabellen, also die Werte $K_k(i,m)$, $1 \leq i \leq m$ gespeichert.

Um den Schlüssel zu finden, der aus einem Klartext x einen Schlüsseltext c macht, berechnen wir

$$K(1,k) = g_k(c), 1 \leq k \leq m$$

und dann

$$K(j,k) = g_k(E_{K(j-1,k)}(x), \quad 1 \leq k \leq m, 1 < j \leq m.$$

Wir versuchen, einen dieser Schlüssel in unserer Tabelle zu finden. Wir suchen also Indizes $i, j, k \in \{1, \ldots, m\}$ mit der Eigenschaft

$$K(j,k) = K_k(i,m).$$

Sobald diese gefunden sind, liegt es nahe zu vermuten, daß $K(1,k) = K_k(i, m-j+1)$ und der verwendete Schlüssel $K_k(i, m-j)$ ist. Wir konstruieren diesen Schlüssel und prüften, ob $c = E_{K_k(i,m-j)}(x)$ ist. Wenn ja, ist der gesuchte Schlüssel gefunden. Unter geeigneten Voraussetzungen kann man zeigen, daß auf diese Weise der gesuchte Schlüssel mit hoher Wahrscheinlichkeit gefunden wird. Die Anzahl der untersuchten Schlüssel ist ungefähr $N^{2/3}$. Speichert man die letzten Spalten der Tabellen als Hashtabellen, ist die gesamte Laufzeit $O(N^{2/3})$, wenn man davon ausgeht, daß die Berechnung jedes einzelnen Schlüssels und jeder Tabellenzugriff Zeit $O(1)$ benötigt.

4.15.4 Differentielle Kryptoanalyse

Die differentielle Kryptoanalyse [11] wurde 1990 von Biham und Shamir erfunden, um den DES anzugreifen. Diese Technik kann aber gegen Blockchiffren im allgemeinen angewendet werden. Differentielle Angriffe wurden zum Beispiel auch auf IDEA, SAFER K und Skipjack angewendet. Die Referenzen finden sich in [41].

Die differentielle Kryptoanalyse ist ein Chosen-Plaintext-Angriff. Aus vielen Paaren Klartext-Schlüsseltext versucht der Angreifer den verwendeten Schlüssel zu bestimmen. Dabei verwendet er die "Differenzen" der Klar- und Schlüsseltexte, d.h sind p und p' Klartexte und sind c und c' die zugehörigen Schlüsseltexte, dann berechnet der Angreifer $p \oplus p'$ und $c \oplus c'$. Er nutzt aus, daß in vielen Verschlüsselungsverfahren aus dem Paar $(p \oplus p', c \oplus c')$ Rückschlüsse auf den verwendeten Schlüssel gezogen werden können.

4.16 Übungen

Übung 4.16.1. Der Schlüsseltext JIVSOMPMQUVQA wurde mit der Verschiebungschiffre erzeugt. Ermitteln Sie den Schlüssel und den Klartext.

Übung 4.16.2. Zeigen Sie, daß auf folgende Weise ein Kryptosystem definiert ist.

Sei w ein String über $\{\text{A,B},\ldots,\text{Z}\}$. Wähle zwei Schlüssel k_1 und k_2 für die Verschiebungschiffre. Verschlüssele Zeichen mit ungeradem Index unter Verwendung von k_1 und die mit geradem Index unter Verwendung von k_2. Dann kehre die Reihenfolge der Zeichen um.

Bestimmen Sie den Klartextraum, den Schlüsseltextraum und den Schlüsselraum.

Übung 4.16.3. Zeigen Sie, daß die Verschlüsselungsfunktionen eines Kryptosystems immer injektiv sind.

Übung 4.16.4. Bestimmen Sie die Anzahl der Wörter der Länge n über einem Alphabet Σ, die sich nicht ändern, wenn sie umgekehrt werden.

Übung 4.16.5. Sei Σ ein Alphabet. Zeigen Sie, daß die Menge Σ^* zusammen mit der Konkatenation eine Halbgruppe mit neutralem Element ist. Welches ist das neutrale Element? Ist diese Halbgruppe sogar eine Gruppe?

Übung 4.16.6. Wieviele verschiedene Verschlüsselungsfunktionen kann eine Blockchiffre mit Alphabet $\{0,1\}$ und Blocklänge n höchstens haben?

Übung 4.16.7. Welches der folgenden Systeme ist ein Verschlüsselungsverfahren? Geben Sie gegebenenfalls Klartextraum, Schlüsseltextraum, Schlüsselraum, Verschlüsselungs- und Entschlüsselungsfunktion an. In der Beschreibung werden Buchstaben aus $\Sigma = \{\text{A,B,}\ldots\text{,Z}\}$ gemäß Tabelle 4.1 durch Zahlen ersetzt.

1. Jeder Buchstabe σ aus Σ wird durch $k\sigma \bmod 26$ ersetzt, $k \in \{1,2,\ldots,26\}$.
2. Jeder Buchstabe σ aus Σ wird durch $k\sigma \bmod 26$ ersetzt, $k \in \{1,2,\ldots,26\}$, $\gcd(k,26) = 1$.

Übung 4.16.8. Geben Sie ein Beispiel für ein Kryptosystem, das Verschlüsselungsfunktionen besitzt, die zwar injektiv aber nicht surjektiv sind.

Übung 4.16.9. Bestimmen Sie die Anzahl der Bitpermutationen der Menge $\{0,1\}^n$, $n \in \mathbb{N}$. Bestimmen Sie auch die Anzahl der zirkulären Links- und Rechtsshifts von $\{0,1\}^n$.

Übung 4.16.10. Eine *Transposition* ist eine Permutation, die zwei Elemente vertauscht und die anderen unverändert läßt. Zeigen Sie, daß jede Permutation als Komposition von Transpositionen dargestellt werden kann.

Übung 4.16.11. Geben Sie eine Permutation von $\{0,1\}^n$ an, die keine Bitpermutation ist.

Übung 4.16.12. Geben Sie eine Permutation von $\{0,1\}^n$ an, die nicht affin linear ist.

Übung 4.16.13. Sei X eine Menge. Man zeige, daß die Menge $S(X)$ der Permutationen von X eine Gruppe bezüglich der Hintereinanderausführung ist. Man zeige auch, daß diese Gruppe im allgemeinen nicht kommutativ ist.

Übung 4.16.14. Entschlüsseln Sie 111111111111 im ECB-Mode, im CBC-Mode, im CFB-Mode und im OFB-Mode. Verwenden Sie die Permutationschiffre mit Blocklänge 3 und Schlüssel

$$k = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix}.$$

Der Initialisierungsvektor ist 000. Im OFB- und CFB-Mode verwenden Sie $r = 2$.

Übung 4.16.15. Verschlüsseln Sie den String 101010101010 im ECB-Mode, im CBC-Mode, im CFB-Mode und im OFB-Mode. Verwenden Sie die Permutationschiffre mit Blocklänge 3 und Schlüssel

$$k = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix}.$$

Der Initialisierungsvektor ist 000. Im OFB- und CFB-Mode verwenden Sie $r = 2$.

Übung 4.16.16. Sei $k = 1010101$, $c = 1110011$ und $w = 1110001\ 1110001\ 1110001$. Verschlüsseln Sie w unter Verwendung der Stromchiffre aus Abschnitt 4.9.

Übung 4.16.17. Man zeige, daß man die Stromchiffre, die mit Hilfe eines linearen Schieberegisters erzeugt wird, auch mit einer Blockchiffre im OFB-Mode erzeugen kann, sofern die Länge der verschlüsselten Wörter ein Vielfaches der Blocklänge und $c_1 = 1$ ist.

Übung 4.16.18. Bestimmen Sie die Determinante der Matrix

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{pmatrix}.$$

Übung 4.16.19. Geben Sie eine geschlossene Formel für die Determinante einer 3×3-Matrix an.

Übung 4.16.20. Finden Sie eine injektive affin lineare Abbildung $(\mathbb{Z}/2\mathbb{Z})^3 \to (\mathbb{Z}/2\mathbb{Z})^3$ die $(1,1,1)$ auf $(0,0,0)$ abbildet.

Übung 4.16.21. Bestimmen Sie die Inverse der Matrix

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

mod 2.

Übung 4.16.22. Geben Sie einen Schlüssel für eine affin lineare Chiffre mit Alphabet {A,B,C,...,Z} und Blocklänge 3 an, mit dem "ROT" in "GUT" verschlüsselt wird.

5. Wahrscheinlichkeit und perfekte Geheimhaltung

Im vorigen Kapitel wurden eine Reihe von historischen Verschlüsselungsverfahren beschrieben. Es stellte sich heraus, daß sie alle affin linear und daher unsicher sind. Es fragt sich also, ob es wirklich sichere Verschlüsselungsverfahren gibt. Solche Systeme wurden von Claude Shannon [73] 1949 beschrieben. Wir beschreiben in diesem Kapitel, wie Shannon perfekt geheime Verschlüsselungsverfahren definiert hat und wie er ihre Existenz bewiesen hat. Es wird sich aber herausstellen, daß die perfekt geheimen Verschlüsselungsverfahren nicht besonders effizient sind. Außerdem sind perfekt geheime Verschlüsselungsverfahren nur gegen passive Angriffe sicher. Sie bieten keine Sicherheit gegen aktive Angreifer, die zum Beispiel versuchen einen Chiffretext so zu verändern, dass der entsprechende Klartext in ihrem Sinn modifiziert wird. Schutz gegen solche Angriffe wird in Kapitel 12 beschrieben.

Um die Theorie von Shannon wiederzugeben, führen wir einige Begriffe und Ergebnisse der elementaren Wahrscheinlichkeitstheorie auf.

5.1 Wahrscheinlichkeit

Sei S eine endliche nicht leere Menge. Wir nennen sie *Ergebnismenge*. Ihre Elemente heißen *Elementarereignisse*. Die Ergebnismenge braucht man, um die möglichen Ergebnisse von Experimenten zu modellieren.

Beispiel 5.1.1. Wenn man eine Münze wirft, erhält man entweder Zahl oder Wappen. Die entsprechende Ergebnismenge ist $S = \{Z, W\}$.

Wenn man würfelt, erhält man eine der Zahlen $1, 2, 3, 4, 5, 6$. Die Ergebnismenge ist dann $S = \{1, 2, 3, 4, 5, 6\}$.

Ein *Ereignis* (für S) ist eine Teilmenge der Ergebnismenge S. Das *sichere Ereignis* ist die Menge S selbst. Das *leere Ereignis* ist $\emptyset$. Zwei Ereignisse A und B *schließen sich gegenseitig aus*, wenn ihr Durchschnitt leer ist. Die Menge aller Ereignisse ist die *Potenzmenge* $P(S)$ von S.

Beispiel 5.1.2. Ein Ereignis beim Würfeln ist, eine gerade Zahl zu würfeln. Formal ist das Ereignis $\{2, 4, 6\}$. Es schließt das Ereignis $\{1, 3, 5\}$, eine ungerade Zahl zu würfeln, aus.

Eine *Wahrscheinlichkeitsverteilung* auf S ist eine Abbildung Pr, die jedem Ereignis eine reelle Zahl zuordnet, also

$$\Pr : P(S) \to \mathbb{R},$$

und die folgende Eigenschaften erfüllt:

1. $\Pr(A) \geq 0$ für alle Ereignisse A,
2. $\Pr(S) = 1$,
3. $\Pr(A \cup B) = \Pr(A) + \Pr(B)$ für zwei Ereignisse A und B, die sich gegenseitig ausschließen.

Ist A ein Ereignis, so heißt $\Pr(A)$ *Wahrscheinlichkeit* dieses Ereignisses. Die Wahrscheinlichkeit eines Elementarereignisses $a \in S$ ist $\Pr(a) = \Pr(\{a\})$.

Man sieht leicht, daß $\Pr(\emptyset) = 0$ ist. Außerdem folgt aus $A \subset B$, daß $\Pr(A) \leq \Pr(B)$ gilt. Daher ist $0 \leq \Pr(A) \leq 1$ für alle $A \in P(S)$. Ferner ist $\Pr(S \setminus A) = 1 - \Pr(A)$. Sind $A_1, \ldots, A_n$ Ereignisse, die sich paarweise ausschließen, dann gilt $\Pr(\cup_{i=1}^{n} A_i) = \sum_{i=1}^{n} \Pr(A_i)$.

Weil S eine endliche Menge ist, genügt es, eine Wahrscheinlichkeitsverteilung auf den Elementarereignissen zu definieren, denn es gilt $\Pr(A) = \sum_{a \in A} \Pr(a)$ für jedes Ereignis A.

Beispiel 5.1.3. Eine Wahrscheinlichkeitsverteilung auf der Menge $\{1, 2, 3, 4, 5, 6\}$, die einem Würfelexperiment entspricht, ordnet jedem Elementarereignis die Wahrscheinlichkeit $1/6$ zu. Die Wahrscheinlichkeit für das Ereignis, eine gerade Zahl zu würfeln, ist dann $\Pr(\{2, 4, 6\}) = \Pr(2) + \Pr(4) + \Pr(6) = 1/6 + 1/6 + 1/6 = 1/2$.

Die Wahrscheinlichkeitsverteilung, die jedem Elementarereignis $a \in S$ die Wahrscheinlichkeit $P(a) = 1/|S|$ zuordnet, heißt *Gleichverteilung*.

5.2 Bedingte Wahrscheinlichkeit

Sei S eine Ergebnismenge und sei Pr eine Wahrscheinlichkeitsverteilung auf S. Wir erläutern die bedingte Wahrscheinlichkeit erst an einem Beispiel.

Beispiel 5.2.1. Wir modellieren Würfeln mit einem Würfel. Die Ergebnismenge ist $\{1, 2, 3, 4, 5, 6\}$ und Pr ordnet jedem Elementarereignis die Wahrscheinlichkeit $1/6$ zu. Angenommen, man weiß, daß Klaus eine der Zahlen $4, 5, 6$ gewürfelt hat. Man weiß also, daß das Ereignis $B = \{4, 5, 6\}$ eingetreten ist. Unter dieser Voraussetzung möchte man die Wahrscheinlichkeit dafür ermitteln, daß Klaus eine gerade Zahl gewürfelt hat. Da jedes Ereignis aus B gleich wahrscheinlich ist, hat jedes die Wahrscheinlichkeit $1/3$. Da zwei Zahlen in B gerade sind, ist die Wahrscheinlichkeit dafür, daß eine gerade Zahl gewürfelt wurde, $2/3$.

Definition 5.2.2. *Sind A und B Ereignisse und $Pr(B) > 0$. Die Wahrscheinlichkeit für "A unter der Bedingung B" ist definiert als*

$$Pr(A|B) = \frac{Pr(A \cap B)}{Pr(B)}.$$

Diese Definition kann man folgendermaßen verstehen: Voraussetzung ist, daß das Ereignis B sicher eintritt, also nur Elementarereignisse aus B. Die Wahrscheinlichkeit eines Elementarereignisses x aus B ist dann $\Pr(x)/\Pr(B)$. Dann hat das Ereignis B die modifizierte Wahrscheinlichkeit 1, ist also sichr. Wir wollen wissen, wie wahrscheinlich es unter dieser Voraussetzung ist, daß A eintritt, also ein Elementarereignis x aus $A \cap B$. Diese Wahrscheinlichkeit ist $\Pr(A \cap B)/\Pr(B)$.

Zwei Ereignisse A und B heißen *unabhängig*, wenn

$$\Pr(A \cap B) = \Pr(A)\Pr(B)$$

gilt. Diese Bedingung ist äquivalent zu

$$\Pr(A|B) = \Pr(A).$$

Beispiel 5.2.3. Wenn man zwei Münzen wirft, ist das Ereignis, mit der ersten Münze "Zahl" zu werfen, unabhängig von dem Ereignis, mit der zweiten Münze "Zahl" zu werfen. Die Wahrscheinlichkeit dafür, mit beiden Münzen "Zahl" zu werfen, ist nämlich 1/4. Die Wahrscheinlichkeit dafür, mit der ersten Münze "Zahl" zu werfen, ist gleich der Wahrscheinlichkeit, mit der zweiten Münze "Zahl" zu werfen, und die ist 1/2. Wie in der Definition gefordert ist $1/4 = (1/2)(1/2)$.

Wenn man die Münzen zusammenlötet, so daß sie entweder beide "Zahl" oder beide "Kopf" zeigen, sind die Wahrscheinlichkeiten nicht mehr unabhängig. Die Wahrscheinlichkeit dafür, mit der ersten Münze "Zahl" zu werfen ist gleich der Wahrscheinlichkeit, mit der zweiten Münze "Zahl" zu werfen, und die ist 1/2. Die Wahrscheinlichkeit dafür, mit beiden Münzen "Zahl" zu werfen, ist aber auch 1/2.

Wir formulieren und beweisen den Satz von Bayes.

Theorem 5.2.4. *Sind A und B Ereignisse mit $Pr(A) > 0$ und $Pr(B) > 0$, so gilt*

$$Pr(B)Pr(A|B) = Pr(A)Pr(B|A)$$

Beweis. Nach Definition gilt $\Pr(A|B)\Pr(B) = \Pr(A \cap B)$ und $\Pr(B|A)\Pr(A) = \Pr(A \cap B)$. Daraus folgt die Behauptung unmittelbar. □

5.3 Geburtstagsparadox

Ein gutes Beispiel für wahrscheinlichkeitstheoretische Überlegungen ist das Geburtstagsparadox. Wieviele Leute müssen in einem Raum sein, damit man eine gute Chance hat, daß wenigstens zwei von ihnen am gleichen Tag Geburtstag haben? Es sind erstaunlich wenige, jedenfalls deutlich weniger als 365, wie wir nun zeigen werden.

Wir machen eine etwas allgemeinere Analyse. Es gibt n verschiedene Geburtstage. Im Raum sind k Personen. Ein Elementarereignis ist ein Tupel $(g_1, \ldots, g_k) \in \{1, 2, \ldots, n\}^k$. Tritt es ein, so hat die i-te Person den Geburtstag g_i, $1 \leq i \leq k$. Es gibt also n^k Elementarereignisse. Wir nehmen an, daß alle Elementarereignisse gleich wahrscheinlich sind. Jedes Elementarereignis hat also die Wahrscheinlichkeit $1/n^k$.

Wir möchten die Wahrscheinlichkeit dafür berechnen, daß wenigstens zwei Personen am gleichen Tag Geburtstag haben. Bezeichne diese Wahrscheinlichkeit mit p. Dann ist $q = 1 - p$ die Wahrscheinlichkeit dafür, daß alle Personen verschiedene Geburtstage haben. Diese Wahrscheinlichkeit q rechnen wir aus. Das Ereignis E, das uns interessiert, ist die Menge aller Vektoren $(g_1, \ldots, g_k) \in \{1, 2, \ldots, n\}^k$, deren sämtliche Koordinaten verschieden sind. Alle Elementarereignisse haben die gleiche Wahrscheinlichkeit $1/n^k$. Die Wahrscheinlichkeit für E ist also die Anzahl der Elemente in E multipliziert mit $1/n^k$. Die Anzahl der Vektoren in $\{1, \ldots, n\}^k$ mit lauter verschiedenen Koordinaten bestimmt sich so: Auf der ersten Position können n Zahlen stehen. Liegt die erste Position fest, so können auf der zweiten Position noch $n - 1$ Zahlen stehen usw. Es gilt also

$$|E| = \prod_{i=0}^{k-1} (n - i).$$

Die gesuchte Wahrscheinlichkeit ist

$$q = \frac{1}{n^k} \prod_{i=0}^{k-1} (n - i) = \prod_{i=1}^{k-1} (1 - \frac{i}{n}). \tag{5.1}$$

Nun gilt aber $1 + x \leq e^x$ für alle reellen Zahlen. Aus (5.1) folgt daher

$$q \leq \prod_{i=1}^{k-1} e^{-i/n} = e^{-\sum_{i=1}^{k-1} i/n} = e^{-k(k-1)/(2n)}. \tag{5.2}$$

Ist

$$k \geq (1 + \sqrt{1 + 8n \log 2})/2 \tag{5.3}$$

so folgt aus (5.2), daß $q \leq 1/2$ ist. Dann ist die Wahrscheinlichkeit $p = 1 - q$ dafür, daß zwei Personen im Raum am gleichen Tag Geburtstag haben $\geq 1/2$. Für $n = 365$ genügt $k = 23$, damit $q \leq 1/2$ ist. Sind also 23 Personen im

Raum, so haben mit Wahrscheinlichkeit $\geq 1/2$ wenigstens zwei am gleichen Tag Geburtstag. Allgemein genügen etwas mehr als $\sqrt{n}$ viele Personen, damit zwei am gleichen Tag Geburtstag haben.

5.4 Perfekte Geheimhaltung

Wir werden nun perfekt geheime Kryptosysteme definieren.

Wir gehen von folgendem Szenario aus: Alice benutzt ein Kryptosystem, um verschlüsselte Nachrichten an Bob zu schicken. Wenn Alice an Bob eine verschlüsselte Nachricht schickt, kann Oskar den Schlüsseltext lesen. Aus dem Schlüsseltext versucht Oskar, Informationen über den Klartext zu gewinnen. Verwendet Alice ein perfekt geheimes System, soll das unmöglich sein. Dies soll nun formalisiert und die perfekt geheimen Systeme sollen charakterisiert werden.

Alice benutzt ein Kryptosystem mit endlichem Klartextraum $\mathcal{P}$, endlichem Schlüsseltextraum $\mathcal{C}$ und endlichem Schlüsselraum $\mathcal{K}$, Verschlüsselungsfunktionen E_k, $k \in \mathcal{K}$, und Entschlüsselungsfunktionen D_k, $k \in \mathcal{K}$.

Wir nehmen an, daß die Klartexte gemäß einer Wahrscheinlichkeitsverteilung $\mathrm{Pr}_{\mathcal{P}}$ auftauchen. Diese Wahrscheinlichkeitsverteilung hängt von der verwendeten Sprache ab. Zusätzlich kann $\mathrm{Pr}_{\mathcal{P}}$ vom Anwendungskontext abhängen. Sind Alice und Bob zum Beispiel Lehrer, dann ist die Wahrscheinlichkeit dafür, daß in einer Nachricht von Alice an Bob das Wort "Schüler" vorkommt, ziemlich groß.

Alice wählt für jeden neuen Klartext einen neuen Schlüssel. Die Auswahl folgt einer Wahrscheinlichkeitsverteilung $\mathrm{Pr}_{\mathcal{K}}$. Aus $\mathrm{Pr}_{\mathcal{P}}$ und $\mathrm{Pr}_{\mathcal{K}}$ erhält man eine Wahrscheinlichkeitsverteilung auf der Ergebnismenge $\mathcal{P} \times \mathcal{K}$. Für einen Klartext p und einen Schlüssel k ist $\mathrm{Pr}(p,k)$ die Wahrscheinlichkeit dafür, daß der Klartext p erscheint und mit dem Schlüssel k verschlüsselt wird. Es gilt

$$\mathrm{Pr}(p,k) = \mathrm{Pr}_{\mathcal{P}}(p)\mathrm{Pr}_{\mathcal{K}}(k) \tag{5.4}$$

und dadurch ist Pr festgelegt. Ab jetzt betrachten wir nur noch die Ergebnismenge $\mathcal{P} \times \mathcal{K}$. Für $p \in \mathcal{P}$ bezeichne p das Ereignis, daß p verschlüsselt wird, also das Ereignis $\{(p,k) : k \in \mathcal{K}\}$. Dann gilt

$$\mathrm{Pr}(p) = \mathrm{Pr}_{\mathcal{P}}(p).$$

Für $k \in \mathcal{K}$ bezeichne k das Ereignis, daß k verwendet wird, also das Ereignis $\{(p,k) : p \in \mathcal{P}\}$. Dann gilt

$$\mathrm{Pr}(k) = \mathrm{Pr}_{\mathcal{K}}(k).$$

Nach (5.4) sind die Ereignisse p und k unabhängig. Für $c \in \mathcal{C}$ bezeichne c das Ereignis, daß das Ergebnis der Verschlüsselung c ist, also das Ereignis $\{(p,k) : E_k(p) = c\}$. Oskar kennt die Wahrscheinlichkeitsverteilung $\mathrm{Pr}_{\mathcal{P}}$ auf

den Klartexten, weil er weiß, welche Sprache Alice und Bob benutzen. Oskar sieht die Schlüsseltexte. Er kann Rückschlüsse auf die Klartexte ziehen, wenn Schlüsseltexte bestimmte Klartexte wahrscheinlicher machen als andere. Er lernt nichts über die Klartexte, wenn die Wahrscheinlichkeit dafür, daß ein bestimmter Klartext vorliegt, nicht davon abhängt, wie der Klartext verschlüsselt wurde. Dies rechtfertigt die folgende Definition.

Definition 5.4.1. *Das Kryptosystem heißt* perfekt geheim*, wenn die Ereignisse, daß ein bestimmter Schlüsseltext auftritt und daß ein bestimmter Klartext vorliegt, unabhängig sind, wenn also $Pr(p|c) = Pr(p)$ ist für alle Klartexte p und alle Schlüsseltexte c.*

Beispiel 5.4.2. Sei $\mathcal{P} = \{0, 1\}$, $\Pr(0) = 1/4$, $\Pr(1) = 3/4$. Weiter sei $\mathcal{K} = \{A, B\}$, $\Pr(A) = 1/4$, $\Pr(B) = 3/4$. Schließlich sei $\mathcal{C} = \{a, b\}$. Dann ist die Wahrscheinlichkeit dafür, daß das Zeichen 1 auftritt und mit dem Schlüssel B verschlüsselt wird, $\Pr(1)\Pr(B) = 9/16$. Die Verschlüsselungsfunktionen E_K arbeiten so:

$$E_A(0) = a, E_A(1) = b, E_B(0) = b, E_B(1) = a.$$

Die Wahrscheinlichkeit dafür, daß der Schlüsseltext a auftritt, ist $\Pr(a) = \Pr(0, A) + \Pr(1, B) = 1/16 + 9/16 = 5/8$. Die Wahrscheinlichkeit dafür, daß der Schlüsseltext b auftritt, ist $\Pr(b) = \Pr(1, A) + \Pr(0, B) = 3/16 + 3/16 = 3/8$.

Wir berechnen nun die bedingten Wahrscheinlichkeiten $\Pr(p|c)$ für alle Klartexte p und alle Schlüsseltexte c. Es ist $\Pr(0|a) = 1/10$, $\Pr(1|a) = 9/10$, $\Pr(0|b) = 1/2$, $\Pr(1|b) = 1/2$. Diese Ergebnisse zeigen, daß das beschriebene Kryptosystem nicht perfekt geheim ist. Wenn Oskar den Schlüsseltext a beobachtet, kann er ziemlich sicher sein, daß der zugehörige Klartext 1 war.

Wir formulieren und beweisen den berühmten Satz von Shannon.

Theorem 5.4.3. *Sei $|\mathcal{P}| = |\mathcal{K}| = |\mathcal{C}| < \infty$ und sei $Pr(p) > 0$ für jeden Klartext p. Das Kryptosystem ist genau dann perfekt geheim, wenn die Wahrscheinlichkeitsverteilung auf dem Schlüsselraum die Gleichverteilung ist und wenn es für jeden Klartext p und jeden Schlüsseltext c genau einen Schlüssel k gibt mit $E_k(p) = c$.*

Beweis. Angenommen, das Verschlüsselungssystem ist perfekt geheim. Sei p ein Klartext. Wenn es einen Schlüsseltext c gibt, für den es keinen Schlüssel k gibt mit $E_k(p) = c$, dann ist $\Pr(p) \neq \Pr(p|c) = 0$. Aber dies widerspricht der perfekten Geheimhaltung. Für jeden Schlüsseltext c gibt es also einen Schlüssel k mit $E_k(p) = c$. Da aber die Anzahl der Schlüssel gleich der Anzahl der Schlüsseltexte ist, gibt es für jeden Schlüsseltext c genau einen Schlüssel k mit $E_k(p) = c$. Dies beweist die zweite Behauptung.

Um die erste Behauptung zu beweisen, fixiere einen Schlüsseltext c. Für einen Klartext p sei $k(p)$ der eindeutig bestimmte Schlüssel mit $E_{k(p)}(p) = c$. Weil es genausoviele Klartexte wie Schlüssel gibt, ist

$$\mathcal{K} = \{k(p) : p \in \mathcal{P}\} \tag{5.5}$$

Wir zeigen, daß für alle $p \in \mathcal{P}$ die Wahrscheinlichkeit für $k(p)$ gleich der Wahrscheinlichkeit für c ist. Dann ist die Wahrscheinlichkeit für $k(p)$ unabhängig von p. Da aber nach (5.5) jeder Schlüssel k mit einem $k(p)$, $p \in \mathcal{P}$ übereinstimmt, sind alle Schlüssel gleich wahrscheinlich.

Sei $p \in \mathcal{P}$. Wir zeigen $\Pr(k(p)) = \Pr(c)$. Nach Theorem 5.2.4 gilt für jeden Klartext p

$$\Pr(p|c) = \frac{\Pr(c|p)\Pr(p)}{\Pr(c)} = \frac{\Pr(k(p))\Pr(p)}{\Pr(c)}. \tag{5.6}$$

Weil das Verschlüsselungssystem perfekt geheim ist, gilt $\Pr(p|c) = \Pr(p)$. Aus (5.6) folgt daher $\Pr(k(p)) = \Pr(c)$, und dies ist unabhängig von p.

Wir beweisen die Umkehrung. Angenommen, die Wahrscheinlichkeitsverteilung auf dem Schlüsselraum ist die Gleichverteilung und für jeden Klartext p und jeden Schlüsseltext c gibt es genau einen Schlüssel $k = k(p, c)$ mit $E_k(p) = c$. Dann folgt

$$\Pr(p|c) = \frac{\Pr(p)\Pr(c|p)}{\Pr(c)} = \frac{\Pr(p)\Pr(k(p,c))}{\sum_{q\in\mathcal{P}} \Pr(q)\Pr(k(q,c))}. \tag{5.7}$$

Nun ist $\Pr(k(p,c)) = 1/|\mathcal{K}|$, weil alle Schlüssel gleich wahrscheinlich sind. Außerdem ist

$$\sum_{q\in\mathcal{P}} \Pr(q)\Pr(k(q,c)) = \frac{\sum_{q\in\mathcal{P}} \Pr(q)}{|\mathcal{K}|} = \frac{1}{|\mathcal{K}|}.$$

Setzt man dies in (5.7) ein, so folgt $\Pr(p|c) = \Pr(p)$, wie behauptet. □

Beispiel 5.4.4. Aus Theorem 5.4.3 folgt, daß das Kryptosystem aus Beispiel 5.4.2 perfekt geheim wird, wenn man $\Pr(A) = \Pr(B) = 1/2$ setzt.

5.5 Das Vernam-One-Time-Pad

Das bekannteste Kryptosystem, dessen perfekte Geheimhaltung man mit Theorem 5.4.3 beweisen kann, ist das *Vernam-One-Time-Pad*. Sei n eine natürliche Zahl. Das Vernam-One-Time-Pad verschlüsselt Bitstrings der Länge n. Es ist $\mathcal{P} = \mathcal{C} = \mathcal{K} = \{0,1\}^n$. Die Verschlüsselungsfunktion zum Schlüssel $k \in \{0,1\}^n$ ist

$$E_k : \{0,1\}^n \to \{0,1\}^n, \quad p \mapsto p \oplus k.$$

Die Entschlüsselungsfunktion zum Schlüssel k sieht genauso aus.

Wenn Alice einen Klartext $p \in \{0,1\}^n$ verschlüsseln will, verwendet sie einen Schlüssel k, der gemäß einer Gleichverteilung zufällig aus der Menge $\{0,1\}^n$ gewählt wird, und berechnet den Schlüsseltext $p \oplus k$. Dieses Verschlüsselungssystem ist nach Theorem 5.4.3 perfekt geheim, weil auf dem Schlüsselraum die Gleichverteilung gewählt wurde und weil für jeden Klartext p und jeden Schlüsseltext c genau ein Schlüssel k existiert mit $c = p \oplus k$, nämlich $k = p \oplus c$.

Diese Idee, Texte zu verschlüsseln wurde 1917 von Gilbert Vernam erfunden und patentiert. Aber erst 1949 bewies Shannon, daß das Vernam-One-Time-Pad perfekt geheim ist.

Leider ist das One-Time-Pad nicht besonders effizient. Alice und Bob müssen für jeden neuen Block der Länge n einen neuen Schlüssel der Länge n zufällig erzeugen und austauschen. Daher kommt auch der Name One-Time-Pad. Jeder Schlüssel kann nur einmal verwendet werden.

Wird ein Schlüssel für mehrere Blöcke verwendet, ist das Verfahren nicht mehr perfekt geheim. Außerdem kann Oskar mit einer Known-Plaintext-Attacke den Schlüssel ermitteln. Er kann also den Schlüssel berechnen, wenn er einen Klartext p und den zugehörigen Schlüsseltext c kennt. Es ist dann nämlich $p \oplus c = p \oplus p \oplus k = k$. Das One-Time-Pad schützt auch nicht gegen aktive Angriffe. Das wird im folgenden Beispiel gezeigt.

Beispiel 5.5.1. Ein Bankkunde verschlüsselt seine Überweisungen mit dem One-Time-Pad. Wenn ein Angreifer weiß, an welcher Stelle der Überweisungsbetrag steht, kann er die entsprechenden Bits einfach verändern. Die Wahrscheinlichkeit dafür, daß der Überweisungsbetrag dann höher wird, ist ziemlich hoch.

In Kapitel 12 werden Gegenmaßnahmen gegen aktive Angriffe vorgestellt.

5.6 Zufallszahlen

Will man das Vernam-One-Time-Pad anwenden, braucht man eine Quelle für Zufallsbits. Ob es Zufall wirklich gibt, ist eine philosophische Frage. In der Praxis verwendet man Hardware- und Software-basierte Zufallsbit-Generatoren. Das sind Geräte oder Programme, die Zahlen in der Menge $\{0,1\}$ zufällig und gleichverteilt erzeugen. Ob die Zufallsbit-Generatoren Bits wirklich gleichverteilt erzeugen, überprüft man mit statistischen Tests.

Hardware-Zufallsbit-Generatoren nutzen z.B. die Zufälligkeit des radioaktiven Zerfalls oder andere physikalische Quellen aus. Software-Zufallsbit-Generatoren nutzen z.B. die verstrichene Zeit zwischen zwei Anschlägen des Keyboards aus. Einen Überblick findet man in [63].

Wenn man Zufallsbit-Generatoren in der Kryptographie einsetzt, ist es wichtig, daß Angreifer keine Möglichkeit haben, ihre Arbeit zu beobachten.

Es ist also von Vorteil, solche Generatoren in geheime Hardware-Bausteine zu integrieren.

Wir gehen im folgenden davon aus, daß wir einen Zufallsbit-Generator haben, der Bits zufällig und gleichverteilt erzeugt. Daraus kann man Zufallszahlen-Generatoren machen, wie jetzt erklärt wird.

Will man zufällig und gleichverteilt Zahlen in der Menge $\{0, 1, \ldots, m\}$ erzeugen, $m \in \mathbb{N}$, so setzt man $n = \operatorname{size} m = \lfloor \log m \rfloor + 1$. Man erzeugt dann n zufällige Bits $b_1, \ldots, b_n$. Ist die Zahl $a = \sum_{i=1}^{n} b_i 2^{n-i}$ größer als m, so vergißt man sie und erzeugt eine neue. Andernfalls ist a die erzeugte Zufallszahl. Man verifiziert leicht, daß mit diesem Verfahren tatsächlich zufällig und gleichverteilt Zahlen in der Menge $\{0, 1, \ldots, m\}$ erzeugt werden.

Will man zufällig und gleichverteilt n-Bit-Zahlen erzeugen, $n \in \mathbb{N}$, so erzeugt man $n-1$ zufällige Bits $b_2, \ldots, b_n$ und setzt $b_1 = 1$ und $a = \sum_{i=1}^{n} b_i 2^{n-i}$.

5.7 Pseudozufallszahlen

Wenn es zu aufwendig ist, Zufallszahlen zu erzeugen, dann verwendet man Pseudozufallszahlengeneratoren. Ein Pseudozufallszahlengenerator ist ein Algorithmus, der aus einer kurzen Folge von Zufallszahlen eine lange Folge berechnet, die zufällig "aussieht", die also nicht in Polynomzeit von einer wirklich zufälligen Folge unterschieden werden kann. Eine detailierte Beschreibung der entsprechenden Theorie findet man in [33]. Praktisch verwendete Pseudozufallszahlengeneratoren sind in [52] beschrieben.

5.8 Übungen

Übung 5.8.1. Sei S eine endliche Menge und Pr eine Wahrscheinlichkeitsverteilung auf S. Zeigen Sie:

1. $\Pr(\emptyset) = 0$.
2. Aus $A \subset B \subset S$ folgt $\Pr(A) \leq \Pr(B)$.

Übung 5.8.2. In einem Experiment wird m zufällig und gleichverteilt aus der Menge $\{1, 2, \ldots, 1000\}$ gewählt. Bestimmen Sie die Wahrscheinlichkeit dafür,

1. ein Quadrat zu erhalten;
2. eine Zahl mit i Primfaktoren zu bestimmen, $i \geq 1$.

Übung 5.8.3. Geben Sie die Ergebnismenge und die Wahrscheinlichkeitsverteilung an, die einem Wurf von zwei Münzen entspricht. Geben Sie das Ereignis "wenigstens eine Münze zeigt Kopf" an und berechnen Sie seine Wahrscheinlichkeit.

Übung 5.8.4. Bestimmen Sie die Wahrscheinlichkeit dafür, daß eine zufällig gewählte Abbildung $\{0,1\}^* \to \{0,1\}^*$ affin linear ist.

Übung 5.8.5. Es wird mit zwei Würfeln gewürfelt. Wie groß ist die Wahrscheinlichkeit dafür, daß beide Würfel ein verschiedenes Ergebnis zeigen unter der Bedingung, daß die Summe der Ergebnisse gerade ist?

Übung 5.8.6. Bestimmen Sie n so, daß die Wahrscheinlichkeit dafür, daß zwei von n Personen am gleichen Tag Geburtstag haben, wenigstens 9/10 ist.

Übung 5.8.7. Sei g ein Element einer Gruppe G. Wir wollen die Ordnung dieses Elementes ermitteln. Dazu wählen wir zufällig k ganze Zahlen e_i, $1 \leq i \leq k$, und berechnen $a_i = g^{e_i}$. Sobald zwei Exponenten e_i und e_j gefunden sind mit $g^{e_i} = g^{e_j}$ ist $g^{e_i - e_j} = 1$, also $e = e_i - e_j$ ein Vielfaches der Ordnung von g. Indem man dieses Vielfache faktorisiert, kann man gemäß Theorem 3.14.1 die Ordnung von g finden. Wie groß muß k sein, damit die Wahrscheinlichkeit dafür, daß ein solches Vielfaches gefunden wird, größer als 1/2 ist?

Übung 5.8.8. Zeigen Sie, daß die Verschiebungschiffre nicht perfekt geheim ist.

Übung 5.8.9. Angenommen, vierstellige Geheimnummern von EC-Karten werden zufällig verteilt. Wieviele Leute muß man versammeln, damit die Wahrscheinlichkeit dafür, daß zwei dieselbe Geheimnummer haben, wenigstens 1/2 ist?

Übung 5.8.10. Betrachten Sie die lineare Blockchiffre mit Blocklänge n und Alphabet $\{0,1\}^n$. Wählen Sie auf dem Schlüsselraum aller Matrizen $A \in \{0,1\}^{(n,n)}$ mit $\det(A) \equiv 1 \bmod 2$ die Gleichverteilung. Ist dieses Kryptosystem perfekt geheim, wenn jeder neue Klartext der Länge n mit einem neuen Schlüssel verschlüsselt wird?

6. Der DES-Algorithmus

In Kapitel 4 wurden Kryptosysteme eingeführt und einige historische symmetrische Verschlüsselungsverfahren beschrieben. Die Verfahren aus Kapitel 4 konnten aber alle gebrochen werden, weil sie affin linear sind. Perfekt sichere Systeme wurden in Kapitel 5 beschrieben. Sie stellten sich aber als ineffizient heraus. In diesem Kapitel wird das DES-Verfahren beschrieben. Das DES-Verfahren war viele Jahre lang der Verschlüsselungsstandard in den USA und wird weltweit eingesetzt. Das einfache DES-Verfahren gilt aber nicht mehr als sicher genug. Inzwischen vom US-amerikanischen National Institute of Standards als Nachfolger von DES das Rijndael-Verschlüsselungsverfahren ausgewählt [1], das im nächsten Kapitel beschrieben wird. Als sicher gilt aber nach wie vor die Dreifachvariante Triple-DES (siehe Abschnitt 4.7). Außerdem ist DES ein wichtiges Vorbild für neue symmetrische Verfahren.

6.1 Feistel-Chiffren

Der DES-Algorithmus ist eine sogenannte *Feistel-Chiffre*. Wir erläutern in diesem Abschnitt, wie Feistel-Chiffren funktionieren.

Benötigt wird eine Blockchiffre mit Alphabet $\{0,1\}$. Sei t die Blocklänge. Die Verschlüsselungsfunktion zum Schlüssel K sei f_K. Daraus kann man folgendermaßen eine Feistel-Chiffre konstruieren. Die Feistel-Chiffre ist eine Blockchiffre mit Blocklänge $2t$ und Alphabet $\{0,1\}$. Man legt einen Schlüsselraum $\mathcal{K}$ fest. Außerdem legt man eine *Rundenzahl* $r \geq 1$ fest und wählt eine Methode, die aus einem Schlüssel $k \in \mathcal{K}$ eine Folge $K_1, \ldots, K_r$ von Rundenschlüsseln konstruiert. Die Rundenschlüssel gehören zum Schlüsselraum der zugrundeliegenden Blockchiffre.

Die Verschlüsselungsfunktion E_k der Feistel-Chiffre zum Schlüssel $k \in \mathcal{K}$ funktioniert so: Sei p ein Klartext der Länge $2t$. Den teilt man in zwei Hälften der Länge t auf. Man schreibt also $p = (L_0, R_0)$. Dabei ist L_0 die linke Hälfte des Klartextes, und R_0 ist seine rechte Hälfte. Danach konstruiert man eine Folge $((L_i, R_i))_{1 \leq i \leq r}$ nach folgender Vorschrift:

$$(L_i, R_i) = (R_{i-1}, L_{i-1} \oplus f_{K_i}(R_{i-1})), \quad 1 \leq i \leq r. \tag{6.1}$$

Dann setzt man

$$E_k(L_0, R_0) = (R_r, L_r).$$

Die Sicherheit der Feistelchiffre hängt natürlich zentral von der Sicherheit der internen Blockchiffre ab. Deren Sicherheit wird aber durch iterierte Verwendung noch gesteigert.

Wir erläutern die Entschlüsselung der Feistel-Chiffre. Aus (6.1) folgt unmittelbar

$$(R_{i-1}, L_{i-1}) = (L_i, R_i \oplus f_{K_i}(L_i)), \quad 1 \leq i \leq r. \tag{6.2}$$

Daher kann man unter Verwendung der Schlüsselfolge $(K_r, K_{r-1}, \ldots, K_1)$ in r Runden das Paar (R_0, L_0) aus dem Schlüsseltext (R_r, L_r) zurückgewinnen. Die Feistel-Chiffre wird also entschlüsselt, indem man sie mit umgekehrter Schlüsselfolge auf den Schlüsseltext anwendet.

6.2 Der DES-Algorithmus

Das DES-Verschlüsselungsverfahren ist eine leicht modifizierte Feistel-Chiffre mit Alphabet $\{0, 1\}$ und Blocklänge 64. Wir erläutern seine Funktionsweise im Detail.

6.2.1 Klartext- und Schlüsselraum

Klartext- und Schlüsseltextraum des DES ist $\mathcal{P} = \mathcal{C} = \{0, 1\}^{64}$. Die DES-Schlüssel sind Bitstrings der Länge 64, die folgende Eigenschaft haben: Teilt man einen String der Länge 64 in acht Bytes auf, so ist jeweils das letzte Bit eines jeden Bytes so gesetzt, daß die Quersumme aller Bits im betreffenden Byte ungerade ist. Es ist also

$$\mathcal{K} = \{(b_1, \ldots, b_{64}) \in \{0, 1\}^{64} : \sum_{i=1}^{8} b_{8k+i} \equiv 1 \mod 2, 0 \leq k \leq 7\}.$$

Die ersten sieben Bits eines Bytes in einem DES-Schlüssel legen das achte Bit fest. Dies ermöglicht Korrektur von Speicher- und Übertragungsfehlern. In einem DES-Schlüssel sind also nur 56 Bits frei wählbar. Insgesamt gibt es $2^{56} \sim 7.2 * 10^{16}$ viele DES-Schlüssel. Der DES-Schlüssel für Ver- und Entschlüsselung ist derselbe.

Beispiel 6.2.1. Ein gültiger DES Schlüssel ist hexadezimal geschrieben

133457799BBCDFF1.

Seine Binärentwicklung findet sich in Tabelle 6.1.

0	0	0	1	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	1	0	1	1	1
0	1	1	1	1	0	0	1
1	0	0	1	1	0	1	1
1	0	1	1	1	1	0	0
1	1	0	1	1	1	1	1
1	1	1	1	0	0	0	1

Tabelle 6.1. Gültiger DES-Schlüssel

6.2.2 Die initiale Permutation

Wir werden jetzt den DES-Algorithmus im einzelnen beschreiben. Bei Eingabe eines Klartextwortes p arbeitet er in drei Schritten.

Zusätzlich zur Feistel-Verschlüsselung wird im ersten Schritt auf p eine *initiale Permutation* IP angewandt. Dies ist eine für das Verfahren fest gewählte, vom Schlüssel unabhängige, Bitpermutation auf Bitvektoren der Länge 64. Die Permutation IP und die entsprechende inverse Permutation findet man in Tabelle 6.2

IP							
58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7
IP^{-1}							
40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

Tabelle 6.2. Die initiale Permutation IP

Tabelle 6.2 ist folgendermaßen zu verstehen. Ist $p \in \{0,1\}^{64}$, $p = p_1p_2p_3 \dots p_{64}$, dann ist $\mathrm{IP}(p) = p_{58}p_{50}p_{42} \dots p_7$.

Auf das Ergebnis dieser Permutation wird eine 16-Runden Feistel-Chiffre angewendet. Zuletzt wird die Ausgabe als

$$c = \mathrm{IP}^{-1}(R_{16}L_{16})$$

erzeugt.

6.2.3 Die interne Blockchiffre

Als nächstes wird die interne Blockchiffre beschrieben. Ihr Alphabet ist $\{0,1\}$, ihre Blocklänge ist 32 und ihr Schlüsselraum ist $\{0,1\}^{48}$. Wir erläutern, wie die Verschlüsselungsfunktion f_K zum Schlüssel $K \in \{0,1\}^{48}$ funktioniert (siehe Abbildung 6.1).

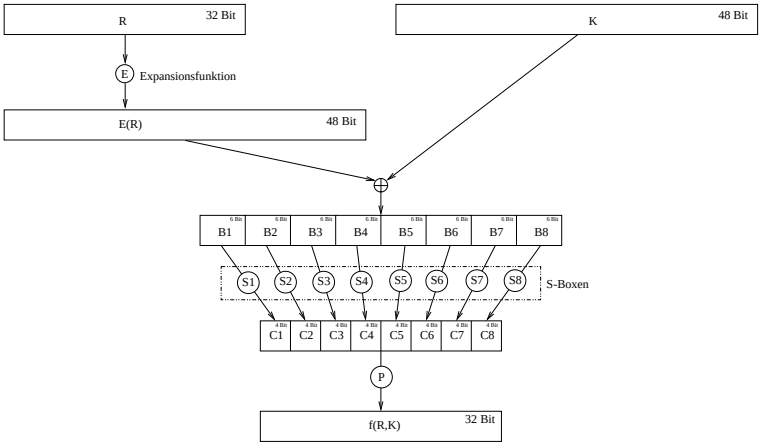


Abb. 6.1. Schema der f-Funktion im DES

Das Argument $R \in \{0,1\}^{32}$ wird mittels einer Expansionsfunktion $\mathrm{E} : \{0,1\}^{32} \to \{0,1\}^{48}$ verlängert. Diese Funktion ist in Tabelle 6.3 dargestellt. Ist $R = R_1R_2 \ldots R_{32}$, dann ist $E(R) = R_{32}R_1R_2 \ldots R_{32}R_1$.

E					
32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

P			
16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

Tabelle 6.3. Die Funktionen E und P

Anschließend wird der String $E(R) \oplus K$ gebildet und in 8 Blöcke B_i, $1 \leq i \leq 8$, der Länge 6 aufgeteilt. Es wird also

$$E(R) \oplus K = B_1 B_2 B_3 B_4 B_5 B_6 B_7 B_8 \tag{6.3}$$

gebildet mit $B_i \in \{0,1\}^6$, $1 \leq i \leq 8$. Im nächsten Schritt werden Funktionen

$$S_i : \{0,1\}^6 \to \{0,1\}^4, \quad 1 \leq i \leq 8$$

verwendet (die sogenannten S-Boxen), die unten noch genauer beschrieben sind. Mit diesen Funktionen wird der String

$$C = C_1 C_2 C_3 C_4 C_5 C_6 C_7 C_8$$

berechnet, wobei $C_i = S_i(B_i)$, $1 \leq i \leq 8$, ist. Er hat die Länge 32. Dieser Bitstring wird gemäß der Permutation P aus Tabelle 6.3 permutiert. Das Ergebnis ist $f_K(R)$.

6.2.4 Die S-Boxen

Wir beschreiben nun die Funktionen S_i, $1 \leq i \leq 8$. Diese Funktionen heißen S-Boxen. Sie werden in Tabelle 6.4 dargestellt. Jede S-Box wird durch eine Tabelle mit vier Zeilen und 16 Spalten beschrieben. Für einen String $B = b_1 b_2 b_3 b_4 b_5 b_6$ wird der Funktionswert $S_i(B)$ folgendermaßen berechnet. Man interpretiert die natürliche Zahl mit Binärentwicklung $b_1 b_6$ als Zeilenindex und die natürliche Zahl mit Binärentwicklung $b_2 b_3 b_4 b_5$ als Spaltenindex. Den Eintrag in dieser Zeile und Spalte der S-Box stellt man binär dar und füllt diese Binärentwicklung vorne so mit Nullen auf, daß ihre Länge 4 wird. Das Ergebnis ist $S_i(B)$.

Beispiel 6.2.2. Wir berechnen $S_1(001011)$. Das erste Bit des Argumentes ist 0 und und das letzte Bit ist 1. Also ist der Zeilenindex die ganze Zahl mit Binärentwicklung 01, also 1. Die vier mittleren Bits des Argumentes sind 0101. Dies ist die Binärentwicklung von 5. Also ist der Spaltenindex 5. In der ersten S-Box steht in Zeile 1 und Spalte 5 die Zahl 2. Die Binärentwicklung von 2 ist 10. Also ist $S_1(001011) = 0010$.

6.2.5 Die Rundenschlüssel

Zuletzt muß noch erklärt werden, wie die Rundenschlüssel berechnet werden. Sei ein DES-Schlüssel $k \in \{0,1\}^{64}$ gegeben. Daraus werden Rundenschlüssel K_i, $1 \leq i \leq 16$, der Länge 48 generiert. Dazu definiert man v_i, $1 \leq i \leq 16$, folgendermaßen:

$$v_i = \begin{cases} 1 & \text{für } i \in \{1,2,9,16\} \\ 2 & \text{andernfalls.} \end{cases}$$

Zeile	Spalte															
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
S_1																
[0]	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
[1]	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
[2]	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
[3]	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13
S_2																
[0]	15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
[1]	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
[2]	0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
[3]	13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9
S_3																
[0]	10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
[1]	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
[2]	13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
[3]	1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12
S_4																
[0]	7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
[1]	13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
[2]	10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
[3]	3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14
S_5																
[0]	2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
[1]	14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
[2]	4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
[3]	11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3
S_6																
[0]	12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
[1]	10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
[2]	9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
[3]	4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13
S_7																
[0]	4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
[1]	13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
[2]	1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
[3]	6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12
S_8																
[0]	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
[1]	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
[2]	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
[3]	2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

Tabelle 6.4. S-Boxen des DES

Nun werden zwei Funktionen

$$\mathrm{PC1} : \{0,1\}^{64} \to \{0,1\}^{28} \times \{0,1\}^{28}, \quad \mathrm{PC2} : \{0,1\}^{28} \times \{0,1\}^{28} \to \{0,1\}^{48},$$

benutzt. Diese Funktionen werden unten beschrieben. Mit diesen Bausteinen erhält man die Schlüssel so:

1. Setze $(C_0, D_0) = \mathrm{PC1}(k)$.
2. Für $1 \leq i \leq 16$ berechne K_i folgendermaßen. Setze C_i auf den String, den man durch einen zirkulären Linksshift um v_i Stellen aus C_{i-1} gewinnt und D_i auf den String, den man durch einen zirkulären Linksshift um v_i Stellen aus D_{i-1} gewinnt. Berechne dann $K_i = \mathrm{PC2}(C_i, D_i)$.

Die Funktion PC1 bildet einen Bitstring k der Länge 64 auf zwei Bitstrings C und D der Länge 28 ab. Dies geschieht gemäß der Tabelle 6.5. Die obere Hälfte der Tabelle beschreibt, welche Bits aus K in C verwendet werden. Ist $k = k_1 k_2 \dots k_{64}$, dann ist $C = k_{57} k_{49} \dots k_{36}$. Die untere Hälfte dient der Konstruktion von D, also $D = k_{63} k_{55} \dots k_4$. Die Funktion PC2 bildet umgekehrt ein Paar (C, D) von Bitstrings der Länge 28 (also einen Bitstring der Länge 56) auf einen Bitstring der Länge 48 ab. Die Funktion wird in Tabelle 6.5 dargestellt. Der Wert $\mathrm{PC2}(b_1 \dots b_{56})$ ist $b_{14} b_{17} \dots b_{32}$.

PC1						
57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

PC2					
14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

Tabelle 6.5. Die Funktionen PC1 und PC2

Dies beendet die Beschreibung des DES-Algorithmus.

6.2.6 Entschlüsselung

Um einen Chiffretext zu entschlüsseln, wendet man DES mit der umgekehrten Schlüsselfolge auf ihn an.

6.3 Ein Beispiel für DES

Im folgenden illustrieren wir die Arbeit von DES an einem Beispiel.

Verschlüsselt wird das Wort $p = 0123456789ABCDEF$. Dessen Binärentwicklung ist

0	0	0	0	0	0	0	1
0	0	1	0	0	0	1	1
0	1	0	0	0	1	0	1
0	1	1	0	0	1	1	1
1	0	0	0	1	0	0	1
1	0	1	0	1	0	1	1
1	1	0	0	1	1	0	1
1	1	1	0	1	1	1	1

Die Anwendung von IP ergibt

1	1	0	0	1	1	0	0
0	0	0	0	0	0	0	0
1	1	0	0	1	1	0	0
1	1	1	1	1	1	1	1
1	1	1	1	0	0	0	0
1	0	1	0	1	0	1	0
1	1	1	1	0	0	0	0
1	0	1	0	1	0	1	0

In der ersten Zeile von IP(p) steht die umgekehrte zweite Spalte von p, in der zweiten Zeile von IP(p) steht die umgekehrte vierte Spalte von p usw. Damit ist

$$L_0 = 11001100000000000110011001111111,$$
$$R_0 = 11110000101010101111000010101010.$$

Wir verwenden den DES-Schlüssel aus Beispiel 6.2.1. Der ist

$$133457799BBCDFF1.$$

Seine Binärentwicklung ist

0	0	0	1	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	1	0	1	1	1
0	1	1	1	1	0	0	1
1	0	0	1	1	0	1	1
1	0	1	1	1	1	0	0
1	1	0	1	1	1	1	1
1	1	1	1	0	0	0	1

Daraus berechnen wir den ersten Rundenschlüssel. Es ist

$$C_0 = 1111000011001100101010101111, D_0 = 0101010101100110011110001111$$
$$C_1 = 1110000110011001010101011111, D_1 = 1010101011001100111100011110$$

und daher

$$K_1 = 000110110000000101110111111111100011100000011100 10.$$

Daraus gewinnt man

$$E(R_0) \oplus K_1 = 011000010001011110111010100001100110010100100111,$$

$$f(K_1, R_0) = 00100011010010101010100110111011$$

und schließlich

$$R_1 = 11101111010010100110010101000100.$$

Die anderen Runden werden analog berechnet.

6.4 Sicherheit des DES

Seit seiner Einführung ist der DES sorgfältig erforscht worden. Dabei wurden spezielle Verfahren entwickelt, mit denen man den DES angreifen kann. Die wichtigsten sind die differentielle und die lineare Kryptoanalyse. Beschreibungen dieser Angriffe und Referenzen finden sich in [52] und [78]. Bis jetzt ist aber der erfolgreichste Angriff die vollständige Durchsuchung des Schlüsselraums. Mit speziellen Computern und weltweiten Computernetzen ist es heute möglich, DES-verschlüsselte Dokumente in wenigen Tagen zu entschlüsseln. Es wird erwartet, daß in Kürze DES auf einem PC gebrochen werden kann, weil PCs immer schneller werden.

DES kann heute nur noch als sicher gelten, wenn die in Abschnitt 4.7 vorgestellte Dreifachverschlüsselung verwendet wird. Es ist dazu wichtig, festzustellen, daß die DES-Verschlüsselungsfunktionen nicht abgeschlossen unter Hintereinanderausführung sind. Sie bilden also keine Untergruppe der Permutationsgruppe $S_{64!}$. Würden die DES-Verschlüsselungsfunktionen eine Gruppe bilden, dann könnte man für zwei DES-Schlüssel k_1, k_2 einen dritten DES-Schlüssel k_3 finden, für den $\mathrm{DES}_{k_1} \circ \mathrm{DES}_{k_2} = \mathrm{DES}_{k_3}$ gelten würde. Mehrfachverschlüsselung würde also keinen Sicherheitsvorteil bieten. Es ist bekannt, daß die 2^{56} DES-Verschlüsselungsfunktionen eine Gruppe erzeugen, die wenigstens die Ordnung 10^{2499} hat (siehe [52]).

6.5 Übungen

Übung 6.5.1. Verifizieren Sie das Beispiel aus Abschnitt 6.3 und berechnen Sie die zweite Runde.

Übung 6.5.2. Berechnen Sie die dritte Verschlüsselungsrunde in Abschnitt 6.3.

Übung 6.5.3. Zeigen Sie, daß für $m, k \in \{0,1\}^{64}$ immer $\overline{\mathrm{DES}(m,k)} = \mathrm{DES}(\overline{m}, \overline{k})$ gilt.

Übung 6.5.4. Zeigen Sie, daß man C_{16} und D_{16} aus C_1 und D_1 durch einen zirkulären Rechtsshift um eine Position erhält.

Übung 6.5.5.
1. Zeigen Sie, daß C_{16} und D_{16} aus C_1 und D_1 durch einen zirkulären Rechtsshift der Länge 1 entstehen.
2. Gelte $K_1 = K_2 = \ldots = K_{16}$. Zeigen Sie, daß alle Bits in C_1 gleich sind und ebenso alle Bits in D_1 gleich sind.
3. Folgern Sie, daß es genau vier DES-Schlüssel gibt, für die alle Teilschlüssel gleich sind. Dies sind die *schwachen DES-Schlüssel*.
4. Geben Sie die vier schwachen DES-Schlüssel an.

Übung 6.5.6. Welche der im DES verwendeten Funktionen IP $E(R) \oplus K$, S_i, $1 \leq i \leq 8$, P, PC1, PC2 sind für festen Schlüssel K linear und welche nicht? Beweisen sie die Linearität oder geben Sie Gegenbeispiele an.

7. Der AES-Verschlüsslungsalgorithmus

1997 wurde vom National Institute of Standards and Technology (NIST) der Auswahlprozess für einen Nachfolger des DES begonnen. Unter den Einreichungen war auch die Rijndael-Chiffre. Sie ist benannt nach den beiden Erfindern Rijmen und Daemen. Dieses Verfahren wurde am 26. November 2001 als Advanced Encryption Standard (AES) standardisiert.

AES ist eine Blockchiffre mit Alphabet $\mathbb{Z}_2$. Sie ist ein Spezialfall der Rijndael-Chiffre. Die Rijndael-Chiffre läßt andere Blocklängen und andere Schlüsselräume als AES zu. Wir beschreiben hier die Rijndael-Chiffre und AES als Spezialfall.

7.1 Bezeichnungen

Um die Rijndael-Chiffre zu beschreiben, werden folgende Größen benötigt:

`Nb` Die Klartext- und Chiffretextblöcke bestehen aus `Nb` vielen 32-Bit Wörtern, $4 \leq \texttt{Nb} \leq 8$
Die Rijndael-Blocklänge ist also $32 * \texttt{Nb}$.
Für AES ist $\texttt{Nb} = 4$. Die AES-Blocklänge ist also 128.

`Nk` Die Schlüssel bestehen aus `Nk` vielen 32-Bit Wörtern, $4 \leq \texttt{Nk} \leq 8$
Der Rijndael-Schlüsselraum ist also $\mathbb{Z}_2^{32*\texttt{Nk}}$.
Für AES ist $\texttt{Nk} = 4$,6 oder 8.
Der AES-Schlüsselraum ist also $\mathbb{Z}_2^{128}$, $\mathbb{Z}_2^{192}$ oder $\mathbb{Z}_2^{256}$.

`Nr` Anzahl der Runden.

$$\text{Für AES ist } \texttt{Nr} = \begin{cases} 10 & \text{für } \texttt{Nk} = 4, \\ 12 & \text{für } \texttt{Nk} = 6, \\ 14 & \text{für } \texttt{Nk} = 8. \end{cases}$$

In den folgenden Beschreibungen werden die Datentypen `byte` und `word` benutzt. Ein `byte` ist ein Bitvektor der Länge 8. Ein `word` ist ein Bitvektor der Länge 32. Klartext und Chiffretext werden als zweidimensionale `byte`-Arrays dargestellt. Diese Arrays haben vier Zeilen und `Nb` Spalten. Im AES-Algorithmus sieht ein Klartext oder Chiffretext also so aus:

$$\begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{pmatrix} \tag{7.1}$$

Die Rijndael-Schlüssel sind `word`-Arrays der Länge `Nk`. Die Rijndael-Chiffre expandiert einen Schlüssel `key` mit der Funktion `KeyExpansion` zu dem expandierten Schlüssel `w`. Danach verschlüsselt sie einen Klartextblock `in` mit dem expandierten Schlüssel `w` zu dem Schlüsseltextblock `out`. Hierzu wird `Cipher` verwendet. In den folgenden Abschnitten beschreiben wir zuerst den Algorithmus `Cipher` und dann den Algorithmus `KeyExpansion`.

7.2 Cipher

Wir beschreiben die Funktion `Cipher`. Eingabe ist der Klartextblock `byte in[4,Nb]` und der expandierte Schlüssel `word w[Nb*(Nr+1)]`. Ausgabe ist der Chiffretextblock `byte out[4,Nb]`. Zuerst wird der Klartext `in` in das `byte`-Array `state` kopiert. Nach einer initialen Transformation durchläuft `state Nr` Runden und wird dann als Chiffretext zurückgegeben. In den ersten `Nr`-1 Runden werden nacheinander die Transformationen `SubBytes`, `ShiftRows`, `MixColumns` und `AddRoundKey` angewendet. In der letzten Runde werden nur noch die Transformationen`SubBytes`, `ShiftRows` und `AddRoundKey` angewendet. `AddRoundKey` ist auch die initiale Transformation.

```
Cipher(byte in[4,Nb], byte out[4,Nb], word w[Nb*(Nr+1)])
begin
   byte state[4,Nb]
   state = in
   AddRoundKey(state, w[0, Nb-1])
   for round = 1 step 1 to Nr-1
      SubBytes(state)
      ShiftRows(state)
      MixColumns(state)
      AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
   end for
   SubBytes(state)
   ShiftRows(state)
   AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])
   out = state
end
```

Abb. 7.1. Die AES-Funktion `Cipher`

In den folgenden Abschnitten werden die Transformationen im einzelnen beschrieben.

7.2.1 Identifikation der Bytes mit Elementen von GF(2^8)

Bytes spielen eine zentrale Rolle in der Rijndael-Chiffre. Sie können auch als ein Paar von Hexadezimalzahlen geschrieben werden.

Beispiel 7.2.1. Das Paar $\{2F\}$ von Hexadezimalzahlen entspricht dem Paar 0010 1111 von Bitvektoren der Länge vier, also dem Byte 00101111. Das Paar $\{A1\}$ von Hexadezimalzahlen entspricht dem Paar 1010 0001 von Bitvektoren, also dem Byte 10100001.

Bytes werden in der Rijndael-Chiffre mit Elementen des endlichen Körpers GF(2^8) identifiziert. Als erzeugendes Polynom (siehe Abschnitt 3.20) wird das über GF(2) irreduzible Polynom

$$m(X) = X^8 + X^4 + X^3 + X + 1 \tag{7.2}$$

gewählt. Damit ist

$$\mathrm{GF}(2^8) = \mathrm{GF}(2)(\alpha)$$

wobei α der Gleichung

$$\alpha^8 + \alpha^4 + \alpha^3 + \alpha + 1 = 0$$

genügt. Ein Byte

$$(b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0)$$

entspricht also dem Element

$$\sum_{i=0}^{7} b_i \alpha^i$$

Damit können Bytes multipliziert und addiert werden. Falls sie von Null verschieden sind, können sie auch invertiert werden. Für das Inverse von b wird b^{-1} geschrieben. Wir definieren auch $0^{-1} = 0$.

Beispiel 7.2.2. Das Byte $b = (0, 0, 0, 0, 0, 0, 1, 1)$ entspricht dem Körperelement $\alpha + 1$. Gemäß Beispiel 3.20.4 ist $(\alpha+1)^{-1} = \alpha^7 + \alpha^6 + \alpha^5 + \alpha^4 + \alpha^2 + \alpha$. Daher ist $b^{-1} = (1, 1, 1, 1, 0, 1, 1, 0)$.

7.2.2 `SubBytes`

`SubBytes(state)` ist eine nicht-lineare Funktion. Sie transformiert die einzelnen Bytes. Die Transformation wird S-Box genannt. Aus jedem Byte b von `state` macht die S-Box das neue Byte

$$b \leftarrow Ab^{-1} \oplus c \tag{7.3}$$

mit

$$A = \begin{pmatrix} 1&0&0&0&1&1&1&1 \\ 1&1&0&0&0&1&1&1 \\ 1&1&1&0&0&0&1&1 \\ 1&1&1&1&0&0&0&1 \\ 1&1&1&1&1&0&0&0 \\ 0&1&1&1&1&1&0&0 \\ 0&0&1&1&1&1&1&0 \\ 0&0&0&1&1&1&1&1 \end{pmatrix}, \quad c = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0. \end{pmatrix}$$

Diese S-Box kann tabelliert werden, weil sie nur 2^8 mögliche Argumente hat. Dann kann die Anwendung von `SubBytes` durch Table-Lookups realisiert werden.

Beispiel 7.2.3. Wir berechnen, welches Byte die S-Box aus dem Vektor $b = (0,0,0,0,0,0,1,1)$ macht. Nach Beispiel 7.2.2 ist $b^{-1} = (1,1,1,1,0,1,1,0)$. Damit gilt $Ab^{-1} + c = (0,1,1,0,0,1,1,1)$.

Die S-Box garantiert die Nicht-Linearität von AES.

7.2.3 `ShiftRows`

Sei s ein `state`, also ein durch AES teiltransformierter Klartext. Schreibe s als Matrix. Die Einträge sind Bytes. Die Matrix hat 4 Zeilen und `Nb` Spalten. Im Fall von AES ist diese Matrix

$$\begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{pmatrix} \tag{7.4}$$

`ShiftRows` wendet auf die Zeilen dieser Matrix einen zyklischen Linksshift an. Genauer: `ShiftRows` hat folgende Wirkung:

$$\begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{pmatrix} \leftarrow \begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,1} & s_{1,2} & s_{1,3} & s_{1,0} \\ s_{2,2} & s_{2,3} & s_{2,0} & s_{2,1} \\ s_{3,3} & s_{3,0} & s_{3,1} & s_{3,2} \end{pmatrix} \tag{7.5}$$

Im allgemeinen wird die i-te Zeile um c_i Positionen nach links verschoben, wobei c_i in Tabelle 7.1 zu finden ist. Diese Transformation sorgt bei Anwendung in mehreren Runden für hohe Diffusion.

Nb	c_0	c_1	c_2	c_3
4	0	1	2	3
5	0	1	2	3
6	0	1	2	3
7	0	1	2	4
8	0	1	3	4

Tabelle 7.1. Zyklischer Linksshift in `ShiftRows`

7.2.4 `MixColumns`

Für $0 \leq j < \texttt{Nb}$ wird die Spalte

$$s_j = (s_{0,j}, s_{1,j}, s_{2,j}, s_{3,j})$$

von `state` mit dem Polynom

$$s_{0,j} + s_{1,j}x + s_{2,j}x^2 + s_{3,j}x^3 \in \mathrm{GF}(2^8)[x] \tag{7.6}$$

identifiziert. Die Transformation `MixColumns` setzt

$$s_j \leftarrow (s_j * a(x)) \bmod (x^4 + 1), \quad 0 \leq j < \texttt{Nb}, \tag{7.7}$$

wobei

$$a(x) = \{03\} * x^3 + \{01\} * x^2 + \{01\} * x + \{02\}. \tag{7.8}$$

Das kann auch als lineare Transformation in $\mathrm{GF}(2^8)^4$ beschrieben werden. `MixColumns` setzt nämlich

$$s_j \leftarrow \begin{pmatrix} \{02\} & \{03\} & \{01\} & \{01\} \\ \{01\} & \{02\} & \{03\} & \{01\} \\ \{01\} & \{01\} & \{02\} & \{03\} \\ \{03\} & \{01\} & \{01\} & \{02\} \end{pmatrix} s_j \quad 0 \leq j < \texttt{Nb}. \tag{7.9}$$

Diese Transformation sorgt für eine Diffusion innerhalb der Spalten von `state`.

7.2.5 `AddRoundKey`

Sind $s_0, \ldots, s_{\texttt{Nb}-1}$ die Spalten von `state`, dann setzt der Aufruf der Funktion `AddRoundKey(state, w[l*Nb, (l+1)*Nb-1])`

$$s_j \leftarrow s_j \oplus w[l * \texttt{Nb} + j], \quad 0 \leq j < \texttt{Nb}, \tag{7.10}$$

wobei $\oplus$ das bitweise $\oplus$ ist. Die Wörter des Rundenschlüssels werden also mod 2 zu den Spalten von `state` addiert. Dies ist eine sehr einfache und effiziente Transformation, die die Transformation einer Runde schlüsselabhängig macht.

7.3 KeyExpansion

Der Algorithmus `KeyExpansion` macht aus dem dem Rijndael-Schlüssel `key`, der ein `byte`-array der Länge `4*Nk` ist, einen expandierten Schlüssel `w`, der ein `word`-array der Länge `Nb*(Nr+1)` ist. Die Verwendung des expandierten Schlüssels wurde in Abschnitt 7.2 erklärt. Zuerst werden die ersten `Nk` Wörter im expandierten Schlüssel `w` mit den Bytes des Schlüssels `key` gefüllt. Die folgenden Wörter in `word` werden erzeugt, wie es im Pseudocode von `KeyExpansion` beschrieben ist. Die Funktion `word` schreibt die Bytes einfach hintereinander.

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
   word temp
   i = 0
   while (i < Nk)
      w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
      i = i+1
   end while
   i = Nk
   while (i < Nb * (Nr+1)]
      temp = w[i-1]
      if (i mod Nk = 0)
         temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
      else if (Nk > 6 and i mod Nk = 4)
         temp = SubWord(temp)
      end if
      w[i] = w[i-Nk] xor temp
      i = i + 1
   end while
end
```

Abb. 7.2. Die AES-Funktion `KeyExpansion`

Wir beschreiben nun die einzelnen Prozeduren.

`SubWord` bekommt als Eingabe ein Wort. Dieses Wort kann als Folge (b_0, b_1, b_2, b_3) von Bytes dargestellt werden. Auf jedes dieser Bytes wird die Funktion `SubBytes` angewendet. Jedes dieser Bytes wird gemäß (7.3) transformiert. Die Folge

$$(b_0, b_1, b_2, b_3) \leftarrow (Ab_0^{-1} + c, Ab_1^{-1} + c, Ab_2^{-1} + c, Ab_3^{-1} + c) \tag{7.11}$$

der transformierten Bytes wird zurückgegeben.

Die Funktion `RotWord` erhält als Eingabe ebenfalls ein Wort (b_0, b_1, b_2, b_3). Die Ausgabe ist

$$(b_0, b_1, b_2, b_3) \leftarrow (b_1, b_2, b_3, b_0). \tag{7.12}$$

Außerdem ist

$$\texttt{Rcon[n]} = (\{02\}^{\texttt{n}}, \{00\}, \{00\}, \{00\}). \tag{7.13}$$

7.4 Ein Beispiel

Wir präsentieren ein Beispiel für die Anwendung der AES-Chiffre. Das Beispiel stammt von Brian Gladman `brg@gladman.uk.net`.

Die Bezeichnungen haben folgende Bedeutung:

`input`	Klartext
`k_sch`	Rundenschlüssel für Runde `r`
`start`	state zu Beginn von Runde `r`
`s_box`	state nach Anwendung der S-Box `SubBytes`
`s_row`	state nach Anwendung von `ShiftRows`
`m_col`	state nach Anwendung von `MixColumns`
`output`	Schlüsseltext

```
PLAINTEXT:     3243f6a8885a308d313198a2e0370734
KEY:           2b7e151628aed2a6abf7158809cf4f3c
ENCRYPT        16 byte block, 16 byte key
R[00].input    3243f6a8885a308d313198a2e0370734
R[00].k_sch    2b7e151628aed2a6abf7158809cf4f3c
R[01].start    193de3bea0f4e22b9ac68d2ae9f84808
R[01].s_box    d42711aee0bf98f1b8b45de51e415230
R[01].s_row    d4bf5d30e0b452aeb84111f11e2798e5
R[01].m_col    046681e5e0cb199a48f8d37a2806264c
R[01].k_sch    a0fafe1788542cb123a339392a6c7605
R[02].start    a49c7ff2689f352b6b5bea43026a5049
R[02].s_box    49ded28945db96f17f39871a7702533b
R[02].s_row    49db873b453953897f02d2f177de961a
R[02].m_col    584dcaf11b4b5aacdbe7caa81b6bb0e5
R[02].k_sch    f2c295f27a96b9435935807a7359f67f
R[03].start    aa8f5f0361dde3ef82d24ad26832469a
R[03].s_box    ac73cf7befc111df13b5d6b545235ab8
R[03].s_row    acc1d6b8efb55a7b1323cfdf457311b5
R[03].m_col    75ec0993200b633353c0cf7cbb25d0dc
R[03].k_sch    3d80477d4716fe3e1e237e446d7a883b
R[04].start    486c4eee671d9d0d4de3b138d65f58e7
R[04].s_box    52502f2885a45ed7e311c807f6cf6a94
R[04].s_row    52a4c89485116a28e3cf2fd7f6505e07
R[04].m_col    0fd6daa9603138bf6fc0106b5eb31301
R[04].k_sch    ef44a541a8525b7fb671253bdb0bad00
```

```
R[05].start   e0927fe8c86363c0d9b1355085b8be01
R[05].s_box   e14fd29be8fbfbba35c89653976cae7c
R[05].s_row   e1fb967ce8c8ae9b356cd2ba974ffb53
R[05].m_col   25d1a9adbd11d168b63a338e4c4cc0b0
R[05].k_sch   d4d1c6f87c839d87caf2b8bc11f915bc
R[06].start   f1006f55c1924cef7cc88b325db5d50c
R[06].s_box   a163a8fc784f29df10e83d234cd503fe
R[06].s_row   a14f3dfe78e803fc10d5a8df4c632923
R[06].m_col   4b868d6d2c4a8980339df4e837d218d8
R[06].k_sch   6d88a37a110b3efddbf98641ca0093fd
R[07].start   260e2e173d41b77de86472a9fdd28b25
R[07].s_box   f7ab31f02783a9ff9b4340d354b53d3f
R[07].s_row   f783403f27433df09bb531ff54aba9d3
R[07].m_col   1415b5bf461615ec274656d7342ad843
R[07].k_sch   4e54f70e5f5fc9f384a64fb24ea6dc4f
R[08].start   5a4142b11949dc1fa3e019657a8c040c
R[08].s_box   be832cc8d43b86c00ae1d44dda64f2fe
R[08].s_row   be3bd4fed4e1f2c80a642cc0da83864d
R[08].m_col   00512fd1b1c889ff54766dcdfa1b99ea
R[08].k_sch   ead27321b58dbad2312bf5607f8d292f
R[09].start   ea835cf00445332d655d98ad8596b0c5
R[09].s_box   87ec4a8cf26ec3d84d4c46959790e7a6
R[09].s_row   876e46a6f24ce78c4d904ad897ecc395
R[09].m_col   473794ed40d4e4a5a3703aa64c9f42bc
R[09].k_sch   ac7766f319fadc2128d12941575c006e
R[10].start   eb40f21e592e38848ba113e71bc342d2
R[10].s_box   e9098972cb31075f3d327d94af2e2cb5
R[10].s_row   e9317db5cb322c723d2e895faf090794
R[10].k_sch   d014f9a8c9ee2589e13f0cc8b6630ca6
R[10].output  3925841d02dc09fbdc118597196a0b32
```

7.5 `InvCipher`

Die Entschlüsselung der Rijndael-Chiffre wird von der Funktion `InvCipher` besorgt. Die Spezifikation der Funktionen `InvShiftRows` und `InvSubBytes` ergibt sich aus der Spezifikation von `ShiftRows` und `SubBytes`.

7.6 Übungen

Übung 7.6.1. Stellen Sie die AES-S-Box wie in Tabelle 17.1 dar.

Übung 7.6.2. Beschreiben Sie die Funktionen `InvShiftRows`, `InvSubBytes` und `InvMixColumns`.

```
InvCipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
   byte state[4,Nb]
   state = in
   AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])
   for round = Nr-1 step -1 downto 1
      InvShiftRows(state)
      InvSubBytes(state)
      AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
      InvMixColumns(state)
   end for
   InvShiftRows(state)
   InvSubBytes(state)
   AddRoundKey(state, w[0, Nb-1])
   out = state
end
```

Abb. 7.3. Die AES-Funktion `InvCipher`

Übung 7.6.3. Entschlüsseln Sie den Schlüsseltext aus Abschnitt 7.4 mit `InvCipher`.

8. Primzahlerzeugung

Für Public-Key-Kryptosysteme braucht man häufig zufällige große Primzahlen. Dazu erzeugt man natürliche Zahlen der richtigen Größe und prüft, ob sie Primzahlen sind. In diesem Kapitel zeigen wir, wie man effizient entscheiden kann, ob eine natürliche Zahl eine Primzahl ist. Wir diskutieren außerdem, wie man die natürlichen Zahlen erzeugen muß, um annähernd eine Gleichverteilung auf den Primzahlen der gewünschten Größe zu erhalten. Die beschriebenen Algorithmen sind in der Bibliothek *LiDIA*[50] implementiert.

Nach Fertigstellung der dritten Auflage dieses Buches wurde von M. Agraval, N. Kayal und N. Saxena [2] ein deterministischer Polynomzeitalgorithmus gefunden, der entscheidet, ob eine vorgelegte natürliche Zahl eine Primzahl ist. Dieser Algorithmus ist aber für die Praxis noch zu ineffizient.

Mit kleinen lateinischen Buchstaben werden ganze Zahlen bezeichnet.

8.1 Probedivision

Sei n eine natürliche Zahl. Wir möchten gerne wissen, ob n eine Primzahl ist oder nicht. Eine einfache Methode, das festzustellen, beruht auf folgendem Satz.

Theorem 8.1.1. *Wenn n eine zusammengesetzte natürliche Zahl ist, dann hat n einen Primteiler p, der nicht größer ist als $\sqrt{n}$.*

Beweis. Da n zusammengesetzt ist, kann man $n = ab$ schreiben mit $a > 1$ und $b > 1$. Es gilt $a \leq \sqrt{n}$ oder $b \leq \sqrt{n}$. Andernfalls wäre $n = ab > \sqrt{n}\sqrt{n} = n$. Nach Theorem 2.11.2 haben a und b Primteiler. Diese Primteiler teilen auch n und daraus folgt die Behauptung. □

Um festzustellen, ob n eine ungerade Primzahl ist, braucht man also nur für alle Primzahlen p, die nicht größer als $\sqrt{n}$ sind, zu testen, ob sie n teilen. Dazu muß man diese Primzahlen bestimmen oder in einer Tabelle nachsehen. Diese Tabelle kann man mit dem Sieb des Eratosthenes berechnen (siehe [4]). Man kann aber auch testen, ob n durch eine ungerade Zahl teilbar ist, die nicht größer als $\sqrt{n}$ ist. Wenn ja, dann ist n keine Primzahl. Andernfalls ist n eine Primzahl. Diese Verfahren bezeichnet man als *Probedivision*.

Beispiel 8.1.2. Wir wollen mit Probedivision feststellen, ob $n = 15413$ eine Primzahl ist. Es gilt $\lfloor\sqrt{n}\rfloor = 124$. Also müssen wir testen, ob eine der Primzahlen $p \leq 124$ ein Teiler von n ist. Die ungeraden Primzahlen $p \leq 124$ sind 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113. Keine dieser Primzahlen teilt n. Daher ist n selbst eine Primzahl.

Probedivision kann man auch verwenden, um die Primfaktorzerlegung einer natürlichen Zahl zu finden. Man hört dann nicht auf, wenn ein Primteiler gefunden ist, sondern man sucht den nächsten Primteiler, den übernächsten usw., bis man fertig ist.

Beispiel 8.1.3. Mit Probedivision wollen wir die Zahl 476 faktorisieren. Der erste Primteiler, den wir finden, ist 2 und $476/2 = 238$. Der nächste Primteiler ist wieder 2 und $238/2 = 119$. Der nächste Primteiler ist 7 und $119/7 = 17$. Die Zahl 17 ist eine Primzahl. Also ist $476 = 2^2 * 7 * 17$ die Primfaktorzerlegung von 476.

In Faktorisierungsalgorithmen verwendet man Probedivision mit Primzahlen bis 10^6, um die kleinen Primteiler zu finden.

Um den Aufwand der Probedivision mit Primzahlen zu bestimmen, geben wir eine Abschätzung für die Anzahl der Primzahlen unterhalb einer Schranke an. Man benutzt folgende Bezeichnung:

Definition 8.1.4. *Ist x eine positive reelle Zahl, so bezeichnet $\pi(x)$ die Anzahl der Primzahlen, die nicht größer als x sind.*

Beispiel 8.1.5. Es ist $\pi(1) = 0$, $\pi(4) = 2$. Wie wir in Beispiel 8.1.2 gesehen haben, ist $\pi(124) = 30$.

Folgenden Satz erwähnen wir ohne Beweis (siehe [66]). Darin bezeichnet log den natürlichen Logarithmus.

Theorem 8.1.6. *1. Für $x \geq 17$ gilt $\pi(x) > x/\log x$.*
2. Für $x > 1$ gilt $\pi(x) < 1.25506(x/\log x)$.

Aus Theorem 8.1.6 folgt, daß man wenigstens $\sqrt{n}/\log\sqrt{n}$ Probedivisionen braucht, um zu beweisen, daß eine natürliche Zahl n eine Primzahl ist. Im RSA-Verfahren benutzt man Primzahlen, die größer als 10^{75} sind. Um die Primalität einer solchen Zahl zu beweisen, müßte man mehr als $10^{75/2}/\log 10^{75/2} > .36 * 10^{36}$ Probedivisionen machen. Das ist nicht durchführbar. In den nächsten Abschnitten geben wir effizientere Verfahren an, die die Primalität einer Zahl beweisen.

8.2 Der Fermat-Test

Verfahren, die beweisen, daß eine Zahl n eine Primzahl ist, sind aufwendig. Es gibt aber eine Reihe von Verfahren, die feststellen können, daß eine natürliche Zahl mit hoher Wahrscheinlichkeit eine Primzahl ist. Solche Verfahren heißen Primzahltests. Der Fermat-Test ist ein solcher Primzahltest. Er beruht auf dem kleinen Satz von Fermat (siehe Theorem 3.11.1). Dieser Satz wird in folgender Version benötigt.

Theorem 8.2.1 (Kleiner Satz von Fermat). *Ist n eine Primzahl, so gilt $a^{n-1} \equiv 1 \bmod n$ für alle $a \in \mathbb{Z}$ mit $\gcd(a, n) = 1$.*

Dieses Theorem eröffnet die Möglichkeit festzustellen, daß eine natürliche Zahl n zusammengesetzt ist. Man wählt eine natürliche Zahl $a \in \{1, 2, \ldots, n-1\}$. Man berechnet unter Verwendung der schnellen Exponentiation aus Abschnitt 3.12 den Wert $y = a^{n-1} \bmod n$. Ist $y \neq 1$, so ist n nach Theorem 8.2.1 keine Primzahl, also zusammengesetzt. Ist dagegen $y = 1$, so kann n sowohl eine Primzahl als auch zusammengesetzt sein, wie das folgende Beispiel zeigt.

Beispiel 8.2.2. Betrachte die Zahl $n = 341 = 11 * 31$. Es gilt

$$2^{340} \equiv 1 \bmod 341$$

obwohl n zusammengesetzt ist. Dagegen ist

$$3^{340} \equiv 56 \bmod 341.$$

Nach dem kleinen Satz von Fermat ist 341 also zusammengesetzt.

Wenn der Fermat-Test bewiesen hat, daß n zusammengesetzt ist, dann hat er damit noch keinen Teiler von n gefunden. Er hat nur gezeigt, daß n eine Eigenschaft fehlt, die alle Primzahlen haben. Daher kann der Fermat-Test auch nicht als Faktorisierungsalgorithmus verwendet werden.

Der Fermat-Test ist in der *LiDIA*-Methode `bool fermat(const bigint & a)` in der Klasse `bigint` implementiert.

8.3 Carmichael-Zahlen

Der Fermat-Test kann also zeigen, daß eine Zahl n zusammengesetzt ist. Er kann aber nicht beweisen, daß n eine Primzahl ist. Wenn der Fermat-Test aber für viele Basen a keinen Beweis gefunden hat, daß n zusammengesetzt ist, scheint es wahrscheinlich zu sein, daß n eine Primzahl ist. Wir werden nun zeigen, daß es natürliche Zahlen gibt, deren Zusammengesetztheit der Fermat-Test nicht feststellen kann.

Wir brauchen zwei Begriffe. Ist n eine ungerade zusammengesetzte Zahl und gilt für eine ganze Zahl a die Kongruenz

$$a^{n-1} \equiv 1 \bmod n,$$

so heißt n *Pseudoprimzahl* zur Basis a. Ist n eine Pseudoprimzahl zur Basis a für alle ganzen Zahlen a mit $\gcd(a, n) = 1$, dann heißt n *Carmichael-Zahl*. Die kleinste Carmichael-Zahl ist $561 = 3 \cdot 11 \cdot 17$. Man kann beweisen, daß es unendlich viele Carmichael-Zahlen gibt. Weil es Pseudoprimzahlen und Carmichael-Zahlen gibt, ist der Fermat-Test für die Praxis nicht besonders geeignet. Besser geeignet ist der Miller-Rabin-Test, den wir unten beschreiben. Um dessen Gültigkeit zu beweisen, brauchen wir aber noch eine Charakterisierung von Carmichael-Zahlen.

Theorem 8.3.1. *Eine ungerade zusammensetzte Zahl $n \geq 3$ ist genau dann eine Carmichael-Zahl, wenn n quadratfrei ist, also keinen mehrfachen Primfaktor hat, und wenn für jeden Primteiler p von n die Zahl $p - 1$ ein Teiler von $n - 1$ ist.*

Beweis. Sei $n \geq 3$ eine Carmichael-Zahl. Dann gilt

$$a^{n-1} \equiv 1 \bmod n \tag{8.1}$$

für jede ganze Zahl a, die zu n teilerfremd ist. Sei p ein Primteiler von n und sei a eine Primitivwurzel mod p, die zu n teilerfremd ist. Eine solche Primitivwurzel kann man nach dem Chinesischen Restsatz konstruieren. Dann folgt aus (8.1)

$$a^{n-1} \equiv 1 \bmod p$$

Nach Theorem 3.9.2 muß die Ordnung $p - 1$ von a ein Teiler von $n - 1$ sein. Wir müssen noch zeigen, daß p^2 kein Teiler von n ist. Dazu benutzt man ein ähnliches Argument. Angenommen, p^2 teilt n. Dann ist $(p - 1)p$ ein Teiler von $\varphi(n)$ und man kann sogar zeigen, daß es in der primen Restklassengruppe mod n ein Element der Ordnung p gibt. Daraus folgt wie oben, daß p ein Teiler von $n - 1$ ist. Das geht aber nicht, weil p ein Teiler von n ist.

Sei umgekehrt n quadratfrei und sei $p - 1$ ein Teiler von $n - 1$ für alle Primteiler p von n. Sei a eine zu n teilerfremde natürliche Zahl. Dann gilt

$$a^{p-1} \equiv 1 \bmod p$$

nach dem kleinen Satz von Fermat und daher

$$a^{n-1} \equiv 1 \bmod p$$

weil $n - 1$ ein Vielfaches von $p - 1$ ist. Dies impliziert

$$a^{n-1} \equiv 1 \bmod n$$

weil die Primteiler von n paarweise verschieden sind. □

8.4 Der Miller-Rabin-Test

Im Gegensatz zum Fermat-Test findet der Miller-Rabin-Test nach hinreichend vielen Versuchen für jede natürliche Zahl heraus, ob sie zusammengesetzt ist oder nicht.

Der Miller-Rabin-Test verwendet eine Verschärfung des kleinen Satzes von Fermat. Die Situation ist folgende. Es sei n eine ungerade natürliche Zahl und es sei

$$s = \max\{r \in \mathbb{N} : 2^r \text{ teilt } n-1\}.$$

Damit ist also 2^s die größte Potenz von 2, die $n-1$ teilt. Setze

$$d = (n-1)/2^s.$$

Dann ist d eine ungerade Zahl. Folgendes Resultat ist für den Miller-Rabin-Test fundamental.

Theorem 8.4.1. *Ist n eine Primzahl und ist a eine zu n teilerfremde ganze Zahl, so gilt mit den Bezeichnungen von oben entweder*

$$a^d \equiv 1 \bmod n \tag{8.2}$$

oder es gibt ein r in der Menge $\{0, 1, \ldots, s-1\}$ mit

$$a^{2^r d} \equiv -1 \bmod n. \tag{8.3}$$

Beweis. Sei a eine ganze Zahl, die zu n teilerfremd ist. Die Ordnung der primen Restklassengruppe mod n ist $n-1 = 2^s d$, weil n eine Primzahl ist. Nach Theorem 3.9.5 ist die Ordnung k der Restklasse $a^d + n\mathbb{Z}$ eine Potenz von 2. Ist diese Ordnung $k = 1 = 2^0$, dann gilt

$$a^d \equiv 1 \bmod n.$$

Ist $k > 1$, dann ist $k = 2^l$ mit $1 \leq l \leq s$. Nach Theorem 3.9.5 hat die Restklasse $a^{2^{l-1}d} + n\mathbb{Z}$ die Ordnung 2. Nach Übung 3.23.20 ist das einzige Element der Ordnung 2 aber $-1 + n\mathbb{Z}$. Also gilt für $r = l - 1$

$$a^{2^r d} \equiv -1 \bmod n.$$

Man beachte, daß $0 \leq r < s$ gilt. □

Wenigstens eine der Bedingungen aus Theorem 8.4.1 ist notwendig dafür, daß n eine Primzahl ist. Findet man also eine ganze Zahl a, die zu n teilerfremd ist und für die weder (8.2) noch (8.3) für ein $r \in \{0, \ldots, s-1\}$ gilt, so ist bewiesen, daß n keine Primzahl, sondern zusammengesetzt ist. Eine solche Zahl a heißt *Zeuge* gegen die Primalität von n.

Beispiel 8.4.2. Sei $n = 561$. Mit Hilfe des Fermat-Tests kann nicht festgestellt werden, daß n zusammengesetzt ist, weil n eine Carmichael-Zahl ist. Aber $a = 2$ ist ein Zeuge gegen die Primalität von n, wie wir jetzt zeigen. Es ist $s = 4$, $d = 35$ und $2^{35} \equiv 263 \bmod 561$, $2^{2*35} \equiv 166 \bmod 561$, $2^{4*35} \equiv 67 \bmod 561$, $2^{8*35} \equiv 1 \bmod 561$. Also ist 561 nach Theorem 8.4.1 keine Primzahl.

Im nächsten Satz wird die Anzahl der Zeugen gegen die Primalität einer zusammengesetzten Zahl abgeschätzt.

Theorem 8.4.3. *Ist $n \geq 3$ eine ungerade zusammengesetzte Zahl, so gibt es in der Menge $\{1, \ldots, n-1\}$ höchstens $(n-1)/4$ Zahlen, die zu n teilerfremd und keine Zeugen gegen die Primalität von n sind.*

Beweis. Sei $n \geq 3$ eine ungerade zusammengesetzte natürliche Zahl.

Wir wollen die Anzahl der $a \in \{1, 2, \ldots, n-1\}$ abschätzen, für die $\gcd(a, n) = 1$ gilt und zusätzlich

$$a^d \equiv 1 \bmod n \tag{8.4}$$

oder

$$a^{2^r d} \equiv -1 \mod n \tag{8.5}$$

für ein $r \in \{0, 1, \ldots, s-1\}$. Wenn es kein solches a gibt, sind wir fertig. Angenommen es gibt einen solchen Nicht-Zeugen a. Dann gibt es auch einen, für den (8.5) gilt. Erfüllt a nämlich (8.4), dann erfüllt $-a$ die Bedingung (8.5). Sei k der größte Wert von r, für den es ein a mit $\gcd(a, n) = 1$ und (8.5) gibt. Wir setzen

$$m = 2^k d.$$

Die Primfaktorzerlegung von n sei

$$n = \prod_{p|n} p^{e(p)}.$$

Wir definieren die folgenden Untergruppen von $(\mathbb{Z}/n\mathbb{Z})^*$:

$$J = \{a + n\mathbb{Z} : \gcd(a, n) = 1, a^{n-1} \equiv 1 \bmod n\},$$

$$K = \{a + n\mathbb{Z} : \gcd(a, n) = 1, a^m \equiv \pm 1 \bmod p^{e(p)} \text{ für alle } p|n\},$$

$$L = \{a + n\mathbb{Z} : \gcd(a, n) = 1, a^m \equiv \pm 1 \bmod n\},$$

$$M = \{a + n\mathbb{Z} : \gcd(a, n) = 1, a^m \equiv 1 \bmod n\}.$$

Dann gilt

$$M \subset L \subset K \subset J \subset (\mathbb{Z}/n\mathbb{Z})^*.$$

Für jedes zu n teilerfremde a, das kein Zeuge gegen die Primalität von n ist, gehört die Restklasse $a + n\mathbb{Z}$ zu L. Wir werden die Behauptung des Satzes

beweisen, indem wir zeigen, daß der Index von L in $(\mathbb{Z}/n\mathbb{Z})^*$ wenigstens vier ist.

Der Index von M in K ist eine Potenz von 2, weil das Quadrat jedes Elementes von K in M liegt. Der Index von L in K ist daher auch eine Potenz von 2, etwa 2^j. Ist $j \geq 2$, sind wir fertig.

Ist $j = 1$, so hat n zwei Primteiler. Nach Übung 8.6.5 ist n keine Carmichael-Zahl und daher ist J eine echte Untergruppe von $(\mathbb{Z}/n\mathbb{Z})^*$, der Index von J in $(\mathbb{Z}/n\mathbb{Z})^*$ wenigstens 2. Weil der Index von L in K nach Definition von m ebenfalls 2 ist, ist der Index von L in $(\mathbb{Z}/n\mathbb{Z})^*$ wenigstens 4.

Sei schließlich $j = 0$. Dann ist n eine Primzahlpotenz. Man kann für diesen Fall verifizieren, daß J genau $p-1$ Elemente hat, nämlich genau die Elemente der Untergruppe der Ordnung $p-1$ der zyklischen Gruppe $(\mathbb{Z}/p^e\mathbb{Z})^*$. Daher ist der Index von J in $(\mathbb{Z}/n\mathbb{Z})^*$ wenigstens 4, es sei denn, $n = 9$. Für $n = 9$ kann man die Behauptung aber direkt verifizieren. □

Beispiel 8.4.4. Wir bestimmen alle Zeugen gegen die Primalität von $n = 15$. Es ist $n - 1 = 14 = 2 * 7$. Daher ist $s = 1$ und $d = 7$. Eine zu 15 teilerfremde Zahl a ist genau dann ein Zeuge gegen die Primalität von n, wenn $a^7 \bmod 15 \neq 1$ und $a^7 \bmod 15 \neq -1$. Folgende Tabelle enthält die entsprechenden Reste:

a	1	2	4	7	8	11	13	14
$a^{14} \bmod 15$	1	4	1	4	4	1	4	1
$a^7 \bmod 15$	1	8	4	13	2	11	7	14

Die Anzahl der zu 15 teilerfremden Zahlen in $\{1, 2, \ldots, 14\}$, die keine Zeugen gegen die Primalität von n sind, ist $2 \leq (15-1)/4 = 7/2$.

Wenn man den Miller-Rabin-Test auf die ungerade Zahl n anwenden will, wählt man zufällig und gleichverteilt eine Zahl $a \in \{2, 3, \ldots, n-1\}$. Ist $\gcd(a, n) > 1$, dann ist n zusammengesetzt. Andernfalls berechnet man $a^d, a^{2d}, \ldots, a^{2^{s-1}d}$. Findet man dabei einen Zeugen gegen die Primalität von n, dann ist bewiesen, daß n zusammengesetzt ist. Nach Theorem 8.4.3 ist die Wahrscheinlichkeit dafür, daß man keinen Zeugen findet und daß n zusammengesetzt ist, höchstens $1/4$. Wiederholt man den Miller-Rabin-Test t-mal und ist n zusammengesetzt, so ist die Wahrscheinlichkeit dafür, daß kein Zeuge gegen die Primalität gefunden wird, höchstens $(1/4)^t$. Für $t = 10$ ist dies $1/2^{20} \approx 1/10^6$. Dies ist sehr unwahrscheinlich. Genauere Analysen des Verfahrens haben gezeigt, daß die Fehlerwahrscheinlichkeit in Wirklichkeit noch kleiner ist.

Der Miller-Rabin-Test mit n Iterationen ist in der *LiDIA*-Methode `bool is_prime(const bigint & a, int n)` in der Klasse `bigint` implementiert.

8.5 Zufällige Wahl von Primzahlen

In vielen Public-Key-Verfahren müssen bei der Schlüsselerzeugung zufällige Primzahlen mit fester Bitlänge erzeugt werden. Wir beschreiben ein Verfahren zur Konstruktion solcher Primzahlen.

Wir wollen eine zufällige Primzahl erzeugen, deren Bitlänge k ist. Dazu erzeugen wir zuerst eine zufällige ungerade k-Bit-Zahl n (siehe Abschnitt 5.6). Wir setzen das erste und letzte Bit von n auf 1 und die restlichen $k-2$ Bits wählen wir unabhängig und zufällig gemäß der Gleichverteilung. Danach überprüfen wir, ob n eine Primzahl ist. Zuerst prüfen wir, ob n durch eine Primzahl unterhalb einer Schranke B teilbar ist. Diese Primzahlen stehen in einer Tabelle. Typischerweise ist $B = 10^6$. Wurde kein Teiler von n gefunden, so wird der Miller-Rabin-Test auf n (mit t Wiederholungen) angewendet. Wenn dabei kein Zeuge gegen die Primalität von n gefunden wird, so gilt n als Primzahl. Andernfalls ist bewiesen, daß n zusammengesetzt ist, und der Test muß mit einer neuen Zufallszahl gemacht werden. Die Wiederholungszahl t wird so gewählt, daß die Fehlerwahrscheinlichkeit des Miller-Rabin-Tests hinreichend klein ist. Bei der Suche nach einer Primzahl mit mehr als 1000 Bits reicht es für eine Fehlerwahrscheinlichkeit von weniger als $(1/2)^{80}$ aus, $t = 3$ zu wählen. Die Auswahl der Primzahlschranke B hängt davon ab, wie schnell bei der verwendeten Hard- und Software eine Probedivision im Verhältnis zu einem Miller-Rabin-Test ausgeführt werden kann.

8.6 Übungen

Übung 8.6.1. Beweisen Sie mit dem Fermat-Test, daß 1111 keine Primzahl ist.

Übung 8.6.2. Bestimmen Sie $\pi(100)$. Vergleichen Sie ihr Resultat mit den Schranken aus Theorem 8.1.6.

Übung 8.6.3. Bestimmen Sie die kleinste Pseudoprimzahl zur Basis 2.

Übung 8.6.4. Beweisen Sie mit dem Fermat-Test, daß die fünfte Fermat-Zahl $F_5 = 2^{2^5} + 1$ zusammengesetzt ist. Beweisen Sie, daß jede Fermat-Zahl eine Pseudoprimzahl zur Basis 2 ist.

Übung 8.6.5. Zeigen Sie, daß eine Carmichael-Zahl wenigstens drei verschiedene Primteiler hat.

Übung 8.6.6. Verwenden Sie den Miller-Rabin-Test, um zu beweisen, daß die fünfte Fermat-Zahl $F_5 = 2^{2^5} + 1$ zusammengesetzt ist. Vergleichen Sie die Effizienz dieser Berechnung mit der Berechnung in Übung 8.6.4.

Übung 8.6.7. Beweisen Sie mit dem Miller-Rabin-Test, daß die Pseudoprimzahl n aus Übung 8.6.3 zusammengesetzt ist. Bestimmen Sie dazu den kleinsten Zeugen gegen die Primalität von n.

Übung 8.6.8. Bestimmen Sie die Anzahl der Miller-Rabin-Zeugen für die Zusammengesetztheit von 221 in $\{1, 2, \ldots, 220\}$. Vergleichen Sie Ihr Resultat mit der Schranke aus Theorem 8.4.3.

Übung 8.6.9. Schreiben Sie ein *LiDIA*-Programm, daß den Miller-Rabin-Test implementiert und bestimmen Sie damit die kleinste 512-Bit-Primzahl.

9. Public-Key Verschlüsselung

9.1 Idee

Ein zentrales Problem, das bei der Anwendung der bis jetzt beschriebenen symmetrischen Verschlüsselungsverfahren auftritt, ist die Verteilung und Verwaltung der Schlüssel. Immer wenn Alice und Bob miteinander geheim kommunizieren wollen, müssen sie vorher einen geheimen Schlüssel austauschen. Dafür muß ein sicherer Kanal zur Verfügung stehen. Ein Kurier muß den Schlüssel überbringen oder eine andere Lösung muß gefunden werden. Dieses Problem wird um so schwieriger, je mehr Teilnehmer in einem Netzwerk miteinander geheim kommunizieren wollen. Wenn es n Teilnehmer im Netz gibt und wenn jeder mit jedem vertraulich kommunizieren will, dann besteht eine Möglichkeit, das zu organisieren, darin, daß je zwei Teilnehmer miteinander einen geheimen Schlüssel austauschen. Dabei müssen dann $n(n-1)/2$ Schlüssel geheim übertragen werden, und genausoviele Schlüssel müssen irgendwo geschützt gespeichert werden. Laut [38] gab es Jahr 2002 ungefähr $6 \cdot 10^8$ Internet-Nutzer. Wollten die alle einen Schlüssel austauschen wären das etwa $1.8 \cdot 10^{17}$ Schlüssel. Das wäre organisatorisch nicht zu bewältigen.

Eine andere Möglichkeit besteht darin, die gesamte Kommunikation über eine zentrale Stelle laufen zu lassen. Jeder Teilnehmer muß dann mit der Zentrale einen Schlüssel austauschen. Auch das ist organisatorisch kaum zu bewältigen. Außerdem setzt dieses Vorgehen voraus, daß jeder Teilnehmer der Zentralstelle vertraut. Die Zentralstelle kann ja die ganze Kommunikation mithören. Außerdem muß die Zentralstelle alle Schlüssel sicher speichern.

Das Schlüsselmanagement wird einfacher, wenn man Public-Key-Verfahren einsetzt. Solche Verfahren wurden schon in Abschnitt 4.2 eingeführt. In einem Public-Key-System muß man nur den Entschlüsselungsschlüssel geheimhalten. Er heißt *geheimer Schlüssel* oder *Private Key.* Den entsprechenden Verschlüsselungsschlüssel kann man veröffentlichen. Er heißt *öffentlicher Schlüssel* oder *Public Key.* Aus dem öffentlichen Schlüssel kann der geheime nicht in vertretbarer Zeit ermittelt werden. Jeder Teilnehmer schreibt seinen öffentlichen Schlüssel in ein öffentliches Verzeichnis. Will Bob an Alice eine verschlüsselte Nachricht schicken, so besorgt er sich ihren öffentlichen Schlüssel aus dem Verzeichnis, verschlüsselt damit die Nachricht und schickt sie an Alice. Er braucht vorher keinen geheimen Schlüssel mit Alice auszutauschen.

Beispiel 9.1.1. Ein Verzeichnis öffentlicher Schlüssel sieht z.B. folgendermaßen aus:

Name	öffentlicher Schlüssel
Buchmann	131213112359127531923751341 23
Paulus	842283496450982361026311357 68
Alice	546282919826246381210250325 10
⋮	⋮

Bei der Verwendung von Public-Key-Verfahren brauchen keine geheimen Schlüssel ausgetauscht zu werden. Das ist ein großer Vorteil. Es gibt aber ein anderes Problem. Wer an Alice eine Nachricht schicken will, muß sicher sein, daß der öffentliche Schlüssel, den er aus dem öffentlichen Schlüsselverzeichnis erhält, tatsächlich der öffentliche Schlüssel von Alice ist. Gelingt es nämlich einem Angreifer, den öffentlichen Schlüssel von Alice in der Datenbank durch seinen eigenen zu ersetzen, dann kann er die Nachrichten, die für Alice bestimmt sind, lesen. Das öffentliche Schlüsselverzeichnis muß also vor Veränderung geschützt werden. Dies macht man mit elektronischen Signaturen (siehe Kapitel 13).

Public-Key-Verschlüsselungsverfahren erleichtern das Schlüsselmanagement und haben viele Anwendungen. Man kann mit ihnen z.B. digitale Signaturen erzeugen, wie wir in Kapitel 13 sehen werden.

Leider sind die bekannten Public-Key-Verfahren nicht so effizient wie viele symmetrische Verfahren. Darum benutzt man in der Praxis Kombinationen aus Public-Key-Verfahren und symmetrischen Verfahren. Eine Möglichkeit ist das sogenannte *Hybrid-Verfahren*. Soll ein Dokument d verschlüsselt werden, so erzeugt man einen *Sitzungsschlüssel*, der nur zur Verschlüsselung von d verwendet wird und danach nicht mehr. Man verschlüsselt d mit dem Sitzungsschlüssel und erhält den Chiffretext c. Den Sitzungsschlüssel verschlüsselt man mit einem Public-Key-Verfahren und hängt ihn in verschlüsselter Form an den Chiffretext c an. Der Empfänger entschlüsselt den Sitzungsschlüssel, den er vorher noch nicht kannte. Mit dem Sitzungsschlüssel entschlüsselt er den Chiffretext c und erhält das Dokument d. Bei diesem Verfahren wird das Public-Key-Verfahren nur zum Schlüsselaustausch verwendet.

Wir werden in diesem Kapitel einige wichtige Public-Key-Verfahren beschreiben.

9.2 Sicherheit

Die Sicherheit von Verschlüsselungsverfahren wurde bereits in Abschnitt 4.3 behandelt. Die dort gemachten Aussagen gelten natürlich speziell für Public-Key-Verschlüsselungsverfahren. In diesem Abschnitt referieren wir kurz die wichtigen Sicherheitsmodelle für Public-Key-Verschlüsselungsverfahren.

9.2.1 Sicherheit des privaten Schlüssels

Ein Public-Key-Verschlüsselungsverfahren kann nur sicher sein, wenn es unmöglich ist, in vertretbarer Zeit aus den öffentlich verfügbaren Informationen, also insbesondere aus den öffentlichen Schlüsseln, die privaten Schlüssel der Nutzer zu berechnen. Bei den heute bekannten Public-Key-Verschlüsselungsverfahren wird das dadurch gewährleistet, daß gewisse Berechnungsprobleme der Zahlentheorie schwer zu lösen sind. Welche Berechnungsprobleme das sind, werden wir bei der Beschreibung der einzelnen Public-Key-Verfahren erklären. Ob diese zahlentheoretischen Probleme immer schwierig bleiben, ist nicht bekannt. Eine gute Idee kann ein bewährtes Public-Key-Kryptosystem unsicher machen. Es ist zum Beispiel bekannt, daß Quantencomputer alle gängigen Public-Key-Verschlüsselungsverfahren unsicher machen (siehe [74]). Es ist nur nicht bekannt, ob und wann solche Computer wirklich gebaut werden können. Es ist daher unbedingt nötig, Sicherheitsinfrastrukturen so zu konstruieren, daß die verwendeten kryptographischen Basistechniken leicht ausgetauscht werden können.

9.2.2 Semantische Sicherheit

Der Versuch, private Schlüssel zu finden, ist nicht die einzige Angriffsmöglichkeit auf ein Public-Key-Verschlüsselungsverfahren. In Abschnitt 4.3.2 wurde gezeigt, wie ein Angreifer leicht den Klartext zu einem Chiffretext bestimmen kann, wenn nur wenige Klartexte möglich sind und wenn das Verschlüsselungsverfahren deterministisch ist, also bei festem Schlüssel jeder Klartext immer zum selben Schlüsseltext verschlüsselt wird.

Soll nachgewiesen werden daß ein Public-Key-Verschlüsselungsverfahren sicher ist, ist ein Sicherheitsmodell nötig. Ein solches Modell ist die *semantische Sicherheit* . Es wurde in [34] eingeführt. Public-Key-Kryptoverfahren gelten heute als sicher gegen passive Angriffe, wenn sie semantisch sicher sind.

Shoup [76] gibt folgende intuitive Beschreibung von semantischer Sicherheit als Spiel. Die Mitspieler sind der Gute, der angegriffen wird, und der Böse, der versucht, den Guten anzugreifen. Das Spiel läuft so ab:

1. Der Gute gibt dem Bösen seinen öffentlichen Schlüssel.
2. Der Böse wählt zwei Nachrichten m_0 und m_1 und gibt sie dem Guten.
3. Der Gute wirft eine Münze. Bei Zahl verschlüsselt er m_0. Bei Kopf verschlüsselt er m_1. Das Ergebnis ist der Schlüsseltext c. Der Gute gibt dem Bösen den Schlüsseltext.
4. Der Böse rät, welcher Klartext verschlüsselt wurde. Er gewinnt, wenn er den richtigen Klartext rät.

Das Public-Key-Verschlüsselungsverfahren ist semantisch sicher, wenn der Böse dieses Spiel nicht mit einer Wahrscheinlichkeit deutlich größer als 1/2

gewinnen kann. Dann kann der Böse garnichts über den Klartext aus dem Chiffretext erfahren.

Wir haben in Abschnitt 4.3.2 bereits dargestellt, daß semantisch sichere Verschlüsselungsverfahren randomisiert sein müssen. Wie diese Randomisierung funktioniert, wird bei der Beschreibung der einzelnen Public-Key-Verschlüsselungsverfahren dargestellt.

9.2.3 Chosen-Ciphertext-Sicherheit

Semantische Sicherheit bietet Schutz gegen passive Angriffe. Bei aktiven Angriffen reicht das aber nicht. Ein aktiver Angriff, der RSA angreift, wurde zum Beispiel in [13] beschrieben. Die heute aktzeptierte Definition von Sicherheit gegen aktive Angriffe findet man in [28], [58] und [62]. Shoup [76] erklärt auch dieses Modell als Spiel. Die Mitspieler sind wieder der Gute und der Böse. Das Spiel läuft so ab:

1. Der Gute gibt dem Bösen seinen öffentlichen Schlüssel.
2. Der Böse kann sich vom Guten Chiffretexte seiner Wahl entschlüsseln lassen. Damit kann er versuchen, die Auswahl im nächsten Schritt vorzubereiten.
3. Der Böse wählt zwei Nachrichten m_0 und m_1 und gibt sie dem Guten.
4. Der Gute wirft eine Münze. Bei Zahl verschlüsselt er m_0. Bei Kopf verschlüsselt er m_1. Das Ergebnis ist der Schlüsseltext c. Der Gute gibt dem Bösen den Schlüsseltext.
5. Der Böse kann sich vom Guten Chiffretexte seiner Wahl entschlüsseln lassen, nur nicht c. Damit kann er versuchen, die Auswahl im nächsten Schritt vorzubereiten.
6. Der Böse rät, welcher Klartext verschlüsselt wurde. Er gewinnt, wenn er den richtigen Klartext rät.

In diesem Spiel wird es dem Angreifer erlaubt, alles zu tun außer den Schlüsseltext c zu entschlüsseln. Der Angreifer könnte sich zum Beispiel als Kollege des Guten ausgeben und sich unverdächtige Schlüsseltexte entschlüsseln lassen.

Das Public-Key-Verschlüsselungsverfahren ist sicher gegen Chosen-Ciphertext-Angriffe, wenn es für den Bösen keine effiziente Strategie gibt, die es ihm ermöglicht, dieses Spiel mit einer Wahrscheinlichkeit deutlich größer als $1/2$ zu gewinnen. Dann kann der Böse garnichts über den Klartext aus dem Chiffretext erfahren.

Es kann gezeigt werden daß Public-Key-Verschlüsselungssysteme, die im obigen Sinne sicher gegen Chosen-Ciphertext-Angriffe sicher sind, einem Angreifer auch keine Möglichkeit bieten, den Schlüsseltext so zu verändern, dass sich der Klartext in kontrollierter Weise ändert. Diese Eigenschaft heißt *Non-Malleability*.

9.2.4 Sicherheitsbeweise

Heute sind keine nachweisbar schwierigen mathematischen Probleme bekannt, die zur Konstruktion von Public-Key-Verschlüsselungsverfahren verwendet werden können. Daher gibt es auch keine beweisbar sicheren Public-Key-Verschlüsselungsverfahren. Daher werden *Sicherheitsreduktionen* verwendet. In einer solchen Reduktion wird bewiesen, daß das Verschlüsselungsverfahren sicher ist, solange wenige, klar definierte, mathematische Berechnungsprobleme schwierig zu lösen sind. Damit ist die Sicherheit auf wenige mathematische Probleme reduziert. Für Benutzer ist es damit einfach, diese Sicherheit einzuschätzen. Die Entwicklung der Schwierigkeit der zugrunde liegenden Berechnungsprobleme muss beobachtet werden. Solange für keines der grundlegenden Probleme eine effiziente Lösung gefunden wurde, ist die Sicherheit gewährleistet. Sobald auch nur eins dieser Probleme leicht lösbar wird, ist das Verschlüsselungsverfahren nicht mehr sicher.

9.3 Das RSA-Verfahren

Das RSA-Verfahren, benannt nach seinen Erfindern Ron Rivest, Adi Shamir und Len Adleman, war das erste Public-Key Verschlüsselungsverfahren und ist noch heute das wichtigste. Seine Sicherheit hängt eng mit der Schwierigkeit zusammen, große Zahlen in ihre Primfaktoren zu zerlegen.

9.3.1 Schlüsselerzeugung

Wir erklären, wie Bob seinen privaten und seinen öffentlichen RSA-Schlüssel erzeugt.

Bob wählt nacheinander zufällig zwei Primzahlen p und q (siehe Abschnitt 8.5) und berechnet das Produkt

$$n = pq.$$

Zusätzlich wählt Bob eine natürliche Zahl e mit

$$1 < e < \varphi(n) = (p-1)(q-1) \text{ und } \gcd(e,(p-1)(q-1)) = 1$$

und berechnet eine natürliche Zahl d mit

$$1 < d < (p-1)(q-1) \text{ und } de \equiv 1 \bmod (p-1)(q-1). \tag{9.1}$$

Da $\gcd(e,(p-1)(q-1)) = 1$ ist, gibt es eine solche Zahl d tatsächlich. Sie kann mit dem erweiterten euklidischen Algorithmus berechnet werden (siehe Abschnitt 2.9). Man beachte, daß e stets ungerade ist. Der öffentliche Schlüssel besteht aus dem Paar (n,e). Der private Schlüssel ist d. Die Zahl n heißt *RSA-Modul*, e heißt *Verschlüsselungsexponent* und d heißt *Entschlüsselungsexponent*.

Beispiel 9.3.1. Als Primzahlen wählt Bob die Zahlen $p = 11$ und $q = 23$. Also ist $n = 253$ und $(p-1)(q-1) = 10 * 22 = 4 \cdot 5 \cdot 11$. Das kleinstmögliche e ist $e = 3$. Der erweiterte euklidische Algorithmus liefert $d = 147$.

In Abschnitt 9.3.4 erläutern wir dann, warum es sehr schwierig ist, aus dem öffentlichen Schlüssel den privaten zu bestimmen.

9.3.2 Verschlüsselung

Wir erklären zuerst, wie man mit dem RSA-Verfahren Zahlen verschlüsselt und danach, wie man daraus eine Art Blockchiffre machen kann.

In der ersten Variante besteht der Klartextraum aus allen natürlichen Zahlen m mit

$$0 \leq m < n.$$

Ein Klartext m wird verschlüsselt zu

$$c = m^e \bmod n. \tag{9.2}$$

Jeder, der den öffentlichen Schlüssel (n, e) kennt, kann die Verschlüsselung durchführen. Bei der Berechnung von $m^e \bmod n$ wird schnelle Exponentiation verwendet (siehe Abschnitt 3.12).

Beispiel 9.3.2. Wie in Beispiel 9.3.1 ist $n = 253$ und $e = 3$. Der Klartextraum ist also $\{0, 1, \ldots, 252\}$. Die Zahl $m = 165$ wird zu $165^3 \bmod 253 = 110$ verschlüsselt.

Wir zeigen nun, wie man mit der RSA-Verschlüsselungsmethode eine Art Blockchiffre realisieren kann. Wir nehmen an, daß das verwendete Alphabet Σ genau N Zeichen hat und die Zeichen die Zahlen $0, 1, \ldots, N-1$ sind. Wir setzen

$$k = \lfloor \log_N n \rfloor. \tag{9.3}$$

Ein Block $m_1 \ldots m_k$, $m_i \in \Sigma$, $1 \leq i \leq k$, wird in die Zahl

$$m = \sum_{i=1}^{k} m_i N^{k-i}$$

verwandelt. Beachte, daß wegen (9.3)

$$0 \leq m \leq (N-1) \sum_{i=1}^{k} N^{k-i} = N^k - 1 < n$$

gilt. Im folgenden werden wir häufig die Blöcke mit den durch sie dargestellten Zahlen identifizieren. Der Block m wird verschlüsselt, indem $c = m^e \bmod n$ bestimmt wird. Die Zahl c kann dann wieder zur Basis N geschrieben werden.

Die N-adische Entwicklung von c kann aber die Länge $k+1$ haben. Wir schreiben also

$$c = \sum_{i=0}^{k} c_i N^{k-i}, \quad c_i \in \Sigma, 0 \leq i \leq k.$$

Der Schlüsselblock ist dann

$$c = c_0 c_1 \dots c_k.$$

In der beschriebenen Weise bildet RSA Blöcke der Länge k injektiv auf Blöcke der Länge $k+1$ ab. Dies ist keine Blockchiffre im Sinne von Definition 4.6.1. Trotzdem kann man mit der beschriebenen Blockversion des RSA-Verfahrens den ECB-Mode und den CBC-Mode (mit einer kleinen Modifikation) zum Verschlüsseln anwenden. Es ist aber nicht möglich, den CFB-Mode oder den OFB-Mode zu benutzen, weil in beiden Modes nur die Verschlüsselungsfunktion verwendet wird, und die ist ja öffentlich. Also kennt sie auch jeder Angreifer.

Beispiel 9.3.3. Wir setzen Beispiel 9.3.1 fort. Verwende $\Sigma = \{0, a, b, c\}$ mit der Entsprechung

0	a	b	c
0	1	2	3

Bei Verwendung von $n = 253$ ist $k = \lfloor \log_4 253 \rfloor = 3$. Das ist die Länge der Klartextblöcke. Die Länge der Schlüsseltextblöcke ist also 4. Es soll der Klartextblock *abb* verschlüsselt werden. Dieser entspricht dem String 122 und damit der Zahl

$$m = 1 * 4^2 + 2 * 4^1 + 2 * 4^0 = 26.$$

Diese Zahl wird zu

$$c = 26^3 \bmod 253 = 119$$

verschlüsselt. Schreibt man diese zur Basis 4, so erhält man

$$c = 1 * 4^3 + 3 * 4^2 + 1 * 4 + 3 * 1.$$

Der Schlüsseltextblock ist dann

$$acac.$$

9.3.3 Entschlüsselung

Die Entschlüsselung von RSA beruht auf folgendem Satz.

Theorem 9.3.4. *Sei (n, e) ein öffentlicher und d der entsprechende private Schlüssel im RSA-Verfahren. Dann gilt*

$$(m^e)^d \bmod n = m$$

für jede natürliche Zahl m mit $0 \leq m < n$.

Beweis. Da $ed \equiv 1 \bmod (p-1)(q-1)$ ist, gibt es eine ganze Zahl l, so daß

$$ed = 1 + l(p-1)(q-1)$$

ist. Daher ist

$$(m^e)^d = m^{ed} = m^{1+l(p-1)(q-1)} = m(m^{(p-1)(q-1)})^l.$$

Aus dieser Gleichung erkennt man, daß

$$(m^e)^d \equiv m(m^{(p-1)})^{(q-1)l} \equiv m \bmod p$$

gilt. Falls p kein Teiler von m ist, folgt diese Kongruenz aus dem kleinen Satz von Fermat. Andernfalls ist die Behauptung trivial. Dann sind nämlich beide Seiten der Kongruenz $0 \bmod p$. Genauso sieht man ein, daß

$$(m^e)^d \equiv m \bmod q$$

gilt. Weil p und q verschiedene Primzahlen sind, erhält man also

$$(m^e)^d \equiv m \bmod n.$$

Da $0 \leq m < n$ ist, erhält man die Behauptung des Satzes. □

Wurde also c wie in (9.2) berechnet, kann man m mittels

$$m = c^d \bmod n$$

rekonstruieren. Damit ist gezeigt, daß das RSA-Verfahren tatsächlich ein Kryptosystem ist, daß es nämlich zu jeder Verschlüsselungsfunktion eine Entschlüsselungsfunktion gibt.

Beispiel 9.3.5. Wir schließen die Beispiele 9.3.1 und 9.3.3 ab. Dort wurde ja $n = 253$, $e = 3$ und $d = 147$ gewählt. Außerdem wurde $c = 119$ berechnet. Tatsächlich gilt $119^{147} \bmod 253 = 26$. Damit ist der Klartext rekonstruiert.

9.3.4 Sicherheit des geheimen Schlüssels

Es wurde behauptet, daß das RSA-Verfahren ein Public-Key-System ist. Es muß also praktisch unmöglich sein, aus dem öffentlichen Schlüssel (n, e) den geheimen Schlüssel d zu berechnen. In diesem Abschnitt werden wir zeigen,

daß die Bestimmung des geheimen Schlüssels d aus dem öffentlichen Schlüssel (n, e) genauso schwierig ist, wie die Zerlegung des RSA-Moduls n in seine Primfaktoren. Dies ist ein sehr wichtiges Ergebnis. Das kryptographische Problem, den geheimen RSA-Schlüssel zu finden, ist damit auf ein Problem von allgemeinem mathematischen Interesse, nämlich das Faktorisierungsproblem für natürliche Zahlen, reduziert. Das Faktorisierungsproblem wird in der Zahlentheorie schon seit Jahrhunderten diskutiert. Es gilt als sehr schwierig (siehe Kapitel 10). Mit dem Faktorisierungsproblem beschäftigen sich viele Wissenschaftler auf der ganzen Welt. Sollte es sich wider Erwarten als einfach herausstellen, so kann man davon ausgehen, daß dies an mehreren Orten gleichzeitig gefunden und daher schnell bekannt würde. Niemand könnte aus der Unsicherheit von RSA dann für längere Zeit einen Vorteil ziehen.

Die beschriebene Eigenschaft teilt das RSA-Verfahren mit den anderen Public-Key-Verfahren. Die Sicherheit der bekannten Public-Key-Verfahren steht in engem Zusammenhang mit der Lösbarkeit schwieriger Probleme von allgemeinem mathematischen Interesse. Dies ist bei den meisten symmetrischen Verfahren nicht der Fall.

Wir beweisen jetzt die angekündigte Äquivalenz.

Wenn der Angreifer Oskar die Primfaktoren p und q des RSA-Moduls n kennt, kann er aus n und dem Verschlüsselungsexponenten e den geheimen Schlüssel d durch Lösen der Kongruenz $de \equiv 1 \bmod (p-1)(q-1)$ berechnen.

Wir zeigen, daß die Umkehrung auch gilt, wie man nämlich aus n, e, d die Faktoren p und q berechnet.

Dazu setzen wir

$$s = \max\{t \in \mathbb{N} : 2^t \text{ teilt } ed - 1\}.$$

und

$$k = (ed - 1)/2^s.$$

Lemma 9.3.6. *Für alle zu n teilerfremden ganzen Zahlen a gilt* $\operatorname{order}(a^k + n\mathbb{Z}) \in \{2^i : 0 \leq i \leq s\}$.

Beweis. Sei a eine zu n teilerfremde ganze Zahl. Nach Theorem 9.3.4 gilt $a^{ed-1} \equiv 1 \bmod n$. Daraus folgt $(a^k)^{2^s} \equiv 1 \bmod n$. Also impliziert Theorem 3.9.2, daß die Ordnung von $a^k + n\mathbb{Z}$ ein Teiler von 2^s ist. □

Der Algorithmus, der n faktorisiert, beruht auf folgendem Theorem.

Theorem 9.3.7. *Sei a eine zu n teilerfremde ganze Zahl. Wenn die Ordnung von a^k mod p und mod q verschieden ist, so ist* $1 < \gcd(a^{2^t k} - 1, n) < n$ *für ein* $t \in \{0, 1, 2, \ldots, s-1\}$.

Beweis. Nach Lemma 9.3.6 liegt die Ordnung von $a^k \bmod p$ und $\bmod q$ in der Menge $\{2^i : 0 \leq i \leq s\}$. Sei die Ordnung von $a^k \bmod p$ größer als die von $a^k \bmod q$. Die Ordnung von $a^k \bmod q$ sei 2^t. Dann gilt $t < s$, $a^{2^t k} \equiv 1 \bmod q$, aber $a^{2^t k} \not\equiv 1 \bmod p$ und daher $\gcd(a^{2^t k} - 1, n) = q$. □

Um n zu faktorisieren, wählt man zufällig und gleichverteilt eine Zahl a in der Menge $\{1, \ldots, n-1\}$. Dann berechnet man $g = \gcd(a, n)$. Ist $g > 1$, so ist g ein echter Teiler von n. Der wurde ja gesucht. Also ist der Algorithmus fertig. Ist $g = 1$, so berechnet man

$$g = \gcd(a^{2^t k}, n), \quad t = s-1, s-2, \ldots, 0.$$

Findet man dabei einen Teiler von n, dann ist der Algorithmus fertig. Andernfalls wird ein neues a gewählt und dieselben Operationen werden für dieses a ausgeführt. Wir wollen jetzt zeigen, daß in jeder Iteration dieses Verfahrens die Wahrscheinlichkeit dafür, daß ein Primteiler von n gefunden wird, wenigstens $1/2$ ist. Die Wahrscheinlichkeit dafür, daß das Verfahren nach r Iterationen einen Faktor gefunden hat, ist dann mindestens $1 - 1/2^r$.

Theorem 9.3.8. *Die Anzahl der zu n primen Zahlen a in der Menge $\{1, 2, \ldots, n-1\}$, für die a^k mod p und mod q eine verschiedene Ordnung hat, ist wenigstens $(p-1)(q-1)/2$.*

Beweis. Sei g eine Primitivwurzel mod p und mod q. Eine solche existiert nach dem chinesischen Restsatz 3.15.2.

Zuerst nehmen wir an, die Ordnung von g^k mod p sei größer als die Ordnung von g^k mod q. Diese beiden Ordnungen sind nach Lemma 9.3.6 Potenzen von 2. Sei x eine ungerade Zahl in $\{1, \ldots, p-1\}$ und sei $y \in \{0, 1, \ldots, q-2\}$. Sei a eine Lösung der simultanen Kongruenz

$$a \equiv g^x \bmod p, \quad a \equiv g^y \bmod q. \tag{9.4}$$

Dann ist die Ordnung von a^k mod p dieselbe wie die Ordnung von g^k mod p. Die Ordnung von a^k mod q ist aber höchstens so groß wie die Ordnung von g^k mod q, also kleiner als die Ordnung von a^k mod p. Schließlich sind diese Lösungen mod n paarweise verschieden, weil g eine Primitivwurzel mod p und q ist. Damit sind $(p-1)(q-1)/2$ zu n teilerfremde Zahlen a gefunden, für die die Ordnung von a^k mod p und q verschieden ist.

Ist die Ordnung von g^k mod q größer als die mod p, so geht man genauso vor.

Sei schließlich angenommen, daß die Ordnung von g^k mod p und mod q gleich ist. Da $p-1$ und $q-1$ beide gerade sind, ist diese Ordnung wenigstens 2. Wir bestimmen die gesuchten Zahlen a wieder als Lösung der simultanen Kongruenz (9.4). Die Exponentenpaare (x, y) müssen diesmal aber aus einer geraden und einer ungeraden Zahl bestehen. Es bleibt dem Leser überlassen, zu verifizieren, daß es $(p-1)(q-1)/2$ mod n verschiedene Lösungen gibt, und diese die gewünschte Eigenschaft haben. □

Aus Theorem 9.3.8 folgt unmittelbar, daß die Erfolgswahrscheinlichkeit des Faktorisierungsverfahrens in jeder Iteration wenigstens $1/2$ ist.

Beispiel 9.3.9. In Beispiel 9.3.1 ist $n = 253$, $e = 3$ und $d = 147$. Also ist $ed - 1 = 440$. Wenn man $a = 2$ verwendet, so erhält man $\gcd(2^{220} - 1, 253) = \gcd(2^{110} - 1, 253) = 253$. Aber $\gcd(2^{55} - 1, 253) = 23$.

9.3.5 RSA und Faktorisierung

Im vorigen Abschnitt wurde gezeigt, daß es genauso schwer ist, den geheimen RSA-Schlüssel zu finden wie den RSA-Modul zu faktorisieren. Der Angreifer kann aber auch das Ziel haben, aus einem RSA-Schlüsseltext den zugehörigen Klartext zu ermitteln. Es ist nicht bekannt, ob es dazu nötig ist, den geheimen RSA-Schlüssel zu finden. Ob es die Möglichkeit gibt, RSA-Schlüsseltexte zu entschlüsseln, ohne den RSA-Modul zu faktorisieren, ist ein wichtiges offenes Problem der Public-Key-Kryptographie.

Aber selbst wenn man zeigen könnte, daß es genauso schwer ist, das RSA-Verfahren zu brechen wie den RSA-Modul n zu faktorisieren, würde das nicht automatisch bedeuten, daß RSA sicher ist. Es ist nämlich nicht bekannt, ob das Faktorisierungsproblem für natürliche Zahlen schwer zu lösen ist. Dies wird in Kapitel 10 diskutiert. Es ist daher sehr gefährlich, Public-Key-Kryptographie-Anwendungen nur mit RSA zu realisieren. Statt dessen sollte man Public-Key-Systeme so implementieren, daß die Basismechanismen leicht ausgetauscht werden können. Außerdem müssen Forscher nach Alternativen zu RSA suchen, die im Notfall RSA ersetzen können. Einige Alternativen werden später noch beschrieben.

9.3.6 Auswahl von p und q

Um die Faktorisierung des RSA-Moduls möglichst schwer zu machen, werden seine Primfaktoren p und q etwa gleich groß gewählt. Bei einem 1024-Bit RSA-Modul sind die beiden Faktoren z.B. 512-Bit-Zahlen. Manchmal wird noch verlangt, daß p und q so gewählt werden, daß bekannte Faktorisierungsalgorithmen kein leichtes Spiel haben. Es ist aber schwer, das vorherzusagen. Daher sollten p und q zufällig und möglichst gleichverteilt gewählt werden.

Nach heutiger Kenntnis sollte der RSA-Modul n wenigstens 512 Bits lang sein. Aus experimentellen Untersuchungen des Faktorisierungsproblems ergibt sich, daß bei längerfristiger Sicherheit der RSA-Modul 1024 oder sogar 2048 Bits lang sein sollte. Genauere Hinweise findet man in [45]. Solche Empfehlungen sind aber mit Vorsicht zu genießen, weil niemand den Fortschritt im Bereich der Algorithmen oder den Fortschritt bei der Hardwareentwicklung prognostizieren kann.

9.3.7 Auswahl von e

Der öffentliche Schlüssel e wird so gewählt, daß die Verschlüsselung effizient möglich ist, ohne daß die Sicherheit gefährdet wird. Die Wahl von $e = 2$ ist natürlich immer ausgeschlossen, weil $\varphi(n) = (p-1)(q-1)$ gerade ist und $\gcd(e, (p-1)(q-1)) = 1$ gelten muß. Der kleinste mögliche Verschlüsselungsexponent ist also $e = 3$. Verwendet man diesen Exponenten, so kann man mit einer Quadrierung und einer Multiplikation mod n verschlüsseln.

Beispiel 9.3.10. Sei $n = 253$, $e = 3$, $m = 165$. Um $m^e \bmod n$ zu berechnen, bestimmt man $m^2 \bmod n = 154$. Dann berechnet man $m^3 \bmod n = ((m^2 \bmod n) * m) \bmod n = 154 * 165 \bmod 253 = 110$.

Die Verwendung des Verschlüsselungsexponenten $e = 3$ ist aber nicht ungefährlich. Ein Angreifer kann nämlich die sogenannte *Low-Exponent-Attacke* anwenden. Diese beruht auf folgendem Satz.

Theorem 9.3.11. *Seien $e \in \mathbb{N}$, $n_1, n_2, \ldots, n_e \in \mathbb{N}$ paarweise teilerfremd und $m \in \mathbb{N}$ mit $0 \leq m < n_i$, $1 \leq i \leq e$. Sei $c \in \mathbb{N}$ mit $c \equiv m^e \bmod n_i$, $1 \leq i \leq e$, und $0 \leq c < \prod_{i=1}^{e} n_i$. Dann folgt $c = m^e$.*

Beweis. Die Zahl $c' = m^e$ erfüllt die simultane Kongruenz $c' \equiv m^e \bmod n_i$, $1 \leq i \leq e$ und es gilt $0 \leq c' < \prod_{i=1}^{e} n_i$, weil $0 \leq m < n_i$, $1 \leq i \leq e$, vorausgesetzt ist. Da eine solche Lösung der simultanen Kongruenz aber nach dem chinesischen Restsatz eindeutig bestimmt ist, folgt $c = c'$. □

Die Low-Exponent-Attacke kann man immer dann anwenden, wenn ein Klartext m mit e zueinander paarweise teilerfremden Moduln, aber immer mit demselben Verschlüsselungsexponenten e verschlüsselt wird. Es ist z.B. denkbar, daß eine Bank an e verschiedene Kunden dieselbe Nachricht verschlüsselt sendet. Dabei werden die verschiedenen öffentlichen Schlüssel n_i, $1 \leq i \leq e$, der Kunden benutzt, aber immer derselbe Verschlüsselungsexponent e. Dann kann der Angreifer Oskar den Klartext m folgendermaßen berechnen. Er kennt die Schlüsseltexte $c_i = m^e \bmod n_i$, $1 \leq i \leq e$. Mit dem chinesischen Restsatz berechnet er eine ganze Zahl c mit $c \equiv c_i \bmod n_i$, $1 \leq i \leq e$ und $0 \leq c < \prod_{i=1}^{e} n_i$. Nach Theorem 9.3.11 gilt $c = m^e$. Also kann Oskar m finden, indem er aus c die e-te Wurzel zieht. Dies ist in Polynomzeit möglich.

Beispiel 9.3.12. Wir wählen $e = 3$, $n_1 = 143$, $n_2 = 391$, $n_3 = 899$, $m = 135$. Dann ist $c_1 = 60$, $c_2 = 203$, $c_3 = 711$. Um den chinesischen Restsatz zu verwenden berechne x_1, x_2, x_3 mit $x_1 n_2 n_3 \equiv 1 \bmod n_1$, $n_1 x_2 n_3 \equiv 1 \bmod n_2$ und $n_1 n_2 x_3 \equiv 1 \bmod n_3$. Es ergibt sich $x_1 = -19$, $x_2 = -62$, $x_3 = 262$. Dann ist $c = (c_1 x_1 n_2 n_3 + c_2 n_1 x_2 n_3 + c_3 n_1 n_2 x_3) \bmod n_1 n_2 n_3 = 2460375$ und $m = 2460375^{1/3} = 135$.

Man kann die Low-Exponent-Attacke verhindern, indem man die Klartextblöcke kürzer wählt, als das eigentlich nötig ist, und die letzten Bits zufällig wählt. Dann ist es praktisch ausgeschlossen, daß zweimal derselbe Block verschlüsselt wird.

Eine andere Möglichkeit, die Low-Exponent-Attacke zu verhindern, besteht darin, größere Verschlüsselungsexponenten zu wählen, die aber immer noch eine effiziente Verschlüsselung zulassen. Üblich ist $e = 2^{16} + 1$ (siehe Übung 9.7.7).

9.3.8 Auswahl von *d*

Wird RSA in der Praxis eingesetzt, muß der private RSA-Schlüssel geschützt werden. Dazu kann der private RSA-Schlüssel zum Beispiel auf einer Chipkarte gespeichert werden. Damit dieser Schlüssel die Chipkarte nie verlassen muß, wird auf der Chipkarte auch entschlüsselt. Chipkarten haben aber keine besonders gute Performance. Um die RSA-Entschlüsselung zu beschleunigen, könnte man zuerst einen kleinen Entschlüsselungsschlüssel d wählen und dazu einen passenden Verschlüsselungschlüssel e berechnen. Aber das ist unsicher. D. Boneh und G. Durfee haben nämlich in [15] bewiesen, daß das RSA-Verfahren gebrochen werden kann, wenn $d < n^{0.292}$ ist.

9.3.9 Effizienz

Die Verschlüsselung beim RSA-Verfahren erfordert eine Exponentiation modulo n. Die Verschlüsselung ist um so effizienter, je kleiner der Exponent ist. Bei kleinen Exponenten muß man aber Vorkehrungen gegen die oben beschriebene Low-Exponent-Attacke treffen.

Die Entschlüsselung eines RSA-verschlüsselten Textes erfordert ebenfalls eine Exponentiation modulo n. Diesmal ist aber der Exponent d in derselben Größenordnung wie n. Die Verwendung kleiner Entschlüsselungsexponenten ist unsicher. Ist k die Bitlänge von n, so erfordert die Entschlüsselung k Quadrierungen modulo n und $k/2$ Multiplikationen modulo n^d, wenn man davon ausgeht, daß die Hälfte der Bits in der Binärentwicklung von n den Wert 1 haben. Da RSA-Moduln wenigstens 512 Bits lang sind, sind also wenigstens 512 Quadrierungen und 256 Multiplikationen nötig. Oft werden RSA-Entschlüsselungen auf langsamen Chipkarten durchgeführt. Das kann also lange dauern.

Die RSA-Entschlüsselung kann beschleunigt werden, wenn man den chinesischen Restsatz benutzt.

Alice möchte den Schlüsseltext c entschlüsseln. Ihr privater Schlüssel ist d. Alice berechnet

$$m_p = c^d \bmod p, \quad m_q = c^d \bmod q$$

und löst dann die simultane Kongruenz

$$m \equiv m_p \bmod p, \quad m \equiv m_q \bmod q.$$

Dann ist m der ursprüngliche Klartext. Um die simultane Kongruenz zu lösen, berechnet Alice mit dem erweiterten euklidischen Algorithmus ganze Zahlen y_p und y_q mit

$$y_p p + y_q q = 1.$$

Dann setzt sie

$$m = (m_p y_q q + m_q y_p p) \bmod n.$$

Man beachte, daß die Zahlen $y_p p \bmod n$ und $y_q q \bmod n$ nicht von der zu entschlüsselnden Nachricht abhängen, und daher ein für allemal vorberechnet werden können.

Beispiel 9.3.13. Um die Entschlüsselung aus Beispiel 9.3.5 zu beschleunigen, berechnet Alice

$$m_p = 119^7 \bmod 11 = 4, \quad m_q = 119^{15} \bmod 23 = 3,$$

sowie $y_p = -2$, $y_q = 1$ und setzt dann

$$m = (4 * 23 - 3 * 2 * 11) \bmod 253 = 26.$$

Wir zeigen, daß Entschlüsseln mit dem chinesischen Restsatz effizienter ist als das Standardverfahren. Dazu machen wir einige teilweise vereinfachende Annahmen. Wir nehmen an, daß der RSA-Modul n eine k-Bit-Zahl ist und daß die Primfaktoren p und q $k/2$-Bit-Zahlen sind. Die Multiplikation zweier Zahlen in $\{0, \ldots, n-1\}$ und die anschließende Reduktion des Ergebnisses modulo n benötigt Zeit Ck^2, wobei C eine Konstante ist. Wir verwenden also zur Multiplikation die Schulmethode. Die Berechnung $m = c^d \bmod n$ kostet dann Zeit $2Ck^3$, wobei l die Anzahl der Einsen in der Binärentwicklung von d ist. Die Berechnung von m_p und m_q kostet nur Zeit $Ck^2/2$. Ignoriert man also die Zeit, die zur Berechnung im chinesischen Restsatz verwendet wird, so hat man eine Beschleunigung um den Faktor 4. Die Anwendung des chinesischen Restsatzes erlaubt eine Vorberechnung und erfordert dann nur noch zwei modulare Multiplikationen. Weil d so groß ist, kann man die Zeit dafür tatsächlich ignorieren. Entschlüsseln mit dem chinesischen Restsatz ist also viermal so schnell wie die Standard-Entschlüsselungsmethode.

9.3.10 Multiplikativität

Sei (n, e) ein öffentlicher RSA-Schlüssel. Werden damit zwei Nachrichten m_1 und m_2 verschlüsselt, so erhält man

$$c_1 = m_1^e \bmod n, \quad c_2 = m_2^e \bmod n.$$

Es gilt dann

$$c = c_1 c_2 \bmod n = (m_1 m_2)^e \bmod n.$$

Wer die beiden Schlüsseltexte c_1 und c_2 kennt, kann daraus die Verschlüsselung von $m = m_1 m_2$ berechnen, ohne den Klartext $m = m_1 m_2$ zu kennen. Dies eröffnet eine Betrugsmöglichkeit.

Damit der Empfänger merken kann, daß hier ein Betrug vorliegt, kann man den Klartextraum auf Klartexte mit bestimmter Struktur einschränken, und zwar so, daß das Produkt von zwei Klartexten kein gültiger Klartext ist.

Man kann z.B. dafür sorgen, daß das erste und letzte Byte in einem gültigen Klartext immer übereinstimmen. Wenn m_1 und m_2 diese Bedingung erfüllen, wird mit ziemlicher Sicherheit das Produkt $m_1 m_2$ diese Eigenschaft nicht haben. Wenn Alice also den Schlüsseltext c empfängt und feststellt, daß c die falsche Struktur hat, dann weiß sie, daß m keine gültige Nachricht ist.

9.3.11 Sichere Verwendung

Bis jetzt wurden eine Reihe von Angriffen auf das RSA-Verfahren beschrieben, die selbst dann möglich sind, wenn die RSA-Parameter richtig gewählt sind. Aber selbst wenn man die entsprechenden Gegenmaßnahmen ergreift, ist RSA nicht semantisch sicher, geschweige denn sicher gegen Chosen-Ciphertext-Attacks. Hierzu müßte RSA randomisiert werden. Das wurde in Abschnitt 4.3.2 erklärt. Die heute als sicher betrachtete Verwendung des RSA-Verschlüsselungsverfahrens findet man im Standard PKCS# 1 [60]. Sie beruht auf dem OAEP-Verfahren (optimal asymmetric encryption protocol) von Bellare und Rogaway [9],[77]. Grundsätzlich funktioniert das so:

Sei k eine natürliche Zahl mit der Eigenschaft, daß die maximale Laufzeit, die ein optimaler Angreiferalgorithmus verbrauchen kann, deutlich kleiner als 2^k ist. Sei b die binäre Länge des RSA-Moduls. Also ist $b \geq 1024$ und sei $l = b - k - 1$. Benötigt werden eine Expansionsfunktion Funktion

$$G : \{0,1\}^k \to \{0,1\}^l$$

und eine Kompressionsfunktion

$$H : \{0,1\}^l \to \{0,1\}^k.$$

Diese Funktionen sind öffentlich bekannt. Der Klartextraum ist $\{0,1\}^l$. Soll ein Klartext $m \in \{0,1\}^l$ verschlüsselt werden, so wird zuerst eine Zufallszahl $r \in \{0,1\}^k$ gewählt. Dann wird der Chiffretext

$$c = ((m \oplus G(r)) \circ (r \oplus H(m \oplus G(r))))^e \bmod n$$

berechnet. Bei der Entschlüsselung berechnet der Empfänger zuerst

$$(m \oplus G(r)) \circ (r \oplus H(m \oplus G(r))) = c^d \bmod n.$$

Dann kann er

$$r = (r \oplus H(m \oplus G(r))) \oplus H(m \oplus G(r))$$

und danach

$$m = (m \oplus G(r)) \oplus G(r)$$

berechnen. Der Klartext wird also zu $m \oplus G(r)$ randomisiert. Der Zufallswert r wird zu $(r \oplus H(m \oplus G(r)))$ maskiert. Sind G und H zufällig gewählte Funktionen, kann die Sicherheit dieses Verfahrens gegen Chosen-Ciphertext-Angriffe unter der Voraussetzung bewiesen werden, daß die RSA-Funktion $m \mapsto m^e \bmod m$ nicht in vertretbarer Zeit invertiert werden kann. In der Praxis werden die Funktionen G und H aus kryptographischen Hashfunktionen (siehe Kapitel 12) konstruiert.

9.3.12 Verallgemeinerung

Wir erklären, wie man das RSA-Verfahren verallgemeinern kann. Sei G eine endliche Gruppe. Die Ordnung o dieser Gruppe sei nur Alice bekannt. Alice wählt einen Verschlüsselungsexponenten $e \in \{2, \ldots, o-1\}$, der zu o teilerfremd ist. Ihr öffentlicher Schüssel ist (G, e). Der geheime Schlüssel ist eine Zahl $d \in \{2, \ldots, o-1\}$, für die $ed \equiv 1 \bmod o$ gilt. Will man eine Nachricht $m \in G$ verschlüsseln, so berechnet man $c = m^e$. Es gilt dann $c^d = m^{ed} = m^{1+ko} = m$ für eine ganze Zahl k. Auf diese Weise kann c also entschlüsselt werden. Die Funktion

$$G \to G, \quad m \mapsto m^e$$

ist eine sogenannte *Trapdoor-One-Way-Funktion* oder einfach *Trapdoor-Funktion*. Sie kann leicht berechnet werden. Sie kann aber nicht in vertretbarer Zeit invertiert werden. Kennt man aber ein Geheimnis, hier die Gruppenordnung, so kann man die Funktion auch effizient invertieren.

Das beschriebene Verfahren ist tatsächlich eine Verallgemeinerung des RSA-Verfahrens: Im RSA-Verfahren ist $G = (\mathbb{Z}/n\mathbb{Z})^*$. Die Ordnung von G ist $o = (p-1)(q-1)$. Ist die Ordnung bekannt, so kann man, wie in Abschnitt 9.3.4 beschrieben, die Faktoren p und q berechnen. Solange also die Faktorisierung von n nicht bekannt ist, ist auch die Ordnung von $G = (\mathbb{Z}/n\mathbb{Z})^*$ geheim. Für RSA ist der öffentliche Schlüssel (n, e). Die Zahl n legt die Gruppe fest, in der gerechnet wird, und e ist der Verschlüsselungsexponent. Der geheime Schlüssel d erfüllt $ed \equiv 1 \bmod o$.

Um das allgemeine Prinzip verwenden zu können, muß man in der Lage sein, endliche Gruppen samt ihrer Ordnung zu erzeugen. Die Ordnung muß man aber geheimhalten und es muß anderen, die die Ordnung nicht kennen, trotzdem möglich sein, effizient in der Gruppe zu rechnen. Es sind tatsächlich solche Gruppen vorgeschlagen worden, z.B. elliptische Kurven über $\mathbb{Z}/n\mathbb{Z}$ mit $n = pq$. Die Schwierigkeit, die Ordnung dieser Gruppen zu berechnen, beruht aber in allen Fällen auf der Schwierigkeit, natürliche Zahlen in ihre Primfaktoren zu zerlegen. Es ist eine interessante Frage, ob sich auf andere Weise Gruppen mit geheimer Ordnung erzeugen lassen, die man in einem RSA-ähnlichen Schema einsetzen könnte.

9.4 Das Rabin-Verschlüsselungsverfahren

Es ist sehr vorteilhaft, wenn die Sicherheit eines Verschlüsselungsverfahrens auf die Schwierigkeit eines wichtigen mathematischen Problems zurückgeführt werden kann, das auch ohne die kryptographische Anwendung von Bedeutung ist. Ein solches mathematisches Problem wird nämlich von vielen Mathematikern weltweit bearbeitet. Solange es ungelöst ist, bleibt das

Kryptoverfahren sicher. Wird es aber gelöst und wird damit das Kryptosystem unsicher, so wird diese Entdeckung wahrscheinlich schnell bekannt, und man kann entsprechende Vorkehrungen treffen. Bei rein kryptographischen Problemen ist es eher möglich, daß nur ein kleiner Kreis von Personen, z.B. ein Geheimdienst, die Lösung findet und dies den meisten Benutzern verborgen bleibt. Die arglosen Benutzer verwenden dann vielleicht ein unsicheres System in der trügerischen Annahme, es sei sicher.

Die Sicherheit des RSA-Verfahrens hängt zwar mit der Schwierigkeit des Faktorisierungsproblems für natürliche Zahlen zusammen. Es ist aber nicht bekannt, ob das Problem, RSA zu brechen, genauso schwer wie das Faktorisierungsproblem für natürliche Zahlen ist. Anders ist es mit dem Rabin-Verfahren, das nun erklärt wird. Wir werden sehen, daß die Schwierigkeit, das Rabin-Verfahren zu brechen, äquivalent zu einem bestimmten Faktorisierungsproblem ist. Zuerst wird aber die Funktionsweise des Rabin-Verfahrens beschrieben.

9.4.1 Schlüsselerzeugung

Alice wählt zufällig zwei Primzahlen p und q mit $p \equiv q \equiv 3 \bmod 4$. Die Kongruenzbedingung vereinfacht und beschleunigt die Entschlüsselung, wie wir unten sehen werden. Aber auch ohne diese Kongruenzbedingung funktioniert das Verfahren. Alice berechnet $n = pq$. Ihr öffentlicher Schlüssel ist n. Ihr geheimer Schlüssel ist (p, q).

9.4.2 Verschlüsselung

Wie beim RSA-Verfahren ist der Klartextraum die Menge $\{0, \ldots, n-1\}$. Um einen Klartext m zu verschlüsseln, besorgt sich Bob den öffentlichen Schlüssel n von Alice und berechnet

$$c = m^2 \bmod n.$$

Der Schlüsseltext ist c.

Wie das RSA-Verfahren kann auch das Rabin-Verfahren zu einer Art Blockchiffre gemacht werden, indem Buchstabenblöcke als Zahlen in der Menge $\{0, 1, \ldots, n-1\}$ aufgefaßt werden.

9.4.3 Entschlüsselung

Alice berechnet den Klartext m aus dem Schlüsseltext c durch Wurzelziehen. Dazu geht sie folgendermaßen vor. Sie berechnet

$$m_p = c^{(p+1)/4} \bmod p, \quad m_q = c^{(q+1)/4} \bmod q.$$

Dann sind $\pm m_p + p\mathbb{Z}$ die beiden Quadratwurzeln von $c + p\mathbb{Z}$ in $\mathbb{Z}/p\mathbb{Z}$ und $\pm m_q + q\mathbb{Z}$ die beiden Quadratwurzeln von $c + q\mathbb{Z}$ in $\mathbb{Z}/q\mathbb{Z}$ (siehe Übung

3.23.21). Die vier Quadratwurzeln von $c + n\mathbb{Z}$ in $\mathbb{Z}/n\mathbb{Z}$ kann Alice mit Hilfe des chinesischen Restsatzes berechnen. Dies funktioniert nach derselben Methode wie die Entschlüsselung eines RSA-Chiffretextes mit dem chinesischen Restsatz. Alice bestimmt mit dem erweiterten euklidischen Algorithmus Koeffizienten $y_p, y_q \in \mathbb{Z}$ mit

$$y_p p + y_q q = 1.$$

Dann berechnet sie

$$r = (y_p p m_q + y_q q m_p) \bmod n, \quad s = (y_p p m_q - y_q q m_p) \bmod n.$$

Man verifiziert leicht, daß $\pm r$, $\pm s$ die vier Quadratwurzeln von c in der Menge $\{0, 1, \ldots, n-1\}$ sind. Eine dieser Quadratwurzeln muß die Nachricht m sein, es ist aber nicht a priori klar, welche.

Beispiel 9.4.1. Alice verwendet die Primzahlen $p = 11$, $q = 23$. Dann ist $n = 253$. Bob will die Nachricht $m = 158$ verschlüsseln. Er berechnet

$$c = m^2 \bmod n = 170.$$

Alice berechnet $y_p = -2$, $y_q = 1$ wie in Beispiel 9.3.13. Sie ermittelt die Quadratwurzeln

$$m_p = c^{(p+1)/4} \bmod p = c^3 \bmod p = 4,$$
$$m_q = c^{(q+1)/4} \bmod q = c^6 \bmod q = 3.$$

Sie bestimmt

$$r = (y_p p m_q + y_q q m_p) \bmod n = -2 * 11 * 3 + 23 * 4 \bmod n = 26,$$

und

$$s = (y_p p m_q - y_q q m_p) \bmod n = -2 * 11 * 3 - 23 * 4 \bmod n = 95.$$

Die Quadratwurzeln von 170 mod 253 in $\{1, \ldots, 252\}$ sind $26, 95, 158, 227$. Eine dieser Quadratwurzeln ist der Klartext.

Es gibt verschiedene Methoden, aus den vier Quadratwurzeln den richtigen Klartext auszusuchen. Alice kann einfach diejenige Nachricht auswählen, die ihr am wahrscheinlichsten erscheint. Das ist aber nicht besonders treffsicher, und möglicherweise wählt Alice die falsche Nachricht aus. Es ist auch möglich, den Klartexten eine spezielle Struktur zu geben. Dann wird die Quadratwurzel ausgewählt, die die betreffende Struktur aufweist. Man kann z.B. nur Klartexte m zulassen, in denen die letzten 64 Bit gleich den vorletzten 64 Bit sind. Verwendet man aber das Rabin-Verfahren in dieser Art, so kann man die Äquivalenz zum Faktorisierungsproblem nicht mehr beweisen.

9.4.4 Effizienz

Zum Verschlüsseln wird beim Rabin-Verfahren nur eine Quadrierung mod n benötigt. Das ist sogar effizienter als die RSA-Verschlüsselung mit dem Exponenten 3, bei der eine Quadrierung und eine Multiplikation mod n nötig ist. Die Entschlüsselung erfordert je eine modulare Exponentiation mod p und mod q und die Anwendung des chinesischen Restsatzes. Das entspricht dem Aufwand, der zur RSA-Entschlüsselung bei Anwendung des chinesischen Restsatz nötig ist.

9.4.5 Sicherheit gegen Ciphertext-Only-Attacks

Wir zeigen, dass die Sicherheit des Rabin-Verfahrens gegen Ciphertext-Only-Angriffe äquivalent ist zur Schwierigkeit des Faktorisierungsproblems.

Offensichtlich kann jeder, der den öffentlichen Modul n faktorisieren kann, auch das Rabin-Verfahren brechen. Es gilt aber auch die Umkehrung, wie wir unten zeigen werden. Das ist beim RSA-Verfahren nicht bekannt und kann als Stärke des Rabin-Verfahrens angesehen werden.

Angenommen, Oskar kann das Rabin-Verfahren brechen, d.h. er kann zu jedem Quadrat $c + n\mathbb{Z}$ eine Quadratwurzel $m + n\mathbb{Z}$ bestimmen, sagen wir mit einem Algorithmus R. Der Algorithmus R liefert bei Eingabe von $c \in \{0, 1, \ldots, n-1\}$ eine ganze Zahl $m = R(c) \in \{0, 1, \ldots, n-1\}$ für die die Restklasse $m + n\mathbb{Z}$ eine Quadratwurzel von $c + n\mathbb{Z}$ ist.

Um n zu faktorisieren, wählt Oskar zufällig und gleichverteilt eine Zahl $x \in \{1, \ldots, n-1\}$. Wenn $\gcd(x, n) \neq 1$ ist, hat Oskar den Modul n faktorisiert. Andernfalls berechnet er

$$c = x^2 \bmod n \text{ und } m = R(c).$$

Die Restklasse $m + n\mathbb{Z}$ ist eine der Quadratwurzeln von $c + n\mathbb{Z}$. Sie muß nicht mit $x + n\mathbb{Z}$ übereinstimmen. Aber m erfüllt eines der folgenden Bedingungspaare

$$m \equiv x \bmod p \text{ und } m \equiv x \bmod q, \tag{9.5}$$

$$m \equiv -x \bmod p \text{ und } m \equiv -x \bmod q, \tag{9.6}$$

$$m \equiv x \bmod p \text{ und } m \equiv -x \bmod q, \tag{9.7}$$

$$m \equiv -x \bmod p \text{ und } m \equiv x \bmod q. \tag{9.8}$$

Im Fall (9.5) ist $m = x$ und $\gcd(m - x, n) = n$. Im Fall (9.6) ist $m = n - x$ und $\gcd(m - x, n) = 1$. Im Fall (9.7) ist $\gcd(m - x, n) = p$. Im Fall (9.8) ist $\gcd(m - x, n) = q$. Da x zufällig gewählt wurde, tritt jeder dieser Fälle mit derselben Wahrscheinlichkeit auf. Daher wird bei einem Durchlauf des Verfahrens die Zahl n mit der Wahrscheinlichkeit $\geq 1/2$ faktorisiert, und bei k Durchläufen des Verfahrens wird n mit Wahrscheinlichkeit $\geq 1 - (1/2)^k$ zerlegt.

Beispiel 9.4.2. Wie in Beispiel 9.4.1 ist $n = 253$. Angenommen, Oskar kann mit dem Algorithmus R Quadratwurzeln modulo 253 bestimmen. Er wählt $x = 17$ und stellt fest, daß $\gcd(17, 253) = 1$ ist. Als nächstes bestimmt Oskar $c = 17^2 \bmod 253 = 36$. Die Quadratwurzeln von 36 mod 253 sind $6, 17, 236, 247$. Es ist $\gcd(6 - 17, n) = 11$ und $\gcd(247 - 17, 253) = 23$. Wenn R also eine dieser beiden Quadratwurzeln berechnet, hat Oskar die Faktorisierung von n gefunden.

Im beschriebenen Faktorisierungsverfahren wurde vorausgesetzt, daß der Klartextraum aus allen Zahlen in der Menge $\{0, 1, \ldots, n-1\}$ besteht. Die Argumentation ist nicht mehr richtig, wenn der Klartextraum wie in Abschnitt 9.4.3 auf Klartexte mit bestimmter Struktur eingeschränkt wird. Dann kann der Algorithmus R nämlich nur Verschlüsselungen dieser speziellen Klartexte entschlüsseln. Oskar muß dann x mit dieser speziellen Struktur wählen. Die anderen Quadratwurzeln von $x^2 + n\mathbb{Z}$ haben mit hoher Wahrscheinlichkeit nicht die spezielle Struktur. Darum liefert R auch nur die Quadratwurzel x, die Oskar ohnehin schon kannte.

9.4.6 Eine Chosen-Ciphertext-Attacke

Wir haben gesehen, daß Oskar natürliche Zahlen faktorisieren kann, wenn er das Rabin-Verfahren brechen kann. Man kann dies als Sicherheitsvorteil ansehen. Andererseits erlaubt dieser Umstand aber auch eine Chosen-Ciphertext-Attacke.

Angenommen, Oskar kann einen selbst gewählten Schlüsseltext entschlüsseln. Dann wählt er $x \in \{1, \ldots, n-1\}$ zufällig und berechnet den Schlüsseltext $c = x^2 \bmod n$. Diesen Schlüsseltext entschlüsselt er anschließend. Wie wir in Abschnitt 9.4.5 gesehen haben, kann Oskar danach den Modul n mit Wahrscheinlichkeit $1/2$ faktorisieren.

Um die Chosen-Ciphertext-Attacke zu verhindern, kann man, wie in 9.4.3 beschrieben, den Klartextraum auf Klartexte mit bestimmter Struktur beschränken. Wie in Abschnitt 9.4.5 gezeigt, geht dann aber die Äquivalenz von Rabin-Entschlüsselung und Faktorisierung verloren.

Einige Angriffe, die beim RSA-Verfahren möglich sind, kann man auch auf das Rabin-Verfahren anwenden. Dies gilt insbesondere für die Low-Exponent-Attacke, die Attacke bei zu kleinem Klartextraum und die Ausnutzung der Multiplikativität. Die Details werden in den Übungen behandelt.

9.4.7 Sichere Verwendung

Will man das Rabin-Verfahren sicher machen, muß man es so verwenden, wie das in Abschnitt 9.3.11 beschrieben wurde.

9.5 Diffie-Hellman-Schlüsselaustausch

In diesem Abschnitt beschreiben wir das Verfahren von Diffie und Hellman, geheime Schlüssel über unsichere Leitungen auszutauschen. Das Diffie-Hellman-Verfahren ist zwar kein Public-Key-Verschlüsselungsverfahren, aber es ist die Grundlage des ElGamal-Verfahrens, das als nächstes behandelt wird.

Die Situation ist folgende: Alice und Bob wollen mit einem symmetrischen Verschlüsselungsverfahren kommunizieren. Sie sind über eine unsichere Leitung verbunden und haben noch keinen Schlüssel ausgetauscht. Das Diffie-Hellman-Verfahren erlaubt es Alice und Bob, einen geheimen Schlüssel über die öffentliche, nicht gesicherte Leitung auszutauschen, ohne daß ein Zuhörer den Schlüssel erfährt.

Die Sicherheit des Diffie-Hellman-Verfahrens beruht nicht auf dem Faktorisierungsproblem für natürliche Zahlen, sondern auf einem anderen zahlentheoretischen Problem, das wir hier kurz vorstellen.

9.5.1 Diskrete Logarithmen

Sei p eine Primzahl. Wir wissen, daß die prime Restklassengruppe mod p zyklisch ist von der Ordnung $p-1$. Sei g eine Primitivwurzel mod p. Dann gibt es für jede Zahl $A \in \{1, 2, \ldots, p-1\}$ einen Exponenten $a \in \{0, 1, 2, \ldots, p-2\}$ mit

$$A \equiv g^a \bmod p.$$

Dieser Exponent a heißt *diskreter Logarithmus* von A zur Basis g. Wir schreiben $a = \log_g A$. Die Berechnung solcher diskreter Logarithmen gilt als schwieriges Problem. Bis heute sind keine effizienten Algorithmen bekannt, die dieses Problem lösen.

Beispiel 9.5.1. Sei $p = 13$. Eine Primitivwurzel modulo 13 ist 2. In der folgenden Tabelle finden sich die diskreten Logarithmen aller Elemente der Menge $\{1, 2, \ldots, 12\}$ zur Basis 2.

A	1	2	3	4	5	6	7	8	9	10	11	12
$\log_2 A$	0	1	4	2	9	5	11	3	8	10	7	6

Diskrete Logarithmen kann man auch in anderen zyklischen Gruppen definieren. Sei G eine endliche zyklische Gruppe der Ordnung n mit Erzeuger g und sei A ein Gruppenelement. Dann gibt es einen Exponenten $a \in \{0, 1, \ldots, n-1\}$ mit

$$A = g^a.$$

Dieser Exponent a heißt *diskreter Logarithmus* von A zur Basis g. Wir werden sehen, daß sich das Diffie-Hellman-Verfahren in allen Gruppen G sicher implementieren läßt, in denen die Berechnung diskreter Logarithmen schwer und die Ausführung der Gruppenoperationen leicht ist.

Beispiel 9.5.2. Betrachte die additive Gruppe $\mathbb{Z}/n\mathbb{Z}$ für eine natürliche Zahl n. Sie ist zyklisch von der Ordnung n. Ein Erzeuger dieser Gruppe ist $1+n\mathbb{Z}$. Sei $A \in \{0, 1, \ldots, n-1\}$. Der diskrete Logarithmus a von $A + n\mathbb{Z}$ zur Basis $1 + n\mathbb{Z}$ genügt der Kongruenz

$$A \equiv a \bmod n.$$

Also ist $a = A$. Die anderen Erzeuger von $\mathbb{Z}/n\mathbb{Z}$ sind die Restklassen $g+n\mathbb{Z}$ mit $\gcd(g, n) = 1$. Der diskrete Logarithmus a von $A+n\mathbb{Z}$ zur Basis $g+n\mathbb{Z}$ genügt der Kongruenz

$$A \equiv ga \bmod n.$$

Diese Kongruenz kann man mit dem erweiterten euklidischen Algorithmus lösen. In $\mathbb{Z}/n\mathbb{Z}$ gibt es also ein effizientes Verfahren zur Berechnung diskreter Logarithmen. Daher ist diese Gruppe für die Implementierung des Diffie-Hellman-Verfahrens ungeeignet.

9.5.2 Schlüsselaustausch

Das Diffie-Hellman-Verfahren funktioniert folgendermaßen: Bob und Alice wollen sich auf einen Schlüssel K einigen, haben aber nur eine Kommunikationsverbindung zur Verfügung, die abgehört werden kann. Bob und Alice einigen sich auf eine Primzahl p und eine Primitivwurzel g mod p mit $2 \leq g \leq p-2$ (siehe Abschnitt 3.22). Die Primzahl p und die Primitivwurzel g können öffentlich bekannt sein. Alice wählt eine natürliche Zahl $a \in \{0, 1, \ldots, p-2\}$ zufällig. Sie berechnet

$$A = g^a \bmod p$$

und schickt das Ergebnis A an Bob. Aber sie hält den Exponenten a geheim. Bob wählt eine natürliche Zahl $b \in \{0, 1, \ldots, p-2\}$ zufällig. Er berechnet

$$B = g^b \bmod p$$

und schickt das Ergebnis an Alice. Den Exponenten b hält er geheim. Um den geheimen Schlüssel zu berechnen, berechnet Alice

$$B^a \bmod p = g^{ab} \bmod p,$$

und Bob berechnet

$$A^b \bmod p = g^{ab} \bmod p.$$

Der gemeinsame Schlüssel ist

$$K = g^{ab} \bmod p.$$

Beispiel 9.5.3. Sei $p = 17$, $g = 3$. Alice wählt den Exponenten $a = 7$, berechnet $g^a \bmod p = 11$ und schickt das Ergebnis $A = 11$ an Bob. Bob wählt den Exponenten $b = 4$, berechnet $g^b \bmod p = 13$ und schickt das Ergebnis $B = 13$ an Alice. Alice berechnet $B^a \bmod p = 4$. Bob berechnet $A^b \bmod p = 4$. Also ist der ausgetauschte Schlüssel 4.

9.5.3 Sicherheit

Ein Lauscher an der unsicheren Leitung erfährt die Zahlen p, g, A und B. Er erfährt aber nicht die diskreten Logarithmen a von A und b von B zur Basis g. Er muß den geheimen Schlüssel $K = g^{ab} \bmod p$ berechnen. Das nennt man das *Diffie-Hellman-Problem*. Wer diskrete Logarithmen mod p berechnen kann, ist in der Lage, das Diffie-Hellman-Problem zu lösen. Das ist auch die einzige bekannte allgemein anwendbare Methode, um das Diffie-Hellman-Verfahren zu brechen. Es ist aber nicht bewiesen, daß das tatsächlich die einzige Methode ist, ob also jemand, der das Diffie-Hellman-Problem effizient lösen kann, auch diskrete Logarithmen effizient berechnen kann.

Solange das Diffie-Hellman-Problem nicht in vertretbarer Zeit lösbar ist, solange ist es für einen Angreifer unmöglich, den geheimen Schlüssel zu bestimmen. Soll es aber für den Angreifer unmöglich sein, aus der öffentlich verfügbaren Information irgendwelche Informationen über den transportierten Schlüssel zu gewinnen, muß das *Decision-Diffie-Hellman-Problem* unangreifbar sein (siehe [14]). Dieses Problem besteht daraus, bei gegebenen $g^a \bmod p$, $g^b \bmod p$ und $g^c \bmod p$ zu entscheiden, ob $g^c = g^{ab}$ ist.

Die Berechnung des geheimen Schlüssels ist aber nicht der einzig mögliche Angriff auf das Protokoll. Ein wichtiges Problem besteht darin, daß Alice und Bob nicht sicher sein können, daß die Nachricht tatsächlich vom jeweils anderen kommt. Ein Angreifer Oskar kann z.B. die *Man-In-The-Middle-Attacke* verwenden. Er tauscht sowohl mit Alice als auch mit Bob einen geheimen Schlüssel aus. Alice glaubt, der Schlüssel komme von Bob und Bob glaubt, der Schlüssel komme von Alice. Alle Nachrichten, die Alice dann an Bob sendet, fängt Oskar ab, entschlüsselt sie und verschlüsselt sie mit dem zweiten Schlüssel und sendet sie an Bob.

9.5.4 Andere Gruppen

Man kann das Diffie-Hellman-Verfahren in allen zyklischen Gruppen sicher implementieren, in denen die Gruppenoperationen effizient realisierbar sind und in denen das Diffie-Hellman-Problem schwer zu lösen ist. Insbesondere muß es schwer sein, diskrete Logarithmen in G zu berechnen. In Kapitel 14 werden wir Beispiele für solche Gruppen noch behandeln. Hier schildern wir nur das Prinzip.

Alice und Bob einigen sich auf eine endliche zyklische Gruppe G und einen Erzeuger g von G. Sei n die Ordnung von G. Alice wählt eine natürliche Zahl $a \in \{1, 2, \ldots, n-1\}$ zufällig. Sie berechnet

$$A = g^a$$

und schickt das Ergebnis A an Bob. Bob wählt eine natürliche Zahl $b \in \{1, 2, \ldots, n-1\}$ zufällig. Er berechnet

$$B = g^b$$

und schickt das Ergebnis an Alice. Alice berechnet nun

$$B^a = g^{ab}$$

und Bob berechnet

$$A^b = g^{ab}.$$

Der gemeinsame Schlüssel ist

$$K = g^{ab}.$$

9.6 Das ElGamal-Verschlüsselungsverfahren

Das ElGamal-Verfahren hängt eng mit dem Diffie-Hellman-Schlüsselaustausch zusammen. Seine Sicherheit beruht auf der Schwierigkeit, das Diffie-Hellman-Problem in $(\mathbb{Z}/p\mathbb{Z})^*$ zu lösen.

9.6.1 Schlüsselerzeugung

Alice wählt eine Primzahl p und, wie in Abschnitt 3.22 beschrieben, eine Primitivwurzel g mod p. Dann wählt sie zufällig und gleichverteilt einen Exponenten $a \in \{0, \ldots, p-2\}$ und berechnet

$$A = g^a \bmod p.$$

Der öffentliche Schlüssel von Alice ist (p, g, A). Der geheime Schlüssel von Alice ist der Exponent a. Man beachte, daß Alice im Diffie-Hellman-Verfahren den Wert A an Bob schickt. Im ElGamal-Verfahren liegt also der Schlüsselanteil von Alice ein für allemal fest und ist öffentlich.

9.6.2 Verschlüsselung

Der Klartextraum ist die Menge $\{0, 1, \ldots, p-1\}$. Um einen Klartext m zu verschlüsseln, besorgt sich Bob den authentischen öffentlichen Schlüssel (p, g, A) von Alice. Er wählt eine Zufallszahl $b \in \{1, \ldots, p-2\}$ und berechnet

$$B = g^b \bmod p.$$

Die Zahl B ist der Schlüsselanteil von Bob aus dem Diffie-Hellman-Verfahren. Bob bestimmt

$$c = A^b m \bmod p.$$

Bob multipliziert also den Klartext m mit dem Diffie-Hellman-Schlüssel $A^b \bmod p = g^{ab} \bmod p$. Der Schlüsseltext ist (B, c).

9.6.3 Entschlüsselung

Alice hat von Bob den Schlüsseltext (B, c) erhalten, und sie kennt ihren geheimen Schlüssel a. Um den Klartext m zu rekonstruieren, bestimmt Alice den Exponenten $x = p - 1 - a$. Weil $1 \leq a \leq p - 2$ ist, gilt $1 \leq x \leq p - 2$. Dann berechnet Alice $B^x c \bmod p$. Dies ist der ursprüngliche Klartext, wie die folgende Rechnung zeigt:

$$B^x c \equiv g^{b(p-1-a)} A^b m \equiv (g^{p-1})^b (g^a)^{-b} A^b m \equiv A^{-b} A^b m \equiv m \bmod p.$$

Beispiel 9.6.1. Alice wählt $p = 23$, $g = 7$, $a = 6$ und berechnet $A = g^a \bmod p = 4$. Ihr öffentlicher Schlüssel ist dann $(p = 23, g = 7, A = 4)$. Ihr geheimer Schlüssel ist $a = 6$. Bob will $m = 7$ verschlüsseln. Er wählt $b = 3$, berechnet $B = g^b \bmod p = 21$ und $c = A^b m \bmod p = 11$. Der Schlüsseltext ist $(B, c) = (21, 11)$. Alice entschlüsselt $B^{p-1-6} c \bmod p = 7 = m$.

9.6.4 Effizienz

Die ElGamal-Entschlüsselung erfordert eine modulare Exponentiation wie das RSA-Verfahren. Die ElGamal-Entschlüsselung kann jedoch nicht mit dem chinesischen Restsatz beschleunigt werden.

Die Verschlüsselung mit dem ElGamal-Verfahren erfordert zwei modulare Exponentiationen, nämlich die Berechnung von $A^b \bmod p$ und $B = g^b \bmod p$. Im Gegensatz dazu erfordert die Verschlüsselung beim RSA-Verfahren nur eine Exponentiation modulo n. Die Primzahl p im ElGamal-Verfahren und der RSA-Modul n sind von derselben Größenordnung. Daher sind die einzelnen modularen Exponentiationen gleich teuer. Beim ElGamal-Verfahren kann Bob die Werte $A^b \bmod p$ und $B = g^b \bmod p$ aber auf Vorrat vorberechnen, weil sie unabhängig von der zu verschlüsselnden Nachricht sind. Die vorberechneten Werte müssen nur in einer sicheren Umgebung, etwa auf einer Chipkarte, gespeichert werden. Dann benötigt die aktuelle Verschlüsselung nur eine Multiplikation modulo p. Das ist effizienter als die RSA-Verschlüsselung.

Beispiel 9.6.2. Wie in Beispiel 9.6.1 ist der öffentliche Schlüssel von Alice $(p = 23, g = 7, A = 4)$. Ihr geheimer Schlüssel ist $a = 6$. Bevor Bob in die Situation kommt, eine Nachricht wirklich verschlüsseln zu müssen, wählt er $b = 3$ und berechnet $B = g^b \bmod p = 21$ und $K = A^b \bmod p = 18$. Später will Bob $m = 7$ verschlüsseln. Dann berechnet er einfach $c = K * m \bmod 23 = 11$. Der Schlüsseltext ist $(B, c) = (21, 11)$. Wieder entschlüsselt Alice $B^{p-1-6} c \bmod p = 7 = m$.

Ein Effizienznachteil des ElGamal-Verfahrens besteht darin, daß der Schlüsseltext doppelt so lang ist wie der Klartext. Das nennt man *Nachrichtenexpansion*. Dafür ist das ElGamal-Verfahren aber ein randomisiertes Verschlüsselungsverfahren. Wir werden darauf in Abschnitt 9.6.7 eingehen.

Die Länge der öffentlichen Schlüssel im ElGamal-Verfahren kann verkürzt werden, wenn im gesamten System dieselbe Primzahl p und dieselbe Primitivwurzel $g \bmod p$ verwendet wird. Dies kann aber auch ein Sicherheitsrisiko sein, wenn sich herausstellt, daß für die verwendete Primzahl die Berechnung diskreter Logarithmen einfach ist.

9.6.5 ElGamal und Diffie-Hellman

Wer diskrete Logarithmen mod p berechnen kann, ist auch in der Lage, das ElGamal-Verfahren zu brechen. Er kann nämlich aus A den geheimen Exponenten a ermitteln und dann $m = B^{p-1-a}c \bmod p$ berechnen. Es ist aber nicht bekannt, ob jemand, der das ElGamal-Verfahren brechen kann, auch diskrete Logarithmen mod p bestimmen kann.

Das ElGamal-Verfahren ist aber genauso schwer zu brechen wie das Diffie-Hellman-Verfahren. Das kann man folgendermaßen einsehen. Angenommen, Oskar kann das Diffie-Hellmann-Problem lösen, d.h. aus p, g, A und B den geheimen Schlüssel $g^{ab} \bmod p$ berechnen. Oskar möchte einen ElGamal-Schlüsseltext (B, c) entschlüsseln. Er kennt auch den entsprechenden öffentlichen Schlüssel (p, g, A). Weil er Diffie-Hellman brechen kann, kann er $K = g^{ab} \bmod p$ bestimmen und damit den Klartext $K^{-1}c \bmod p$ rekonstruieren. Sei umgekehrt angenommen, daß Oskar das ElGamal-Verfahren brechen, also aus der Kenntnis von p, g, A, B und c die Nachricht m ermitteln kann, und zwar für jede beliebige Nachricht m. Wenn er den Schlüssel g^{ab} aus p, g, A, B ermitteln will, wendet er das Entschlüsselungsverfahren für ElGamal mit $p, g, A, B, c = 1$ an und erhält eine Nachricht m. Er weiß, daß $1 = g^{ab}m \bmod p$ ist. Daher kann er $g^{ab} \equiv m^{-1} \bmod p$ berechnen und hat den Schlüssel $g^{ab} \bmod p$ gefunden.

9.6.6 Parameterwahl

Um die Anwendung der heute bekannten Verfahren zur Berechnung diskreter Logarithmen zu verhindern, muß die Primzahl wenigstens 512 Bits, besser 768 oder sogar 1024 Bits lang sein. Außerdem muß die Primzahl p einige Zusatzbedingungen erfüllen, die es verhindern, daß diskrete Logarithmen in $(\mathbb{Z}/p\mathbb{Z})^*$ mit bekannten Methoden (Pohlig-Hellman-Algorithmus, Number-Field-Sieve) leicht berechnet werden können. Da man aber nicht alle möglichen Algorithmen zur Lösung des Diffie-Hellman-Problems vorhersagen kann, erscheint es am sichersten, die Primzahl p zufällig und gleichverteilt zu wählen.

Bei jeder neuen ElGamal-Verschlüsselung muß Bob einen neuen Exponenten b wählen. Wählt Bob nämlich zweimal dasselbe b und berechnet damit aus den Klartexten m und m' die Schlüsseltexte

$$c = A^b m \bmod p, \quad c' = A^b m' \bmod p,$$

so gilt

$$c'c^{-1} \equiv m'm^{-1} \bmod p.$$

Jeder Angreifer, der den Klartext m kennt, kann dann den Klartext m' gemäß der Formel

$$m' = c'c^{-1}m \bmod p$$

berechnen.

9.6.7 ElGamal ist randomisiert

Der Verschlüsselungsprozeß beim ElGamal-Verfahren wird durch die zufällige Wahl des Exponenten b randomisiert. Ein Klartext wird nämlich zum Schlüsseltext $(B = g^b \bmod p, c = A^b m \bmod p)$ verschlüsselt, wobei b eine mit Gleichverteilung gewählte Zufallszahl in $\{0, \ldots, p-1\}$ ist. Der Schlüsseltext (B, c) ist also zufällig und gleichverteilt in $\{1, \ldots, p-1\}^2$, vorausgesetzt, A ist eine Primitivwurzel mod p, d.h. $\gcd(a, p-1) = 1$. Aufgrund dieser Randomisierung ist das ElGamal-Verfahren semantisch sicher, solange das Decisional-Diffie-Hellman-Problem unangreifbar ist. Das ElGamal-Verschlüsselungsverfahren ist aber *malleable*, d.h. ein Angreifer kann den Schlüsseltext so ändern, dass sich der Klartext in kontrollierter Weise ändert. Ist nämlich (B, c) die Verschlüsselung von m, dann ist $(B, cx \pmod p)$ die Verschlüsselung von $mx \pmod p$ für alle $x \in \{0, \ldots, p-1\}$. Eine Variante des ElGamal-Verschlüsselungsverfahrens, die sicher gegen Chosen-Ciphertext-Attacks ist, solange das Decisional-Diffie-Hellman-Problem unangreifbar ist, wurde in [22] vorgeschlagen. Dieses Verfahren ist aber deutlich ineffizienter als das ElGamal-Verfahren.

9.6.8 Verallgemeinerung

Der wichtigste Vorteil des ElGamal-Verfahrens besteht darin, daß es sich nicht nur in der primen Restklassengruppe modulo einer Primzahl, sondern in jeder anderen zyklischen Gruppe verwenden läßt. Es ist nur erforderlich, daß sich die Schlüsselerzeugung, Verschlüsselung und Entschlüsselung effizient durchführen lassen und daß das entsprechende Diffie-Hellman-Problem schwer zu lösen ist. Insbesondere muß die Berechnung diskreter Logarithmen in G schwer sein, weil sonst das Diffie-Hellman-Problem leicht zu lösen ist.

Es ist sehr wichtig, daß das ElGamal-Verfahrens verallgemeinert werden kann, weil es immer möglich ist, daß jemand einen effizienten Algorithmus zur Berechnung diskreter Logarithmen in $(\mathbb{Z}/p\mathbb{Z})^*$ findet. Dann kann man das ElGamal-Verfahren in $(\mathbb{Z}/p\mathbb{Z})^*$ nicht mehr benutzen, weil es unsicher geworden ist. In anderen Gruppen kann das ElGamal-Verfahren aber immer noch sicher sein, und man kann dann diese Gruppen verwenden.

Folgende Gruppen sind z.B. zur Implementierung des ElGamal-Verfahrens geeignet:

1. Die Punktgruppe einer elliptischen Kurve über einem endlichen Körper.
2. Die Jakobische Varietät hyperelliptischer Kurven über endlichen Körpern.
3. Die Klassengruppe imaginär-quadratischer Ordnungen.

9.7 Übungen

Übung 9.7.1. Man zeige, daß man im RSA-Verfahren den Entschlüsselungsexponenten d auch so wählen kann, daß $de \equiv 1 \bmod \operatorname{lcm}(p-1, q-1)$ ist.

Übung 9.7.2. Bestimmen Sie alle für den RSA-Modul $n = 437$ möglichen Verschlüsselungsexponenten. Geben Sie eine Formel für die Anzahl der für einen RSA-Modul n möglichen Verschlüsselungsexponenten an.

Übung 9.7.3. Erzeugen Sie zwei 8-Bit-Primzahlen p und q so, daß $n = pq$ eine 16-Bit-Zahl ist und der öffentliche RSA-Schlüssel $e = 5$ verwendet werden kann. Berechnen Sie den privaten Schlüssel d zum öffentlichen Schlüssel $e = 5$. Verschlüsseln Sie den String 110100110110111 mit dem öffentlichen Exponenten 5.

Übung 9.7.4. Alice verschlüsselt die Nachricht m mit Bobs öffentlichem RSA-Schlüssel $(899, 11)$. Der Schlüsseltext ist 468. Bestimmen Sie den Klartext.

Übung 9.7.5. Entwerfen Sie einen polynomiellen Algorithmus, der bei Eingabe von natürlichen Zahlen c und e entscheidet, ob c eine e-te Potenz ist und wenn dies der Fall ist, auch noch die e-te Wurzel von c berechnet. Beweisen Sie, daß der Algorithmus tatsächlich polynomielle Laufzeit hat.

Übung 9.7.6. Implementieren Sie den Algorithmus aus Übung 9.7.5.

Übung 9.7.7. Wieviele Operationen erfordert die RSA-Verschlüsselung mit Verschlüsselungsexponent $e = 2^{16} + 1$?

Übung 9.7.8. Die Nachricht m wird mit den öffentlichen RSA-Schlüsseln $(391, 3)$, $(55, 3)$ und $(87, 3)$ verschlüsselt. Die Schlüsseltexte sind 208, 38 und 32. Verwenden Sie die Low-Exponent-Attacke, um m zu finden.

Übung 9.7.9 (Common-Modulus-Attacke). Wenn man mit dem RSA-Verfahren eine Nachricht m zweimal verschlüsselt, und zwar mit den öffentlichen Schlüsseln (n, e) und (n, f), und wenn $\gcd(e, f) = 1$ gilt, dann kann man den Klartext m aus den beiden Schlüsseltexten $c_e = m^e \bmod n$ und $c_f = m^f \bmod n$ berechnen. Wie geht das?

Übung 9.7.10. Die Nachricht m wird mit den öffentlichen RSA-Schlüsseln $(493, 3)$ und $(493, 5)$ verschlüsselt. Die Schlüsseltexte sind 293 und 421. Verwenden Sie die Common-Modulus-Attacke, um m zu finden.

Übung 9.7.11. Sei $n = 1591$. Der öffentliche RSA-Schlüssel von Alice ist (n, e) mit minimalem e. Alice erhält die verschlüsselte Nachricht $c = 1292$. Entschlüsseln Sie diese Nachricht mit Hilfe des chinesischen Restsatzes.

Übung 9.7.12. Angenommen, der RSA-Modul ist $n = 493$, der Verschlüsselungsexponent ist $e = 11$, und der Entschlüsselungsexponent ist $d = 163$. Verwenden Sie die Methode aus Abschnitt 9.3.4, um n zu faktorisieren.

Übung 9.7.13 (Cycling-Attacke). Sei (n, e) ein öffentlicher RSA-Schlüssel. Für einen Klartext $m \in \{0, 1, \ldots, n-1\}$ sei $c = m^e \bmod n$ der zugehörige Schlüsseltext. Zeigen Sie, daß es eine natürliche Zahl k gibt mit

$$m^{e^k} \equiv m \bmod n.$$

Beweisen Sie für ein solches k:

$$c^{e^{k-1}} \equiv m \bmod n.$$

Ist dies eine Bedrohung für RSA?

Übung 9.7.14. Sei $n = 493$ und $e = 3$. Bestimmen Sie den kleinsten Wert von k, für den die Cycling-Attacke aus Übung 9.7.13 funktioniert.

Übung 9.7.15. Bob verschlüsselt Nachrichten an Alice mit dem Rabin-Verfahren. Er verwendet dieselben Parameter wie in Beispiel 9.4.1. Die Klartexte sind Blöcke in $\{0, 1\}^8$, in denen die ersten beiden Bits mit den letzten beiden Bits übereinstimmen. Kann Alice alle Klartexte eindeutig entschlüsseln?

Übung 9.7.16. Sei $n = 713$ ein öffentlicher Rabin-Schlüssel und sei $c = 289$ ein Schlüsseltext, den man durch Rabin-Verschlüsselung mit diesem Modul erhalten hat. Bestimmen Sie alle möglichen Klartexte.

Übung 9.7.17. Übertragen Sie die Low-Exponent-Attacke und die Multiplikativitäts-Attacke, die beim RSA-Verfahren besprochen wurden, auf das Rabin-Verfahren und schlagen Sie entsprechende Gegenmaßnahmen vor.

Übung 9.7.18. Wie kann der Rabin-Modul $n = 713$ mit zwei möglichen Klartexten aus Übung 9.7.16 faktorisiert werden?

Übung 9.7.19. Wie kann man aus zwei ElGamal-Schlüsseltexten einen dritten ElGamal-Schlüsseltext machen, ohne den geheimen ElGamal-Schlüssel zu kennen? Wie kann man diesen Angriff verhindern?

Übung 9.7.20. Alice erhält den ElGamal-Chiffretext $(B = 30, c = 7)$. Ihr öffentlicher Schlüssel ist $(p = 43, g = 3)$. Bestimmen Sie den zugehörigen Klartext.

Übung 9.7.21. Der öffentliche ElGamal-Schlüssel von Bob sei $p = 53$, $g = 2$, $A = 30$. Alice erzeugt damit den Schlüsseltext $(24, 37)$. Wie lautet der Klartext?

10. Faktorisierung

Wie wir gezeigt haben, hängt die Sicherheit des RSA-Verfahrens und die Sicherheit des Rabin-Verfahrens eng mit der Schwierigkeit zusammen, natürliche Zahlen in ihre Primfaktoren zu zerlegen. Es ist nicht bekannt, ob das Faktorisierungsproblem für natürliche Zahlen leicht oder schwer ist. In den letzten Jahrzehnten wurden immer effizientere Faktorisierungsmethoden entwickelt. Trotzdem ist RSA heute immer noch sicher, wenn man die Parameter richtig wählt. Es könnte aber sein, daß schon bald ein so effizienter Faktorisierungsalgorithmus gefunden wird und RSA nicht mehr sicher ist. Daher ist es wichtig, kryptographische Systeme so zu implementieren, daß die grundlegenden Verfahren leicht ersetzt werden können.

In diesem Kapitel beschreiben wir einige Faktorisierungsalgorithmen. Dabei ist n immer eine natürliche Zahl, von der schon bekannt ist, daß sie zusammengesetzt ist. Das kann man z.B. mit dem Fermat-Test oder mit dem Miller-Rabin-Test feststellen (siehe Abschnitt 8.2 und Abschnitt 8.4). Diese Tests bestimmen aber keinen Teiler von n. Wir skizzieren die Faktorisierungsverfahren nur. Für weitere Details sei auf [46] und [16] verwiesen. Die beschriebenen Algorithmen sind in der Bibliothek *LiDIA*[50] implementiert.

10.1 Probedivison

Um die kleinen Primfaktoren von n zu finden, berechnet man die Liste aller Primzahlen unter einer festen Schranke B. Dafür kann man das Sieb des Erathostenes verwenden (siehe [4]). Dann bestimmt man für jede Primzahl p in dieser Liste den maximalen Exponenten $e(p)$, für den p die Zahl n teilt. Eine typische Schranke ist $B = 10^6$.

Beispiel 10.1.1. Wir wollen die Zahl $n = 3^{21}+1 = 10460353204$ faktorisieren. Probedivision aller Primzahlen bis 50 ergibt die Faktoren 2^2, 7^2 und 43. Dividiert man die Zahl n durch diese Faktoren, so erhält man $m = 1241143$. Es ist $2^{m-1} \equiv 793958 \bmod m$. Nach dem kleinen Satz von Fermat ist m also zusammengesetzt.

10.2 Die $p-1$-Methode

Es gibt Faktorisierungsmethoden, die Zahlen mit bestimmten Eigenschaften besonders gut zerlegen können. Solche Zahlen müssen als RSA- oder Rabin-Moduln vermieden werden. Als Beispiel für einen solchen Faktorisierungsalgorithmus beschreiben wir die $(p-1)$-Methode von John Pollard.

Das $(p-1)$-Verfahren ist für zusammengesetzte Zahlen n geeignet, die einen Primfaktor p haben, für den $p-1$ nur kleine Primfaktoren hat. Man kann dann nämlich ohne p zu kennen ein Vielfaches k von $p-1$ bestimmen. Wie das geht, beschreiben wir unten. Für dieses Vielfache k gilt nach dem kleinen Satz von Fermat

$$a^k \equiv 1 \mod p$$

für alle ganzen Zahlen a, die nicht durch p teilbar sind. Das bedeutet, daß p ein Teiler von a^k-1 ist. Ist a^k-1 nicht durch n teilbar, so ist $\gcd(a^k-1,n)$ ein echter Teiler von n. Damit ist n faktorisiert.

Der Algorithmus von Pollard verwendet als Kandidaten für k die Produkte aller Primzahlpotenzen, die nicht größer als eine Schranke B sind, also

$$k = \prod_{q \in \mathcal{PP}, q^e \le B} q^e.$$

Wenn die Primzahlpotenzen, die $p-1$ teilen, alle kleiner als B sind, dann ist k ein Vielfaches von $p-1$. Der Algorithmus berechnet $g = \gcd(a^k-1,n)$ für eine geeignete Basis a. Wird dabei kein Teiler von n gefunden, so wird ein neues B verwendet.

Beispiel 10.2.1. Die Zahl $n = 1241143$ ist in Beispiel 10.1.1 übriggeblieben. Sie muß noch faktorisiert werden. Wir setzen $B = 13$. Dann ist $k = 8*9*5*7*11*13$ und

$$\gcd(2^k-1,n) = 547.$$

Also ist $p = 547$ ein Teiler von n. Der Kofaktor ist $q = 2269$. Sowohl 547 als auch 2269 sind Primzahlen.

Eine Weiterentwicklung der $(p-1)$-Methode ist die Faktorisierungsmethode mit elliptischen Kurven (ECM). Sie funktioniert für beliebige zusammengesetzte Zahlen n.

10.3 Das Quadratische Sieb

Einer der effizientesten Faktorisierungsalgorithmen ist das Quadratische Sieb (QS), das in diesem Abschnitt beschrieben wird.

10.3.1 Das Prinzip

Wieder soll die zusammengesetzte Zahl n faktorisiert werden. Wir beschreiben, wie man einen echten Teiler von n findet. Wenn n wie im RSA-Verfahren Produkt zweier Primzahlen ist, dann ist damit die Primfaktorzerlegung von n gefunden. Andernfalls müssen die gefundenen Faktoren ihrerseits faktorisiert werden.

Im Quadratischen Sieb werden ganze Zahlen x und y bestimmt, für die

$$x^2 \equiv y^2 \mod n \tag{10.1}$$

und

$$x \not\equiv \pm y \mod n \tag{10.2}$$

gilt. Dann ist n nämlich ein Teiler von $x^2 - y^2 = (x - y)(x + y)$, aber weder von $x - y$ noch von $x + y$. Also ist $g = \gcd(x - y, n)$ ein echter Teiler von n und das Berechnungsziel ist erreicht.

Beispiel 10.3.1. Sei $n = 7429$, $x = 227$, $y = 210$. Dann ist $x^2 - y^2 = n$, $x - y = 17$, $x + y = 437$. Daher ist $\gcd(x - y, n) = 17$. Das ist ein echter Teiler von n.

10.3.2 Bestimmung von x und y

Das beschriebene Prinzip wird auch in anderen Faktorisierungsalgorithmen, z.B. im Zahlkörpersieb (Number Field Sieve, siehe [48]), angewendet. Die Verfahren unterscheiden sich aber in der Art und Weise, wie x und y berechnet werden. Wir beschreiben, wie das im Quadratischen Sieb gemacht wird.

Sei

$$m = \lfloor \sqrt{n} \rfloor$$

und

$$f(X) = (X + m)^2 - n.$$

Wir erläutern das Verfahren zuerst an einem Beispiel.

Beispiel 10.3.2. Wie in Beispiel 10.3.1 sei $n = 7429$. Dann ist $m = 86$ und $f(X) = (X + 86)^2 - 7429$. Es gilt

$$\begin{array}{lcrcrcl} \text{f(-3)} & = & 83^2 - 7429 & = & \text{-540} & = & -1 * 2^2 * 3^3 * 5 \\ \text{f(1)} & = & 87^2 - 7429 & = & 140 & = & 2^2 * 5 * 7 \\ \text{f(2)} & = & 88^2 - 7429 & = & 315 & = & 3^2 * 5 * 7 \end{array}$$

Hieraus folgt

$$\begin{array}{lcll} 83^2 & \equiv & -1 * 2^2 * 3^3 * 5 & \mod 7429 \\ 87^2 & \equiv & 2^2 * 5 * 7 & \mod 7429 \\ 88^2 & \equiv & 3^2 * 5 * 7 & \mod 7429 \end{array}$$

Multipliziert man die letzten beiden Kongruenzen, so erhält man

$$(87 * 88)^2 \equiv (2 * 3 * 5 * 7)^2 \bmod n.$$

Man kann also

$$x = 87 * 88 \bmod n = 227, \quad y = 2 * 3 * 5 * 7 \bmod n = 210$$

setzen. Das sind die Werte für x und y aus Beispiel 10.3.1.

In Beispiel 10.3.2 werden Zahlen s angegeben, für die $f(s)$ nur kleine Primfaktoren hat. Es wird ausgenutzt, daß

$$(s + m)^2 \equiv f(s) \bmod n \tag{10.3}$$

ist, und es werden Kongruenzen der Form (10.3) ausgewählt, deren Produkt auf der linken und rechten Seite ein Quadrat ergibt. Auf der linken Seite einer Kongruenz (10.3) steht ohnehin ein Quadrat. Das Produkt beliebiger linker Seiten ist also immer ein Quadrat. Auf der rechten Seite ist die Primfaktorzerlegung bekannt. Man erhält also ein Quadrat, wenn die Exponenten aller Primfaktoren und der Exponent von -1 gerade sind. Wir erklären als nächstes, wie geeignete Kongruenzen ausgewählt werden und anschließend, wie die Kongruenzen bestimmt werden.

10.3.3 Auswahl geeigneter Kongruenzen

In Beispiel 10.3.2 kann man direkt sehen, welche Kongruenzen multipliziert werden müssen, damit das Produkt der rechten Seiten der Kongruenzen ein Quadrat ergibt. Bei großen Zahlen n muß man die geeigneten Kongruenzen aus mehr als 100 000 möglichen Kongruenzen auswählen. Dann wendet man lineare Algebra an. Dies wird im nächsten Beispiel illustriert.

Beispiel 10.3.3. Wir zeigen, wie die Auswahl der geeigneten Kongruenzen in Beispiel 10.3.2 durch Lösung eines linearen Gleichungssystems erfolgt. Drei Kongruenzen stehen zur Wahl. Daraus sollen die Kongruenzen so ausgewählt werden, daß das Produkt der rechten Seiten ein Quadrat ergibt. Man sucht also Zahlen $\lambda_i \in \{0, 1\}$, $1 \leq i \leq 3$, für die

$$\begin{aligned}(-1 * 2^2 * 3^3 * 5)^{\lambda_1} * (2^2 * 5 * 7)^{\lambda_2} * (3^2 * 5 * 7)^{\lambda_3} = \\ (-1)^{\lambda_1} * 2^{2\lambda_1+2\lambda_2} * 3^{3\lambda_1+2\lambda_3} * 5^{\lambda_1+\lambda_2+\lambda_3} * 7^{\lambda_2+\lambda_3}\end{aligned}$$

ein Quadrat ist. Diese Zahl ist genau dann ein Quadrat, wenn der Exponent von -1 und die Exponenten aller Primzahlen gerade sind. Man erhält also das folgende Kongruenzensystem:

$$\begin{aligned}
\lambda_1 &\equiv 0 \bmod 2\\
2\lambda_1 + 2\lambda_2 &\equiv 0 \bmod 2\\
3\lambda_1 + 2\lambda_3 &\equiv 0 \bmod 2\\
\lambda_1 + \lambda_2 + \lambda_3 &\equiv 0 \bmod 2\\
\lambda_2 + \lambda_3 &\equiv 0 \bmod 2
\end{aligned}$$

Die Koeffizienten der Unbekannten λ_i kann man modulo 2 reduzieren. Man erhält dann das vereinfachte System

$$\begin{aligned}
\lambda_1 &\equiv 0 \bmod 2\\
\lambda_1 + \lambda_2 + \lambda_3 &\equiv 0 \bmod 2\\
\lambda_2 + \lambda_3 &\equiv 0 \bmod 2.
\end{aligned}$$

Daraus erhält man die Lösung

$$\lambda_1 = 0, \quad \lambda_2 = \lambda_3 = 1.$$

Wir skizzieren kurz, wie das Quadratische Sieb geeignete Kongruenzen im allgemeinen findet.

Man wählt eine natürliche Zahl B. Gesucht werden kleine ganze Zahlen s, für die $f(s)$ nur Primfaktoren in der *Faktorbasis*

$$F(B) = \{p \in \mathbb{P} : p \leq B\} \cup \{-1\}$$

hat. Solche Werte $f(s)$ heißen B-glatt. Tabelle 10.2 gibt einen Eindruck von den Faktorbasisgrößen. Hat man soviele Zahlen s gefunden, wie die Faktorbasis Elemente hat, so stellt man das entsprechende lineare Kongruenzensystem auf und löst es. Da das Kongruenzensystem tatsächlich ein lineares Gleichungssystem über dem Körper $\mathbb{Z}/2\mathbb{Z}$ ist, kann man zu seiner Lösung den Gauß-Algorithmus verwenden. Da aber die Anzahl der Gleichungen und die Anzahl der Variablen sehr groß ist, verwendet man statt dessen spezialisierte Verfahren, auf die wir hier aber nicht näher eingehen.

10.3.4 Das Sieb

Es bleibt noch zu klären, wie die Zahlen s gefunden werden, für die $f(s)$ B-glatt ist. Man könnte für $s = 0, \pm 1, \pm 2, \pm 3, \ldots$ den Wert $f(s)$ berechnen und dann durch Probedivision ausprobieren, ob $f(s)$ B-glatt ist. Das ist aber sehr aufwendig. Um herauszufinden, daß $f(s)$ nicht B-glatt ist, muß man nämlich durch alle Primzahlen $p \leq B$ dividieren. Wie man in Tabelle 10.2 sieht, sind die Faktorbasen sehr groß, und daher kann die Probedivison sehr lange dauern. Schneller geht ein Siebverfahren.

Wir beschreiben eine vereinfachte Form des Siebverfahrens. Man fixiert ein *Siebintervall*

$$S = \{-C, -C+1, \ldots, 0, 1, \ldots, C\}.$$

Gesucht werden alle $s \in S$,für die $f(s)$ B-glatt ist. Man berechnet zuerst alle Werte $f(s)$, $s \in S$. Für jede Primzahl p der Faktorbasis dividiert man alle Werte $f(s)$ durch die höchstmögliche p-Potenz. Die B-glatten Werte $f(s)$ sind genau diejenigen, bei denen eine 1 oder -1 übrigbleibt.

Um herauszufinden, welche Werte $f(s) = (s+m)^2 - n$ durch eine Primzahl p der Faktorbasis teilbar sind, bestimmt man zuerst die Zahlen $s \in \{0, 1, \ldots, p-1\}$, für die $f(s)$ durch p teilbar ist. Da das Polynom $f(X)$ höchstens zwei Nullstellen modulo p hat, sind das entweder zwei, eine oder keine Zahl s. Für kleine Primzahlen kann man die Nullstellen durch ausprobieren finden. Ist p groß, muß man andere Methoden anwenden (siehe [4]). Geht man von diesen Nullstellen in Schritten der Länge p nach rechts und links durch das Siebintervall, so findet man alle s-Werte, für die $f(s)$ durch p teilbar ist. Diesen Vorgang nennt man *Sieb* mit p. Man dividiert dabei nur die teilbaren Werte $f(s)$. Es gibt keine erfolglosen Probedivisionen mehr.

Beispiel 10.3.4. Wie in Beispiel 10.3.1 und Beispiel 10.3.2 sei $n = 7429$, $m = 86$ und $f(X) = (X+86)^2 - 7429$. Als Faktorbasis wähle die Menge $\{2, 3, 5, 7\} \cup \{-1\}$ und als Siebintervall die Menge $\{-3, -2, \ldots, 3\}$. Das Sieb ist in Tabelle 10.1 dargestellt.

s	-3	-2	-1	0	1	2	3
$(s+m)^2 - n$	-540	-373	-204	-33	140	315	492
Sieb mit 2	-135		-51		35		123
Sieb mit 3	-5		-17	-11		35	41
Sieb mit 5	-1				7	7	
Sieb mit 7					1	1	

Tabelle 10.1. Das Sieb

Das Sieb kann noch sehr viel effizienter gestaltet werden. Das wird hier aber nicht weiter beschrieben. Wir verweisen statt dessen auf [64].

# Dezimalstellen von n	50	60	70	80	90	100	110	120
# Faktorbasis in Tausend	3	4	7	15	30	51	120	245
# Siebintervall in Millionen	0,2	2	5	6	8	14	16	26

Tabelle 10.2. Siebintervall- und Faktorbasisgrößen

10.4 Analyse des Quadratischen Siebs

In diesem Abschnitt skizzieren wir die Analyse des Quadratischen Siebs, damit der Leser einen Eindruck erhält, warum das Quadratische Sieb effizienter ist als Probedivision. Die in der Analyse verwendeten Techniken gehen über den Rahmen dieses Buches hinaus. Darum werden sie nur angedeutet. Interessierten Lesern wird als Einstieg in eine vertiefte Beschäftigung mit dem Gegenstand [47] empfohlen.

Seien n, u, v reelle Zahlen und sei n größer als die Eulersche Konstante e. Dann schreibt man

$$L_n[u,v] = e^{v(\log n)^u (\log\log n)^{1-u}}. \tag{10.4}$$

Diese Funktion wird zur Beschreibung der Laufzeit von Faktorisierungsalgorithmen verwendet. Wir erläutern zuerst ihre Bedeutung.

Es ist

$$L_n[0,v] = e^{v(\log n)^0 (\log\log n)^1} = (\log n)^v \tag{10.5}$$

und

$$L_n[1,v] = e^{v(\log n)^1 (\log\log n)^0} = e^{v\log n}. \tag{10.6}$$

Ein Algorithmus, der die Zahl n faktorisieren soll, erhält als Eingabe n. Die binäre Länge von n ist $\lfloor \log_2 n \rfloor + 1 = O(\log n)$. Hat der Algorithmus die Laufzeit $L_n[0,v]$, so ist seine Laufzeit polynomiell, wie man aus (10.5) sieht. Dabei ist v der Grad des Polynoms. Der Algorithmus gilt dann als effizient. Die praktische Effizienz hängt natürlich vom Polynomgrad ab. Hat der Algorithmus die Laufzeit $L_n[1,v]$, so ist seine Laufzeit exponentiell, wie man aus (10.6) sieht. Der Algorithmus gilt als ineffizient. Hat der Algorithmus die Laufzeit $L_n[u,v]$ mit $0 < u < 1$, so ist heißt seine Laufzeit *subexponentiell*. Sie ist schlechter als polynomiell und besser als exponentiell. Die schnellsten Faktorisierungsalgorithmen haben subexponentielle Laufzeit.

Die Laufzeit der Probedivision zur Faktorisierung ist exponentiell.

Die Laufzeit des Quadratischen Siebs konnte bis jetzt nicht völlig analysiert werden. Wenn man aber einige plausible Annahmen macht, dann ist die Laufzeit des Quadratischen Siebs $L_n[1/2, 1+o(1)]$. Hierin steht $o(1)$ für eine Funktion, die gegen 0 konvergiert, wenn n gegen Unendlich strebt. Die Laufzeit des Quadratischen Siebs liegt also genau in der Mitte zwischen polynomiell und exponentiell.

Wir begründen die Laufzeit des Quadratischen Siebs. Im Quadratischen Sieb werden Schranken B und C festgelegt und dann werden diejenigen Zahlen s im Siebintervall $S = \{-C, -C+1, \ldots, C\}$ bestimmt, für die

$$f(s) = (s+m)^2 - n = s^2 + 2ms + m^2 - n \tag{10.7}$$

B-glatt ist. Die Schranken B und C müssen so gewählt sein, daß die Anzahl der gefundenen Werte s genauso groß ist wie die Anzahl der Elemente der Faktorbasis.

Da $m = \lfloor \sqrt{n} \rfloor$, ist $m^2 - n$ sehr klein. Für kleines s ist $f(s)$ nach (10.7) daher in derselben Größenordnung wie $\sqrt{n}$. Wir nehmen an, daß der Anteil der B-glatten Werte $f(s)$, $s \in S$, genauso groß ist wie der Anteil der B-glatten Werte aller natürlichen Zahlen $\leq \sqrt{n}$. Diese Annahme wurde nie bewiesen, und es ist auch unklar, wie sie bewiesen werden kann. Sie ist aber experimentell verifizierbar und ermöglicht die Analyse des Quadratischen Siebs.

Die Anzahl der B-glatten natürlichen Zahlen unter einer Schranke x wird mit $\psi(x, B)$ bezeichnet. Sie wird im folgenden Satz abgeschätzt, der in [23] bewiesen wurde.

Theorem 10.4.1. *Sei ε eine positive reelle Zahl. Dann gilt für alle reellen Zahlen $x \geq 10$ und $w \leq (\log x)^{1-\varepsilon}$*

$$\psi(x, x^{1/w}) = xw^{-w+f(x,w)}$$

für eine Funktion f, die $f(x, w)/w \to 0$ für $w \to \infty$ und gleichmäßig für alle x erfüllt.

Theorem 10.4.1 bedeutet, daß der Anteil der $x^{1/w}$-glatten Zahlen, die kleiner gleich x sind, ungefähr w^{-w} ist.

Aus diesem Satz läßt sich folgendes Resultat ableiten:

Korollar 10.4.2. *Seien a, u, v positive reelle Zahlen. Dann gilt für $n \in \mathbb{N}$, $n \to \infty$*

$$\psi(n^a, L_n[u, v]) = n^a L_n[1 - u, -(a/v)(1 - u) + o(1)].$$

Beweis. Es ist

$$L_n[u, v] = (e^{(\log n)^u (\log\log n)^{1-u}})^v = n^{v((\log\log n)/\log n)^{1-u}}.$$

Setzt man also

$$w = (a/v)((\log n)/(\log\log n))^{1-u}$$

und wendet Theorem 10.4.1 an, so erhält man

$$\psi(n^a, L_n[u, v]) = n^a w^{-w(1+o(1))}.$$

Nun ist

$$w^{-w(1+o(1))} = (e^{(1-u)(\log(a/v)+\log\log n-\log\log\log n)(-(a/v)((\log n)/(\log\log n))^{1-u}(1+o(1))})^{1-u}.$$

Hierin ist

$$\log(a/v) + \log\log n - \log\log\log n = \log\log n(1 + o(1)).$$

Daher ist

$$
\begin{aligned}
& w^{-w(1+o(1))} \\
&= e^{(\log n)^{1-u}(\log\log n)^u(-(a/v)(1-u)+o(1))} \\
&= L_n[1-u, -(a/v)(1-u)+o(1)].
\end{aligned}
$$

Damit ist die Behauptung bewiesen. □

Im Quadratischen Sieb werden Zahlen $f(s)$ erzeugt, die von der Größenordnung $n^{1/2}$ sind. In Korollar 10.4.2 ist also $a = 1/2$. Um ein s zu finden, für das $f(s)$ $L_n[u,v]$-glatt ist, braucht man nach Korollar 10.4.2 $L_n[1-u, (1/(2v))(1-u)+o(1)]$ Elemente im Siebintervall. Die Anzahl der Elemente der Faktorbasis ist höchstens $L_n[u,v]$. Insgesamt braucht man also $L_n[u,v]$ solche Werte s, damit man das Gleichungssystem lösen kann. Die Zeit zur Berechnung der passenden Werte für s ist also ein Vielfaches von $L_n[u,v]L_n[1-u,(1/(2v))(1-u)+o(1)]$. Dieser Wert wird minimal für $u = 1/2$. Wir wählen also $u = 1/2$.

Die Faktorbasis enthält also alle Primzahlen $p \leq B = L_n[1/2, v]$. Für jede erfolgreiche Zahl s braucht das Siebintervall $L_n[1/2, 1/(4v)]$ Elemente. Da insgesamt $L_n[1/2,v]$ erfolgreiche Werte s berechnet werden müssen, ist die Größe des Siebintervalls $L_n[1/2,v]L_n[1/2,1/(4v)] = L_n[1/2, v+1/(4v)]$.

Ein geeigneter Wert für v wird in der Analyse noch gefunden. Wir tragen zuerst die Laufzeiten für die einzelnen Schritte zusammen.

Die Berechnung der Quadratwurzeln von $f(X)$ modulo p für eine Primzahl p in der Faktorbasis ist in erwarteter Polynomzeit möglich. Daraus leitet man ab, daß die Zeit zur Berechnung der Wurzeln für alle Faktorbasiselemente $L_n[1/2, v+o(1)]$ ist.

Die Siebzeit pro Primzahl p ist $O(L_n[1/2, v+1/(4v)+o(1)]/p)$, weil man in Schritten der Länge p durch ein Intervall der Länge $L[1/2,v]$ geht. Daraus kann man ableiten, daß die gesamte Siebzeit einschließlich der Vorberechnung $L_n[1/2, v+1/(4v)+o(1)]$ ist.

Mit dem Algorithmus von Wiedemann, der ein spezialisierter Gleichungslöser für dünn besetzte Systeme ist, benötigt die Lösung des Gleichungssystems Zeit $L_n[1/2, 2v+o(1)]$. Der Wert $v = 1/2$ minimiert die Siebzeit und macht Siebzeit und Zeit zum Gleichungslösen gleich. Wir erhalten also insgesamt die Laufzeit $L_n[1/2, 1+o(1)]$.

10.5 Effizienz anderer Faktorisierungsverfahren

Nach der Analyse des Quadratischen Siebs im letzten Abschnitt stellen sich zwei Fragen: Gibt es effizientere Faktorisierungsalgorithmen und gibt es Faktorisierungsverfahren, deren Laufzeit man wirklich beweisen kann?

Der effizienteste Faktorisierungsalgorithmus, dessen Laufzeit bewiesen werden kann, benutzt quadratische Formen. Es handelt sich um einen probabilistischen Algorithmus mit erwarteter Laufzeit $L_n[1/2, 1+o(1)]$. Seine

Laufzeit entspricht also der des Quadratischen Siebs. Die Laufzeit wurde in [49] bewiesen.

Die Elliptische-Kurven-Methode (ECM) ist ebenfalls ein probabilistischer Algorithmus mit erwarteter Laufzeit $L_p[1/2, \sqrt{1/2}]$ wobei p der kleinste Primfaktor von n ist. Während das Quadratische Sieb für Zahlen n gleicher Größe gleich lang braucht, wird ECM schneller, wenn n einen kleinen Primfaktor hat. Ist der kleinste Primfaktor aber von der Größenordnung $\sqrt{n}$, dann hat ECM die erwartete Laufzeit $L_n[1/2, 1]$ genau wie das Quadratische Sieb. In der Praxis ist das Quadratische Sieb in solchen Fällen sogar schneller.

Bis 1988 hatten die schnellsten Faktorisierungsalgorithmen die Laufzeit $L_n[1/2, 1]$. Es gab sogar die Meinung, daß es keine schnelleren Faktorisierungsalgorithmen geben könne. Wie man aus 10.4.2 sieht, ist das auch richtig, solange man versucht, natürliche Zahlen n mit glatten Zahlen der Größenordnung n^a für eine feste positive reelle Zahl a zu faktorisieren. Im Jahre 1988 zeigte aber John Pollard, daß es mit Hilfe der algebraischen Zahlentheorie möglich ist, zur Erzeugung der Kongruenzen (10.1) und (10.2) systematisch kleinere Zahlen zu verwenden. Aus der Idee von Pollard wurde das Zahlkörpersieb (Number Field Sieve, NFS). Unter geeigneten Annahmen kann man zeigen, daß es die Laufzeit $L_n[1/3, (64/9)^{1/3}]$ hat. Es ist damit einem Polynomzeitalgorithmus wesentlich näher als das Quadratische Sieb. Eine Sammlung von Arbeiten zum Zahlkörpersieb findet man in [48].

In den letzten zwanzig Jahren hat es dramatische Fortschritte bei der Lösung des Faktorisierungsproblems gegeben. Shor [74] hat bewiesen, dass auf Quantencomputern natürliche Zahlen in Polynomzeit faktorisiert werden können. Es ist nur nicht klar, ob und wann es entsprechende Quantencomputer geben wird. Aber es ist auch möglich, daß ein klassischer polynomieller Faktorisierungsalgorithmus gefunden wird. Mathematische Fortschritte sind eben nicht voraussagbar. Dann sind das RSA-Verfahren, das Rabin-Verfahren und all die anderen Verfahren, die ihre Sicherheit aus der Schwierigkeit des Faktorisierungsproblems beziehen, unsicher.

Aktuelle Faktorisierungsrekorde findet man zum Beispiel in [29]. So konnte am 9. Mai 2005 die von den RSA-Laboratories veröffentlichte 200-stellige Challenge-Zahl mit dem Zahlkörpersieb faktorisiert werden [67]. Am 21. Mai 2007 wurde die 307-stellige Mersenne-Zahl $2^{1039} - 1$ faktorisiert werden [54].

10.6 Übungen

Übung 10.6.1 (Fermat-Faktorisierungsmethode). Fermat faktorisierte eine Zahl n, indem er eine Darstellung $n = x^2 - y^2 = (x-y)(x+y)$ berechnete. Faktorisieren Sie auf diese Weise $n = 13199$ möglichst effizient. Funktioniert diese Methode immer? Wie lange braucht die Methode höchstens?

Übung 10.6.2. Faktorisieren Sie 831802500 mit Probedivision.

Übung 10.6.3. Faktorisieren Sie $n = 138277151$ mit der $p-1$-Methode.

Übung 10.6.4. Faktorisieren Sie $n = 18533588383$ mit der $p-1$-Methode.

Übung 10.6.5. Schätzen Sie die Laufzeit der $(p-1)$-Methode ab.

Übung 10.6.6. Die *Random-Square-Methode* von Dixon ist der Quadratisches-Sieb-Methode ähnlich. Der Hauptunterschied besteht darin, daß die Relationen gefunden werden, indem x^2 mod n faktorisiert wird, wobei x eine Zufallszahl in $\{1, \ldots, n-1\}$ ist. Verwenden Sie die Random-Square-Methode, um 11111 mit einer möglichst kleinen Faktorbasis zu faktorisieren.

Übung 10.6.7. Finden Sie mit dem quadratischen Sieb einen echten Teiler von 11111.

Übung 10.6.8. Zeichnen Sie die Funktion $f(k) = L_{2^k}[1/2, 1]$ für $k \in \{1, 2, \ldots, 2048\}$.

11. Diskrete Logarithmen

In diesem Kapitel geht es um das Problem, diskrete Logarithmen zu berechnen (DL-Problem). Nur in Gruppen, in denen das DL-Problem schwierig zu lösen ist, können das ElGamal-Verschlüsselungsverfahren (siehe Abschnitt 9.6) und viele andere Public-Key-Verfahren sicher sein. Daher ist das DL-Problem von großer Bedeutung in der Kryptographie.

Wir werden zuerst Algorithmen behandeln, die in allen Gruppen funktionieren. Dann werden spezielle Algorithmen für endliche Körper beschrieben. Eine Übersicht über Techniken und neuere Resultate findet man in [69] und in [46].

11.1 Das DL-Problem

In diesem Kapitel sei G eine endliche zyklische Gruppe der Ordnung n, γ sei ein Erzeuger dieser Gruppe und 1 sei das neutrale Element in G. Wir gehen davon aus, daß die Gruppenordnung n bekannt ist. Viele Algorithmen zur Lösung des DL-Problems funktionieren auch mit einer oberen Schranke für die Gruppenordnung. Ferner sei α ein Gruppenelement. Wir wollen die kleinste nicht negative ganze Zahl x finden, für die

$$\alpha = \gamma^x \tag{11.1}$$

gilt, d.h. wir wollen den *diskreten Logarithmus* von α zur Basis γ berechnen.

Man kann das DL-Problem auch allgemeiner formulieren: In einer Gruppe H, die nicht notwendig zyklisch ist, sind zwei Elemente α und γ gegeben. Gefragt ist, ob es einen Exponenten x gibt, der (11.1) erfüllt. Wenn ja, ist das kleinste nicht negative x gesucht. Man muß also erst entscheiden, ob es überhaupt einen diskreten Logarithmus gibt. Wenn ja, muß man ihn finden. In kryptographischen Anwendungen ist es aber fast immer klar, daß der diskrete Logarithmus existiert. Das Entscheidungsproblem ist also aus kryptographischer Sicht unbedeutend. Daher betrachten wir nur DL-Probleme in zyklischen Gruppen. Die Basis ist immer ein Erzeuger der Gruppe.

11.2 Enumeration

Die einfachste Methode, den diskreten Logarithmus x aus (11.1) zu berechnen, besteht darin, für $x = 0, 1, 2, 3, \ldots$ zu prüfen, ob (11.1) erfüllt ist. Sobald die Antwort positiv ist, ist der diskrete Logarithmus gefunden. Dieses Verfahren bezeichnen wir als *Enumerationsverfahren*. Es erfordert $x - 1$ Multiplikationen und x Vergleiche in G. Man braucht dabei nur die Elemente α, γ und γ^x zu speichern. Also braucht man Speicherplatz für drei Gruppenelemente.

Beispiel 11.2.1. Wir bestimmen den diskreten Logarithmus von 3 zur Basis 5 in $(\mathbb{Z}/2017\mathbb{Z})^*$. Probieren ergibt $x = 1030$. Dafür braucht man 1029 Multiplikationen modulo 2017.

In kryptographischen Verfahren ist $x \geq 2^{160}$. Das Enumerationsverfahren ist dann nicht durchführbar. Man braucht dann nämlich wenigstens $2^{160} - 1$ Gruppenoperationen.

11.3 Shanks Babystep-Giantstep-Algorithmus

Eine erste Methode zur schnelleren Berechnung diskreter Logarithmen ist der Babystep-Giantstep-Algorithmus von Shanks. Bei diesem Verfahren muß man viel weniger Gruppenoperationen machen als bei der Enumeration, aber man braucht mehr Speicherplatz. Man setzt

$$m = \lceil \sqrt{n} \rceil$$

und macht den Ansatz

$$x = qm + r, \quad 0 \leq r < m.$$

Dabei ist r also der Rest und q ist der Quotient der Division von x durch m. Der Babystep-Giantstep-Algorithmus berechnet q und r. Dies geschieht folgendermaßen:

Es gilt

$$\gamma^{qm+r} = \gamma^x = \alpha.$$

Daraus folgt

$$(\gamma^m)^q = \alpha\gamma^{-r}.$$

Man berechnet nun zuerst die Menge der *Babysteps*

$$B = \{(\alpha\gamma^{-r}, r) : 0 \leq r < m\}.$$

Findet man ein Paar $(1, r)$, so kann man $x = r$ setzen und hat das DL-Problem gelöst. Andernfalls bestimmt man

$$\delta = \gamma^m$$

und prüft, ob für $q = 1, 2, 3, \ldots$ das Gruppenelement δ^q als erste Komponente eines Elementes von B vorkommt, ob also ein Paar (δ^q, r) zu B gehört. Sobald dies der Fall ist, gilt

$$\alpha\gamma^{-r} = \delta^q = \gamma^{qm}$$

und man hat den diskreten Logarithmus

$$x = qm + r$$

gefunden. Die Berechnung der Elemente δ^q, $q = 1, 2, 3 \ldots$ nennt man *Giantsteps* . Um die Überprüfung, ob δ^q als erste Komponente eines Elementes der Babystep-Menge vorkommt, effizient zu gestalten, nimmt man die Elemente dieser Menge in eine Hashtabelle auf (siehe [21], Kapitel 12), wobei die erste Komponente eines jeden Elementes als Schlüssel dient.

Beispiel 11.3.1. Wir bestimmen den diskreten Logarithmus von 3 zur Basis 5 in $(\mathbb{Z}/2017\mathbb{Z})^*$. Es ist $\gamma = 5 + 2017\mathbb{Z}$, $\alpha = 3 + 2017\mathbb{Z}$, $m = \lceil\sqrt{2016}\rceil = 45$. Die Babystep-Menge ist

$$\begin{aligned} B = \{&(3,0),(404,1),(1291,2),(1065,3),(213,4),(446,5),(896,6),\\ &(986,7),(1004,8),(1411,9),(1089,10),(1428,11),(689,12),(1348,13),\\ &(673,14),(538,15),(511,16),(909,17),(1392,18),(1892,19),(1992,20),\\ &(2012,21),(2016,22),(1210,23),(242,24),(1662,25),(1946,26),\\ &(1196,27),(1046,28),(1016,29),(1010,30),(202,31),(1654,32),\\ &(1541,33),(1115,34),(223,35),(448,36),(493,37),(502,38),(1714,39),\\ &(1553,40),(714,41),1353,42),(674,43),(1345,44)\} \end{aligned}$$

Hierbei wurden die Restklassen durch ihre kleinsten nicht negativen Vertreter dargestellt.

Man berechnet nun $\delta = \gamma^m = 45 + 2017\mathbb{Z}$. Die Giantsteps berechnen sich zu

$$\begin{aligned} &45, 8, 360, 64, 863, 512, 853, 62, 773, 496, 133, 1951,\\ &1064, 1489, 444, 1827, 1535, 497, 178, 1959, 1424, 1553. \end{aligned}$$

Man findet $(1553, 40)$ in der Babystep-Menge. Es ist also $\alpha\gamma^{-40} = 1553 + 2017\mathbb{Z}$. Andererseits wurde 1553 als zweiundzwanzigster Giantstep gefunden. Also gilt

$$\gamma^{22*45} = \alpha\gamma^{-40}.$$

Damit ist

$$\gamma^{22*45+40} = \alpha.$$

Als Lösung des DL-Problems findet man $x = 22 * 45 + 40 = 1030$. Um die Babystep-Menge zu berechnen, brauchte man 45 Multiplikationen. Um die

Giantsteps zu berechnen, mußte man zuerst δ berechnen und dann brauchte man 21 Multiplikationen in G. Die Anzahl der Multiplikationen ist also deutlich geringer als beim Enumerationsverfahren, aber man muß viel mehr Elemente speichern. Außerdem muß man für 22 Gruppenelemente β prüfen, ob es ein Paar (β, r) in der Babystep-Menge gibt.

Wenn man unterstellt, daß es mittels einer Hashtabelle möglich ist, mit konstant vielen Vergleichen zu prüfen, ob ein gegebenes Gruppenelement erste Komponente eines Paares in der Babystep-Menge ist, dann kann man folgenden Satz leicht verifizieren.

Theorem 11.3.2. *Der Babystep-Giantstep-Algorithmus benötigt* $O(\sqrt{|G|})$ *Multiplikationen und Vergleiche in* G*. Er muß* $O(\sqrt{|G|})$ *viele Elemente in* G *speichern.*

Der Zeit- und Platzbedarf des Babystep-Giantstep-Algorithmus ist von der Größenordnung $\sqrt{|G|}$. Ist $|G| > 2^{160}$, so ist der Algorithmus in der Praxis nicht mehr einsetzbar.

11.4 Der Pollard-ρ-Algorithmus

Das Verfahren von Pollard, das in diesem Abschnitt beschrieben wird, benötigt wie der Babystep-Giantstep-Algorithmus $O(\sqrt{|G|})$ viele Gruppenoperationen, aber nur konstant viele Speicherplätze.

Wieder wollen wir das DL-Problem (11.1) lösen. Gebraucht werden drei paarweise disjunkte Teilmengen G_1, G_2, G_3 von G, deren Vereinigung die ganze Gruppe G ist. Sei die Funktion $f : G \to G$ definiert durch

$$f(\beta) = \begin{cases} \gamma\beta & \text{falls } \beta \in G_1, \\ \beta^2 & \text{falls } \beta \in G_2, \\ \alpha\beta & \text{falls } \beta \in G_3. \end{cases}$$

Wir wählen eine Zufallszahl x_0 in der Menge $\{1, \ldots, n\}$ und setzen $\beta_0 = \gamma^{x_0}$. Dann berechnen wir die Folge (β_i) nach der Rekursion

$$\beta_{i+1} = f(\beta_i).$$

Wir können die Glieder dieser Folge darstellen als

$$\beta_i = \gamma^{x_i}\alpha^{y_i}, \quad i \geq 0.$$

Dabei ist x_0 der zufällig gewählte Startwert, $y_0 = 0$, und es gilt

$$x_{i+1} = \begin{cases} x_i + 1 \mod n & \text{falls } \beta_i \in G_1, \\ 2x_i \mod n & \text{falls } \beta_i \in G_2, \\ x_i & \text{falls } \beta_i \in G_3 \end{cases}$$

und

$$y_{i+1} = \begin{cases} y_i & \text{falls } \beta_i \in G_1, \\ 2y_i \mod n & \text{falls } \beta_i \in G_2, \\ y_i + 1 \mod n & \text{falls } \beta_i \in G_3. \end{cases}$$

Da es nur endlich viele verschiedene Gruppenelemente gibt, müssen in dieser Folge zwei gleiche Gruppenelemente vorkommen. Es muß also $i \geq 0$ und $k \geq 1$ geben mit $\beta_{i+k} = \beta_i$. Das bedeutet, daß

$$\gamma^{x_i} \alpha^{y_i} = \gamma^{x_{i+k}} \alpha^{y_{i+k}}.$$

Daraus folgt

$$\gamma^{x_i - x_{i+k}} = \alpha^{y_{i+k} - y_i}.$$

Für den diskreten Logarithmus x von α zur Basis γ gilt also

$$(x_i - x_{i+k}) \equiv x(y_{i+k} - y_i) \mod n.$$

Diese Kongruenz muß man lösen. Ist die Lösung nicht eindeutig mod n, so muß man die richtige Lösung durch Ausprobieren ermitteln. Wenn das nicht effizient genug geht, weil zu viele Möglichkeiten bestehen, wiederholt man die gesamte Berechnung mit einem neuen Startwert x_0.

Wir schätzen die Anzahl der Folgenglieder β_i ab, die berechnet werden müssen, bis ein *Match* gefunden ist, also ein Paar $(i, i+k)$ von Indizes, für das $\beta_{i+k} = \beta_i$ gilt. Dazu verwenden wir das Geburtstagsparadox. Die Geburtstage sind die Gruppenelemente. Wir nehmen an, daß die Elemente der Folge $(\beta_i)_{i \geq 0}$ unabhängig und gleichverteilt zufällig gewählt sind. Das stimmt zwar nicht, aber sie ist so konstruiert, daß sie einer Zufallsfolge sehr ähnlich ist. Wie in Abschnitt 5.3 gezeigt, werden $O(\sqrt{|G|})$ Folgenelemente benötigt, damit die Wahrscheinlichkeit für ein Match größer als $1/2$ ist.

So, wie der Algorithmus bis jetzt beschrieben wurde, muß man alle Tripel (β_i, x_i, y_i) speichern. Der Speicherplatzbedarf ist dann $O(\sqrt{|G|})$, wie beim Algorithmus von Shanks. Tatsächlich genügt es aber, nur ein Tripel zu speichern. Der Pollard-ρ-Algorithmus ist also viel Speicher-effizienter als der Algorithmus von Shanks. Am Anfang speichert man (β_1, x_1, y_1). Hat man gerade (β_i, x_i, y_i) gespeichert, so berechnet man (β_j, x_j, y_j) für $j = i+1, i+2, \ldots$ bis man ein Match (i, j) findet oder bis $j = 2i$ ist. Im letzteren Fall löscht man β_i und speichert statt dessen β_{2i}. Gespeichert werden also nur die Tripel (β_i, x_i, y_i) mit $i = 2^k$. Bevor gezeigt wird, daß im modifizierten Algorithmus wirklich ein Match gefunden wird, geben wir ein Beispiel:

Beispiel 11.4.1. Wir lösen mit dem Pollard-ρ-Algorithmus die Kongruenz

$$5^x \equiv 3 \mod 2017.$$

Alle Restklassen werden durch ihre kleinsten nicht negativen Vertreter dargestellt. Wir setzen

$$G_1 = \{1, \ldots, 672\}, G_2 = \{673, \ldots, 1344\}, G_3 = \{1345, \ldots, 2016\}.$$

Als Startwert nehmen wir $x_0 = 1023$. Wir geben nur die Werte für die gespeicherten Tripel an und zusätzlich das letzte Tripel, das die Berechnung des diskreten Logarithmus ermöglicht.

j	β_j	x_j	y_j
0	986	1023	0
1	2	30	0
2	10	31	0
4	250	33	0
8	1366	136	1
16	1490	277	8
32	613	447	155
64	1476	1766	1000
98	1476	966	1128

Man erkennt, daß

$$5^{800} \equiv 3^{128} \mod 2017$$

ist. Zur Berechnung von x müssen wir die Kongruenz

$$128x \equiv 800 \mod 2016$$

lösen. Da $\gcd(128, 2016) = 32$ ein Teiler von 800 ist, existiert eine Lösung, die mod 63 eindeutig ist. Um x zu finden, löst man erst

$$4z \equiv 25 \bmod 63.$$

Wir erhalten $z = 22$. Der gesuchte diskrete Logarithmus ist einer der Werte $x = 22 + k * 63$, $0 \leq k < 32$. Für $k = 16$ findet man den diskreten Logarithmus $x = 1030$.

Auf die oben beschriebene Weise wird tatsächlich immer ein Match gefunden. Das wird jetzt bewiesen.

Zuerst zeigen wir, daß die Folge $(\beta_i)_{i \geq 0}$ periodisch wird. Sei $(s, s + k)$ das erste Match. Dann ist $k > 0$ und $\beta_{s+k} = \beta_s$. Weiter gilt $\beta_{s+k+l} = \beta_{s+l}$ für alle $l \geq 0$. Das liegt an der Konstruktion der Folge $(\beta_i)_{i \geq 0}$. Die Folge wird also tatsächlich periodisch. Man kann sie zeichnen wie den griechischen Buchstaben ρ. Die *Vorperiode* ist die Folge $\beta_0, \beta_1, \ldots, \beta_{s-1}$. Sie hat die Länge s. Die *Periode* ist die Folge $\beta_s, \beta_{s+1}, \ldots, \beta_{s+k-1}$. Sie hat die Länge k.

Nun erläutern wir, wann ein Match gefunden wird. Ist $i = 2^j \geq s$, so liegt das gespeicherte Element β_i in der Periode. Ist zusätzlich $2^j \geq k$, so ist die Folge

$$\beta_{2^j+1}, \beta_{2^j+2}, \ldots, \beta_{2^{j+1}}$$

wenigstens so lang wie die Periode. Eins ihrer Glieder stimmt also mit β_{2^j} überein. Diese Folge wird aber berechnet, nachdem b_{2^j} gespeichert wurde,

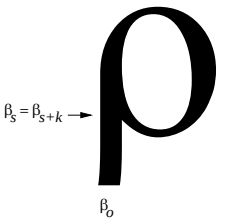


Abb. 11.1. Der Pollard-ρ-Algorithmus

und alle ihre Elemente werden mit β_{2^j} verglichen. Bei diesen Vergleichen wird also ein Match gefunden. Da die Summe aus Vorperioden- und Periodenlänge $O(\sqrt{|G|})$ ist, wird also nach Berechnung von $O(\sqrt{|G|})$ Folgengliedern ein Match gefunden. Der Algorithmus braucht also $O(\sqrt{|G|})$ Gruppenoperationen und muß $O(1)$ Tripel speichern.

Der Algorithmus wird effizienter, wenn man nicht nur ein, sondern acht Tripel speichert. Das macht man so: Zuerst sind diese 8 Tripel alle gleich (β_0, x_0, y_0). Später werden die Tripel nach und nach durch andere ersetzt. Sei i der Index des letzten gespeicherten Tripels. Am Anfang ist $i = 1$.

Für $j = 1, 2, \ldots$ berechnet man nun (β_j, x_j, y_j) und macht folgendes:

1. Wenn β_j mit einem gespeicherten Gruppenelement übereinstimmt, bricht man die Berechnung der β_j ab und versucht, den diskreten Logarithmus zu bestimmen.
2. Wenn $j \geq 3i$ ist, löscht man das erste gespeicherte Tripel und nimmt (β_j, x_j, y_j) als neues letztes gespeichertes Tripel.

Diese Modifikation ändert aber nichts an der Laufzeit oder dem Speicherbedarf des Algorithmus.

11.5 Der Pohlig-Hellman-Algorithmus

Wir zeigen nun, wie man das Problem der Berechnung diskreter Logarithmen in G auf dasselbe Problem in Gruppen von Primzahlordnung reduzieren kann, wenn man die Faktorisierung der Gruppenordnung $|G|$ kennt. Es sei also

$$n = |G| = \prod_{p|n} p^{e(p)}$$

die Primfaktorzerlegung von $n = |G|$.

11.5.1 Reduktion auf Primzahlpotenzordnung

Für jeden Primteiler p von n setzen wir

$$n_p = n/p^{e(p)}, \quad \gamma_p = \gamma^{n_p}, \quad \alpha_p = \alpha^{n_p}.$$

Dann ist die Ordnung von γ_p genau $p^{e(p)}$ und es gilt

$$\gamma_p^x = \alpha_p.$$

Das Element α_p liegt also in der von γ_p erzeugten zyklischen Untergruppe von G. Daher existiert der diskrete Logarithmus von α_p zur Basis γ_p. Der folgende Satz beschreibt die Berechnung von x aus den diskreten Logarithmen der α_p zur Basis γ_p.

Theorem 11.5.1. *Für alle Primteiler p von n sei $x(p)$ der diskrete Logarithmus von α_p zur Basis γ_p. Außerdem sei $x \in \{0, 1, \ldots, n-1\}$ Lösung der simultanen Kongruenz $x \equiv x(p) \bmod p^{e(p)}$ für alle Primteiler p von n. Dann ist x der diskrete Logarithmus von α zur Basis γ.*

Beweis. Es gilt

$$(\gamma^{-x}\alpha)^{n_p} = \gamma_p^{-x(p)}\alpha_p = 1$$

für alle Primteiler p von n. Daher ist die Ordnung des Elementes $\gamma^{-x}\alpha$ ein Teiler von n_p für alle Primteiler p von n und damit ein Teiler des größten gemeinsamen Teilers aller n_p. Dieser größte gemeinsame Teiler ist aber 1. Die Ordnung ist also 1 und damit gilt $\alpha = \gamma^x$. □

Man kann also x berechnen, indem man zuerst alle $x(p)$ bestimmt und dann den chinesischen Restsatz anwendet. Zur Berechnung eines $x(p)$ braucht der Babystep-Giantstep-Algorithmus oder der Algorithmus von Pollard nur noch $O(\sqrt{p^{e(p)}})$ viele Gruppenoperationen. Wenn n mehr als einen Primfaktor hat, ist das bereits deutlich schneller als wenn einer dieser Algorithmen in der gesamten Gruppe G angewendet wird. Die Rechenzeit für den chinesischen Restsatz kann man vernachlässigen.

Beispiel 11.5.2. Wie in Beispiel 11.3.1 sei G die prime Restklassengruppe mod 2017. Ihre Ordnung ist

$$2016 = 2^5 * 3^2 * 7.$$

Nach obiger Reduktion muß man $x(2)$ in einer Untergruppe der Ordnung $2^5 = 32$ bestimmen, $x(3)$ wird in einer Untergruppe der Ordnung 9 berechnet und $x(7)$ in einer Untergruppe der Ordnung 7. Dies wird im nächsten Abschnitt noch weiter vereinfacht.

11.5.2 Reduktion auf Primzahlordnung

Im vorigen Abschnitt haben wir gesehen, daß man die Berechnung diskreter Logarithmen in einer zyklischen Gruppe, für die man die Faktorisierung der Gruppenordnung kennt, auf DL-Berechnungen in Gruppen von Primzahlpotenzordnung zurückführen kann. Jetzt vereinfachen wir das DL-Problem noch weiter. Wir zeigen, wie die DL-Berechnung in einer zyklischen Gruppe von Primzahlpotenzordnung auf DL-Berechnungen in Gruppen von Primzahlordnung reduziert werden kann.

Sei also $|G| = n = p^e$ für eine Primzahl p. Wir wollen (11.1) in dieser Gruppe lösen. Wir wissen, daß $x < p^e$ gelten muß. Gemäß Theorem 2.3.3 kann man x in der Form

$$x = x_0 + x_1 p + \ldots + x_{e-1} p^{e-1}, \quad 0 \leq x_i < p, \quad 0 \leq i \leq e-1 \tag{11.2}$$

schreiben. Wir zeigen, daß sich jeder Koeffizient x_i, $0 \leq i \leq e-1$, als Lösung eines DL-Problems in einer Gruppe der Ordnung p bestimmen läßt.

Wir potenzieren die Gleichung $\gamma^x = \alpha$ mit p^{e-1}. Dann ergibt sich

$$\gamma^{p^{e-1}x} = \alpha^{p^{e-1}}. \tag{11.3}$$

Nun gilt nach (11.2)

$$p^{e-1}x = x_0 p^{e-1} + p^e(x_1 + x_2 p + \ldots x_{e-1} p^{e-2}). \tag{11.4}$$

Aus dem kleinen Satz von Fermat (siehe Theorem 3.11.1), (11.4) und (11.3) erhält man

$$(\gamma^{p^{e-1}})^{x_0} = \alpha^{p^{e-1}}. \tag{11.5}$$

Gemäß (11.5) ist der Koeffizient x_0 Lösung eines DL-Problems in einer Gruppe der Ordnung p, weil $\gamma^{p^{e-1}}$ die Ordnung p hat. Die anderen Koeffizienten bestimmt man rekursiv. Angenommen, $x_0, x_1, \ldots, x_{i-1}$ sind schon bestimmt. Dann gilt

$$\gamma^{x_i p^i + \ldots + x_{e-1} p^{e-1}} = \alpha \gamma^{-(x_0 + x_1 p + \ldots + x_{i-1} p^{i-1})}.$$

Bezeichne das Gruppenelement auf der rechten Seite mit α_i. Potenzieren dieser Gleichung mit p^{e-i-1} liefert

$$(\gamma^{p^{e-1}})^{x_i} = \alpha_i^{p^{e-i-1}}, \quad 0 \leq i \leq e-1. \tag{11.6}$$

Zur Berechnung der Koeffizienten x_i hat man also e DL-Probleme in Gruppen der Ordnung p zu lösen.

Beispiel 11.5.3. Wie in Beispiel 11.3.1 lösen wir

$$5^x \equiv 3 \bmod 2017.$$

Die Gruppenordnung der primen Restklassengruppe mod 2017 ist

$$n = 2016 = 2^5 * 3^2 * 7.$$

Zuerst bestimmen wir $x(2) = x \bmod 2^5$. Wir erhalten $x(2)$ als Lösung der Kongruenz

$$(5^{3^2*7})^{x(2)} \equiv 3^{3^2*7} \bmod 2017.$$

Dies ergibt die Kongruenz

$$500^{x(2)} \equiv 913 \bmod 2017.$$

Um diese Kongruenz zu lösen, schreiben wir

$$x(2) = x_0(2) + x_1(2) * 2 + x_2(2) * 2^2 + x_3(2) * 2^3 + x_4(2) * 2^4.$$

Gemäß (11.6) ist $x_0(2)$ die Lösung von

$$2016^{x_0(2)} \equiv 1 \bmod 2017.$$

Man erhält $x_0(2) = 0$ und $\alpha_1 = \alpha_0 = 913 + 2017\mathbb{Z}$. Damit ist $x_1(2)$ Lösung von

$$2016^{x_1(2)} \equiv 2016 \bmod 2017.$$

Dies ergibt $x_1(2) = 1$ und $\alpha_2 = 1579 + 2017\mathbb{Z}$. Damit ist $x_2(2)$ Lösung von

$$2016^{x_2(2)} \equiv 2016 \bmod 2017.$$

Dies ergibt $x_2(2) = 1$ und $\alpha_3 = 1 + 2017\mathbb{Z}$. Damit ist $x_3(2) = x_4(2) = 0$. Insgesamt ergibt sich

$$x(2) = 6.$$

Nun berechnen wir

$$x(3) = x_0(3) + x_1(3) * 3.$$

Wir erhalten $x_0(3)$ als Lösung von

$$294^{x_0(3)} \equiv 294 \bmod 2017.$$

Dies ergibt $x_0(3) = 1$ und $\alpha_1 = 294 + 2017\mathbb{Z}$. Damit ist $x_1(3) = 1$ und

$$x(3) = 4.$$

Schließlich berechnen wir $x(7)$ als Lösung der Kongruenz

$$1879^{x(7)} \equiv 1879 \bmod 2017.$$

Also ist $x(7) = 1$. Wir erhalten dann x als Lösung der simultanen Kongruenz

$$x \equiv 6 \bmod 32, \quad x \equiv 4 \bmod 9, \quad x \equiv 1 \bmod 7.$$

Die Lösung ist $x = 1030$.

11.5.3 Gesamtalgorithmus und Analyse

Um die Gleichung (11.1) zu lösen, geht man im Algorithmus von Pohlig-Hellman folgendermaßen vor. Man berechnet die Werte $\gamma_p = \gamma^{n_p}$ und $\alpha_p = \alpha^{n_p}$ für alle Primteiler p von n. Anschließend werden die Koeffizienten $x_i(p)$ ermittelt für alle Primteiler p von n und $0 \leq i \leq e(p) - 1$. Dazu kann man den Algorithmus von Pollard oder den Babystep-Giantstep-Algorithmus von Shanks benutzen. Zuletzt wird der chinesische Restsatz benutzt, um den diskreten Logarithmus x zu konstruieren.

Der Aufwand, den der Pohlig-Hellman-Algorithmus zur Berechnung diskreter Logarithmen benötigt, kann folgendermaßen abgeschätzt werden.

Theorem 11.5.4. *Der Pohlig-Hellman-Algorithmus berechnet diskrete Logarithmen der Gruppe G unter Verwendung von $\mathrm{O}(\sum_{p||G|}(e(p)(\log|G| + \sqrt{p})) + (\log|G|)^2)$ Gruppenoperationen.*

Beweis. Wir benutzen dieselben Bezeichnungen wie im vorigen Abschnitt. Die Berechnung einer Ziffer von $x(p)$ für einen Primteiler p von $n = |G|$ erfordert $O(\log n)$ für die Potenzen und $O(\sqrt{p})$ für den Babystep-Giantstep-Algorithmus. Die Anzahl der Ziffern ist höchstens $e(p)$. Daraus folgt die Behauptung.

Theorem 11.5.3 zeigt, daß die Zeit, die der Pohlig-Hellman-Algorithmus zur Berechnung diskreter Logarithmen braucht, von der Quadratwurzel des größten Primteilers der Gruppenordnung dominiert wird. Wenn also der größte Primteiler der Gruppenordnung zu klein ist, kann man in der Gruppe leicht diskrete Logarithmen berechnen.

Beispiel 11.5.5. Die Zahl $p = 2 * 3 * 5^{278} + 1$ ist eine Primzahl. Ihre binäre Länge ist 649. Die Ordnung der primen Restklassengruppe mod p ist $p - 1 = 2 * 3 * 5^{278}$. Die Berechnung diskreter Logarithmen in dieser Gruppe ist sehr einfach, wenn man den Pohlig-Hellman-Algorithmus verwendet, weil der größte Primteiler der Gruppenordnung 5 ist. Darum kann man diese Primzahl p nicht im ElGamal-Verfahren benutzen.

11.6 Index-Calculus

Für die prime Restklassengruppe modulo einer Primzahl und genereller für die multiplikative Gruppe eines endlichen Körpers gibt es effizientere Methoden zur Berechnung diskreter Logarithmen, nämlich sogenannte Index-Calculus-Methoden. Sie sind eng verwandt mit Faktorisierungsverfahren wie dem Quadratischen Sieb und dem Zahlkörpersieb. Wir beschreiben hier die einfachste Version eines solchen Algorithmus.

11.6.1 Idee

Sei p eine Primzahl, g eine Primitivwurzel mod p und $a \in \{1, \ldots, p-1\}$. Wir wollen die Kongruenz

$$g^x \equiv a \bmod p \tag{11.7}$$

lösen. Dazu wählen wir eine Schranke B, bestimmen die Menge

$$F(B) = \{q \in \mathbb{P} : q \leq B\}.$$

Diese Menge ist die *Faktorbasis*. Eine ganze Zahl b heißt *B-glatt*, wenn in der Primfaktorzerlegung von b nur Primzahlen $q \leq B$ vorkommen.

Beispiel 11.6.1. Sei $B = 15$. Dann ist $F(B) = \{2, 3, 5, 7, 11, 13\}$. Die Zahl 990 ist 15-glatt. Ihre Primfaktorzerlegung ist nämlich $990 = 2 * 3^2 * 5 * 11$.

Wir gehen in zwei Schritten vor. Zuerst berechnen wir die diskreten Logarithmen für alle Faktorbasiselemente. Wir lösen also

$$g^{x(q)} \equiv q \bmod p \tag{11.8}$$

für alle $q \in F(B)$. Dann bestimmen wir einen Exponenten $y \in \{1, 2, \ldots, p-1\}$, für den $ag^y \bmod p$ B-glatt ist. Dann gilt also

$$ag^y \equiv \prod_{q \in F(B)} q^{e(q)} \bmod p \tag{11.9}$$

mit nicht negativen ganzen Exponenten $e(q)$, $q \in F(B)$. Aus (11.8) und (11.9) folgt

$$ag^y \equiv \prod_{q \in F(B)} q^{e(q)} \equiv \prod_{q \in F(B)} g^{x(q)e(q)} \equiv g^{\sum_{q \in F(B)} x(q)e(q)} \bmod p,$$

also

$$a \equiv g^{\sum_{q \in F(B)} x(q)e(q) - y} \bmod p.$$

Daher ist

$$x = (\sum_{q \in F(B)} x(q)e(q) - y) \bmod (p-1) \tag{11.10}$$

der gesuchte diskrete Logarithmus.

11.6.2 Diskrete Logarithmen der Faktorbasiselemente

Um die diskreten Logarithmen der Faktorbasiselemente zu berechnen, wählt man zufällige Elemente $z \in \{1, \ldots, p-1\}$ und berechnet $g^z \bmod p$. Man prüft, ob diese Zahlen B-glatt sind. Wenn ja, berechnet man die Zerlegung

$$g^z \bmod p = \prod_{q \in F(B)} q^{f(q,z)}.$$

Jeder Exponentenvektor $(f(q,z))_{q \in F(B)}$ heißt *Relation*.

Beispiel 11.6.2. Wir wählen $p = 2027$, $g = 2$ und bestimmen Relationen für die Faktorbasis $\{2, 3, 5, 7, 11\}$. Wir erhalten

$$\begin{aligned} 3 * 11 &= 33 \equiv 2^{1593} \bmod 2027 \\ 5 * 7 * 11 &= 385 \equiv 2^{983} \bmod 2027 \\ 2^7 * 11 &= 1408 \equiv 2^{1318} \bmod 2027 \\ 3^2 * 7 &= 63 \equiv 2^{293} \bmod 2027 \\ 2^6 * 5^2 &= 1600 \equiv 2^{1918} \bmod 2027. \end{aligned}$$

Wenn man viele Relationen gefunden hat, kann man für die diskreten Logarithmen folgendermaßen ein lineares Kongruenzensystem aufstellen. Unter Verwendung von (11.8) erhält man

$$g^z \equiv \prod_{q \in F(B)} q^{f(q,z)} \equiv \prod_{q \in F(B)} g^{x(q) f(q,z)} \equiv g^{\sum_{q \in F(B)} x(q) f(q,z)} \bmod p.$$

Daher ist

$$z \equiv \sum_{q \in F(B)} x(q) f(q, z) \bmod (p-1) \tag{11.11}$$

für alle z. Für jede Relation hat man also eine Kongruenz gefunden. Wenn man $n = |F(B)|$ viele Relationen gefunden hat, versucht man die diskreten Logarithmen $x(q)$ durch Verwendung des Gaußalgorithmus zu berechnen und zwar modulo jedes Primteilers l von $p-1$. Teilt ein Primteiler l die Ordnung $p-1$ in höherer Potenz, dann berechnet man die $x(q)$ modulo dieser Potenz. Dadurch wird die lineare Algebra etwas schwieriger. Danach berechnet man die $x(q)$ mit dem chinesischen Restsatz.

Beispiel 11.6.3. Wir setzen Beispiel 11.6.2 fort. Der Ansatz

$$q \equiv g^{x(q)} \bmod 2027, \quad q = 2, 3, 5, 7, 11$$

führt mit den Relationen aus Beispiel 11.6.2 zu dem Kongruenzensystem

$$\begin{aligned} x(3) + x(11) &\equiv 1593 \bmod 2026 \\ x(5) + x(7) + x(11) &\equiv 983 \bmod 2026 \\ 7x(2) + x(11) &\equiv 1318 \bmod 2026 \\ 2x(3) + x(7) &\equiv 293 \bmod 2026 \\ 6x(2) + 2x(5) &\equiv 1918 \bmod 2026. \end{aligned} \tag{11.12}$$

Da $2026 = 2 * 1013$ ist, und 1013 eine Primzahl ist, lösen wir jetzt das Kongruenzensystem mod 2 und mod 1013. Es ergibt sich

$$\begin{aligned} x(3) + x(11) &\equiv 1 \bmod 2 \\ x(5) + x(7) + x(11) &\equiv 1 \bmod 2 \\ x(2) + x(11) &\equiv 0 \bmod 2 \\ x(7) &\equiv 1 \bmod 2 \end{aligned} \tag{11.13}$$

Wir wissen bereits, daß $x(2) = 1$ ist, weil als Primitivwurzel $g = 2$ gewählt wurde. Daraus gewinnt man

$$x(2) \equiv x(5) \equiv x(7) \equiv x(11) \equiv 1 \bmod 2, \quad x(3) \equiv 0 \bmod 2. \tag{11.14}$$

Als nächstes berechnen wir die diskreten Logarithmen der Faktorbasiselemente mod 1013. Wieder ist $x(2) = 1$. Aus (11.12) erhalten wir

$$\begin{aligned} x(3) + x(11) &\equiv 580 \bmod 1013 \\ x(5) + x(7) + x(11) &\equiv 983 \bmod 1013 \\ x(11) &\equiv 298 \bmod 1013 \\ 2x(3) + x(7) &\equiv 293 \bmod 1013 \\ 2x(5) &\equiv 899 \bmod 1013 \end{aligned} \tag{11.15}$$

Wir erhalten $x(11) \equiv 298 \bmod 1013$. Um $x(5)$ auszurechnen, müssen wir 2 mod 1013 invertieren. Wir erhalten $2 * 507 \equiv 1 \bmod 1013$. Daraus ergibt sich $x(5) \equiv 956 \bmod 1013$. Aus der zweiten Kongruenz erhalten wir $x(7) \equiv 742 \bmod 1013$. Aus der ersten Kongruenz erhalten wir $x(3) \equiv 282 \bmod 1013$. Unter Berücksichtigung von (11.14) erhalten wir schließlich

$$x(2) = 1, x(3) = 282, x(5) = 1969, x(7) = 1755, x(11) = 1311.$$

Man verifiziert leicht, daß dies korrekt ist.

11.6.3 Individuelle Logarithmen

Sind die diskreten Logarithmen der Faktorbasiselemente berechnet, bestimmt man den diskreten Logarithmus von a zur Basis g, indem man ein $y \in \{1, \ldots, p-1\}$ zufällig bestimmt. Wenn $ag^y \bmod p$ B-glatt ist, wendet man (11.10) an. Andernfalls wählt man ein neues y.

Beispiel 11.6.4. Wir lösen

$$2^x \equiv 13 \bmod 2027.$$

Wir wählen $y \in \{1, \ldots, 2026\}$ zufällig, bis $13 * 2^y \bmod 2027$ nur noch Primfaktoren aus der Menge $\{2, 3, 5, 7, 11\}$ hat. Wir finden

$$2 * 5 * 11 = 110 \equiv 13 * 2^{1397} \bmod 2027.$$

Unter Verwendung von (11.10) ergibt sich $x = (1 + 1969 + 1311 - 1397) \bmod 2026 = 1884$.

11.6.4 Analyse

Man kann zeigen, daß der beschriebene Index-Calculus-Algorithmus die subexponentielle Laufzeit $L_p[1/2, c + o(1)]$ hat, wobei c eine Konstante ist, die von der technischen Umsetzung des Algorithmus abhängt. Die Analyse wird ähnlich durchgeführt wie die Analyse des quadratischen Siebs in Abschnitt 10.4.

11.7 Andere Algorithmen

Es gibt eine Reihe effizienterer Varianten des Index-Calculus-Algorithmus. Der zur Zeit effizienteste Algorithmus ist das Zahlkörpersieb. Es hat die Laufzeit $L_p[1/3, (64/9)^{1/3}]$. Das Zahlkörpersieb zur DL-Berechnung wurde kurz nach der Erfindung des Zahlkörpersiebs zur Faktorisierung entdeckt. Rekordberechnungen von diskreten Logarithmen in endlichen Körpern findet man unter [24]. So wurde am 2. Februar 2007 ein diskreter Logarithmus modulo einer 160-stelligen Primzahl berechnet. Die Primzahl ist $p = \lfloor 10159 * Pi \rfloor + 119849 = 31415926535897932384626433832795028841971693993751058209749445923078164062862089986280348253421170679821480865132823066470938446095505822317253594081284812372 99$ Auch $(p-1)/2$ ist eine Primzahl. Eine Primitivwurzel modulo p ist $g = 2$. Der diskrete Logarithmus modulo p zur Basis 2 von $y = \lfloor 10159 * e \rfloor = 27182818284590452353602874713526624977572470936999595749669676277240766303535475945713821785251664274274663919320030599218174135966290435729003342952605956307 38$ sollte berechnet werden. Hierbei ist e die Eulersche Konstante. Das Ergebnis ist $y = g^x$ mit $x = 82989716465034897051864680264075784402496146932312647219853184518689598402644834266625285046612688143761738165394262430753767931963671156105352608242351366559 6$.

So wie dem Zahlkörpersieb ist es vorher auch mit anderen Faktorisierungsalgorithmen gegangen. Durch Modifikation entstanden DL-Algorithmen, die fast genauso effizient sind. Obwohl dies bis jetzt nicht bewiesen werden konnte, hat sich immer wieder gezeigt, daß das DL-Problem in $(\mathbb{Z}/p\mathbb{Z})^*$, p eine Primzahl, nicht schwieriger zu lösen ist als das Faktorisierungsproblem für natürliche Zahlen. Daher sind kryptographische Algorithmen, die ihre Sicherheit aus der Schwierigkeit von DL-Problemen über $(\mathbb{Z}/p\mathbb{Z})^*$ beziehen, nicht sicherer als Verfahren, die auf Faktorisierungsproblemen beruhen. Will man Alternativen, braucht man DL-Probleme in anderen Gruppen, z.B. in der Punktgruppe von elliptischen Kurven über endlichen Körpern oder in der Klassengruppe von algebraischen Zahlkörpern.

Alle DL-Probleme, die im Kontext der Kryptographie von Bedeutung sind, sind auf Quantencomputern leicht lösbar. Das wurde von Shor in [74] gezeigt. Es nur noch nicht klar, ob und wann es Quantencomputer geben wird.

11.8 Verallgemeinerung des Index-Calculus-Verfahrens

Das Index-Calculus-Verfahren ist nur für die prime Restklassengruppe modulo einer Primzahl erklärt worden. Die Verfahren von Shanks, Pollard oder Pohlig-Hellman funktionieren aber in beliebigen endlichen Gruppen. Auch das Index-Calculus-Verfahren kann verallgemeinert werden. In beliebigen Gruppen braucht man eine Faktorbasis von Gruppenelementen. Man muß

zwischen den Gruppenelementen genügend Relationen finden, also Potenzprodukte, deren Wert die Eins in der Gruppe ist. Hat man die Relationen gefunden, kann man mit linearer Algebra die diskreten Logarithmen genauso berechnen, wie das oben beschrieben wurde. Die entscheidende Schwierigkeit ist die Bestimmung der Relationen. In primen Restklassengruppen kann man Relationen finden, weil man die Gruppenelemente in den Ring der ganzen Zahlen liften kann und dort die eindeutige Primfaktorzerlegung gilt. Für andere Gruppen wie z.B. die Punktgruppe elliptischer Kurven ist nicht bekannt, wie die Relationen gefunden werden können und daher ist in diesen Gruppen auch das Index-Calculus-Verfahren bis jetzt nicht anwendbar.

11.9 Übungen

Übung 11.9.1. Lösen Sie $3^x \equiv 693 \bmod 1823$ mit dem Babystep-Giantstep-Algorithmus.

Übung 11.9.2. Verwenden Sie den Babystep-Giantstep-Algorithmus, um den diskreten Logarithmus von 15 zur Basis 2 mod 239 zu berechnen.

Übung 11.9.3. Lösen Sie $a^x \equiv 507 \bmod 1117$ für die kleinste Primitivwurzel a mod 1117 mit dem Pohlig-Hellman-Algorithmus.

Übung 11.9.4. Verwenden Sie den Pohlig-Hellman-Algorithmus, um den diskreten Logarithmus von 2 zur Basis 3 mod 65537 zu berechnen.

Übung 11.9.5. Berechnen Sie mit dem Pollard-ρ-Algorithmus die Lösung von $g^x \equiv 15 \bmod 3167$ für die kleinste Primitivwurzel g mod 3167.

Übung 11.9.6. Verwenden Sie die Variante des Pollard-ρ-Algorithmus, die acht Tripel (β, x, y) speichert, um das DL-Problem $g^x \equiv 15 \bmod 3167$ für die kleinste Primitivwurzel g mod 3167 zu berechnen. Vergleichen Sie die Effizienz dieser Berechnung mit dem Ergebnis von Übung 11.9.5.

Übung 11.9.7. Berechnen Sie mit dem Index-Calculus-Algorithmus unter Verwendung der Faktorbasis $\{2, 3, 5, 7, 11\}$ die Lösung von $7^x \equiv 13 \bmod 2039$.

12. Kryptographische Hashfunktionen

In diesem Kapitel behandeln wir kryptographische Hashfunktionen. Solche Funktionen braucht man z.B. für digitale Signaturen. Im ganzen Kapitel ist Σ ein Alphabet.

12.1 Hashfunktionen und Kompressionsfunktionen

Unter einer *Hashfunktion* verstehen wir eine Abbildung

$$h : \Sigma^* \to \Sigma^n, \quad n \in \mathbb{N}.$$

Hashfunktionen bilden also beliebig lange Strings auf Strings fester Länge ab. Sie sind nie injektiv.

Beispiel 12.1.1. Die Abbildung, die jedem Wort $b_1 b_2 \dots b_k$ aus $\{0,1\}^*$ die Zahl $b_1 \oplus b_2 \oplus b_3 \oplus \cdots \oplus b_k$ zuordnet, ist eine Hashfunktion. Sie bildet z.B. 01101 auf 1 ab. Allgemein bildet sie einen String b auf 1 ab, wenn die Anzahl der Einsen in b ungerade ist und auf 0 andernfalls.

Hashfunktionen können mit Hilfe von *Kompressionsfunktionen* generiert werden. Eine Kompressionsfunktion ist eine Abbildung

$$h : \Sigma^m \to \Sigma^n, \quad n, m \in \mathbb{N}, \quad m > n.$$

Sie bildet Strings einer festen Länge auf Strings fester kürzerer Länge ab.

Beispiel 12.1.2. Die Abbildung, die jedem Wort $b_1 b_2 \dots b_m$ aus $\{0,1\}^m$ die Zahl $b_1 \oplus b_2 \oplus b_3 \oplus \cdots \oplus b_m$ zuordnet, ist eine Kompressionsfunktion, wenn $m > 1$ ist.

Hashfunktionen und Kompressionsfunktionen werden für viele Zwecke gebraucht, z.B. zum Anlegen von Wörterbüchern. Auch in der Kryptographie spielen sie eine wichtige Rolle. Kryptographische Hash- und Kompressionsfunktionen müssen Eigenschaften haben, die ihre sichere Verwendbarkeit garantieren. Diese Eigenschaften werden jetzt informell beschrieben. Dabei ist $h : \Sigma^* \to \Sigma^n$ eine Hashfunktion oder $h : \Sigma^m \to \Sigma^n$ eine Kompressionsfunktion. Den Definitionsbereich von h bezeichnen wir mit D. Es ist also

$D = \Sigma^*$, wenn h eine Hashfunktion ist, und es ist $D = \Sigma^m$, wenn h eine Kompressionsfunktion ist.

Um h in der Kryptographie verwenden zu können, verlangt man, daß der Wert $h(x)$ für alle $x \in D$ effizient berechenbar ist. Wir setzen dies im folgenden voraus.

Die Funktion h heißt *Einwegfunktion*, wenn es praktisch unmöglich ist, zu einem $s \in \Sigma^n$ ein $x \in D$ mit $h(x) = s$ zu finden. Was heißt "praktisch unmöglich"? Dies mathematisch zu beschreiben ist kompliziert. Man braucht dazu die Komplexitätstheorie. Das geht über den Rahmen dieses Buches hinaus. Wir geben uns mit einer intuitiven Beschreibung zufrieden. Jeder Algorithmus, der versucht, bei Eingabe von $s \in \Sigma^n$ ein x mit $h(x) = s$ zu berechnen, soll fast immer scheitern, weil er zuviel Zeit oder Platz verbraucht. Man kann nicht verlangen, daß der Algorithmus immer scheitert, weil er ja für einige x die Hashwerte $h(x)$ vorberechnet und gespeichert haben könnte. Es ist nicht bekannt, ob es Einwegfunktionen gibt. Es gibt aber Funktionen, die nach heutiger Kenntnis Einwegfunktionen sind. Ihre Funktionswerte sind leicht zu berechnen, aber es ist kein Algorithmus bekannt, der die Funktion schnell umkehren kann.

Beispiel 12.1.3. Ist p eine zufällig gewählte 1024-Bit-Primzahl und g eine Primitivwurzel mod p, dann ist nach heutiger Kenntnis die Funktion $f : \{0, 2, \ldots, p-1\} \to \{1, 2, \ldots, p-1\}$, $x \mapsto g^x \bmod p$ eine Einwegfunktion, weil kein effizientes Verfahren zur Berechnung diskreter Logarithmen bekannt ist (siehe Kapitel 11).

Eine *Kollision* von h ist ein Paar $(x, x') \in D^2$ von Strings, für die $x \neq x'$ und $h(x) = h(x')$ gilt. Alle Hashfunktionen und Kompressionsfunktionen besitzen Kollisionen, weil sie nicht injektiv sind.

Beispiel 12.1.4. Eine Kollision der Hashfunktion aus Beispiel 12.1.1 ist ein Paar verschiedener Strings, die beide eine ungerade Anzahl von Einsen haben, also z.B. $(111, 001)$.

Die Funktion h heißt *schwach kollisionsresistent*, wenn es praktisch unmöglich ist, für ein vorgegebenes $x \in D$ eine Kollision (x, x') zu finden. Nachfolgend findet sich ein Beispiel für die Verwendung einer schwach kollisionsresistenten Hashfunktion.

Beispiel 12.1.5. Alice möchte ein Verschlüsselungsprogramm auf ihrer Festplatte gegen unerlaubte Änderung schützen. Mit einer Hashfunktion $h : \Sigma^* \to \Sigma^n$ berechnet sie den Hashwert $y = h(x)$ dieses Programms x und speichert den Hashwert y auf ihrer persönlichen Chipkarte. Abends geht Alice nach Hause und nimmt ihre Chipkarte mit. Am Morgen kommt sie wieder in ihr Büro. Sie schaltet ihren Computer ein und will das Verschlüsselungsprogramm benutzen. Erst prüft sie aber, ob das Programm nicht geändert wurde. Sie berechnet den Hashwert $h(x)$ erneut und vergleicht das Resultat

y' mit dem Hashwert y, der auf ihrer Chipkarte gespeichert ist. Wenn beide Werte übereinstimmen, wurde das Programm x nicht geändert. Weil h schwach kollisionsresistent ist, kann nämlich niemand ein verändertes Programm x' erzeugen mit $y' = h(x') = h(x) = y$.

Beispiel 12.1.5 zeigt eine typische Verwendung von kollisionsresistenten Hashfunktionen. Sie erlauben die Überprüfung der *Integrität* eines Textes, Programms, etc., also der Übereinstimmung mit dem Original. Mit der Hashfunktion kann die Integrität der Originaldaten auf die Integrität eines viel kleineren Hashwertes zurückgeführt werden. Dieser kleinere Hashwert kann an einem sicheren Ort, z.B. auf einer Chipkarte, gespeichert werden.

Die Funktion h heißt *(stark) kollisionsresistent*, wenn es praktisch unmöglich ist, irgendeine Kollision (x, x') von h zu finden. In manchen Anwendungen muß man Hashfunktionen benutzen, die stark kollisionsresistent sind. Ein wichtiges Beispiel sind elektronische Signaturen, die im nächsten Kapitel beschrieben werden. Man kann zeigen, daß kollisionsresistente Hashfunktionen Einwegfunktionen sein müssen. Die Idee ist folgende: Angenommen, es gibt einen Algorithmus, der zu einem Bild y ein Urbild x ausrechnen kann. Dann wählt man einen Text x' zufällig. Man berechnet $y = h(x')$ und dazu ein Urbild x. Dann ist (x, x') eine Kollision von h, falls $x \neq x'$ ist.

12.2 Geburtstagsattacke

In diesem Abschnitt beschreiben wir eine einfache Attacke, nämlich die Geburtstagsattacke, auf eine Hashfunktion

$$h : \Sigma^* \to \Sigma^n.$$

Der Angriff gilt der starken Kollisionsresistenz von h. Er versucht also, eine Kollision von h zu finden.

Die Geburtstagsattacke besteht darin, so viele Hashwerte, wie man in der zur Verfügung stehenden Zeit berechnen und im vorhandenen Speicher speichern kann, zu erzeugen und zu speichern. Diese Werte werden sortiert und dann wird nach einer Kollision gesucht. Das Geburtstagsparadox (siehe Abschnitt 5.3) erlaubt die Analyse dieses Verfahrens. Die Hashwerte entsprechen den Geburtstagen. Wir nehmen an, daß wir Strings aus Σ^* zufällig wählen können, und daß die Verteilung der Hashwerte dabei die Gleichverteilung ist. In Abschnitt 5.3 wurde folgendes gezeigt: Wählt man k Argumente $x \in \Sigma^*$ zufällig aus, wobei

$$k \geq (1 + \sqrt{1 + (8 \ln 2)|\Sigma|^n})/2$$

ist, dann ist die Wahrscheinlichkeit dafür, daß zwei von ihnen denselben Hashwert haben, größer als $1/2$. Der Einfachheit halber nehmen wir an, daß $\Sigma = \{0, 1\}$ ist. Dann brauchen wir

$$k \geq f(n) = (1 + \sqrt{1 + (8 \ln 2)2^n})/2.$$

Folgende Tabelle zeigt einige Werte von $\log_2(f(n))$.

n	50	100	150	200
$\log_2(f(n))$	25.24	50.24	75.24	100.24

Wenn man also etwas mehr als $2^{n/2}$ viele Hashwerte bildet, findet die Geburtstagsattacke mit Wahrscheinlichkeit $\geq 1/2$ eine Kollision. Um die Geburtstagsattacke zu verhindern, muß man n so groß wählen, daß es unmöglich ist, $2^{n/2}$ Hashwerte zu berechnen und zu speichern. Heutzutage wählt man $n \geq 128$. Für die Hashfunktion im digitalen Signaturstandard wird sogar $n \geq 160$ verlangt.

12.3 Kompressionsfunktionen aus Verschlüsselungsfunktionen

Genausowenig wie es bekannt ist, ob es effiziente und sichere Verschlüsselungsverfahren wirklich gibt, weiß man, ob es kollisionsresistente Kompressionsfunktionen gibt. In der Praxis werden Kompressionsverfahren verwendet, deren Kollisionsresistenz bis jetzt nicht widerlegt wurde. Man kann Kompressionsfunktionen z.B. aus Verschlüsselungsfunktionen konstruieren. Dies wird nun beschrieben.

Wir benötigen ein Verschlüsselungsverfahren mit Klartextraum, Schlüsselraum und Schlüsseltextraum $\{0,1\}^n$ und Verschlüsselungsfunktionen $e_k : \{0,1\}^n \to \{0,1\}^n$, $k \in \{0,1\}^n$. Die Hashwerte haben die Länge n. Um die Geburtstagsattacke zu verhindern, muß daher $n \geq 128$ sein. Daher kann man in diesem Zusammenhang DES nicht benutzen.

Auf folgende Weisen kann man Kompressionsfunktionen

$$h : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$$

definieren:

$$\begin{aligned} h(k,x) &= e_k(x) \oplus x \\ h(k,x) &= e_k(x) \oplus x \oplus k \\ h(k,x) &= e_k(x \oplus k) \oplus x \\ h(k,x) &= e_k(x \oplus k) \oplus x \oplus k \end{aligned}$$

Ist das Verschlüsselungsverfahren sicher, so scheinen diese Kompressionsfunktionen auch sicher zu sein.

12.4 Hashfunktionen aus Kompressionsfunktionen

Wenn es kollisionsresistente Kompressionsfunktionen gibt, dann gibt es auch kollisionsresistente Hashfunktionen. R. Merkle hat nämlich ein Verfahren beschrieben, wie man aus einer kollisionsresistenten Kompressionsfunktion eine kollisionsresistente Hashfunktion machen kann. Dieses Verfahren beschreiben wir.

Sei

$$g : \{0,1\}^m \to \{0,1\}^n$$

eine Kompressionsfunktion und sei

$$r = m - n.$$

Weil g eine Kompressionsfunktion ist, gilt $r > 0$. Eine typische Situation ist die, daß $n = 128$ und $r = 512$ ist. Wir erläutern die Konstruktion für $r \geq 2$. Der Fall $r = 1$ bleibt dem Leser als Übung überlassen. Aus g will man eine Hashfunktion

$$h : \{0,1\}^* \to \{0,1\}^n$$

konstruieren. Sei also $x \in \{0,1\}^*$. Vor x wird eine minimale Anzahl von Nullen geschrieben, so daß die neue Länge durch r teilbar ist. An diesen String werden nochmal r Nullen angehängt. Jetzt wird die Binärentwicklung der Länge des originalen Strings x bestimmt. Ihr werden so viele Nullen vorangestellt, daß ihre Länge durch $r-1$ teilbar ist. Vor diese Binärentwicklung und vor jedes $(r-1)*j$-te Zeichen, $j = 1, 2, 3, \ldots$, wird eine Eins geschrieben. Dieser neue String wird wiederum an den vorigen angehängt. Der Gesamtstring wird in eine Folge

$$x = x_1 x_2 \ldots x_t, \quad x_i \in \{0,1\}^r, \quad 1 \leq i \leq t.$$

von Wörtern der Länge r zerlegt. Man beachte, daß alle Wörter, die aus der Binärentwicklung der Länge von x stammen, mit einer Eins beginnen.

Beispiel 12.4.1. Sei $r = 4$, $x = 111011$. Zuerst wird x in 00111011 verwandelt. An diesen String wird 0000 angehängt. Man erhält 001110110000. Die Originallänge von x ist 6. Die Binärentwicklung von 6 ist 110. Sie wird als 1110 geschrieben. Insgesamt haben wir jetzt den String 0011101100001110.

Der Hashwert $h(x)$ wird iterativ berechnet. Man setzt

$$H_0 = 0^n.$$

Das ist der String, der aus n Nullen besteht. Dann bestimmt man

$$H_i = g(H_{i-1} \circ x_i), \quad 1 \leq i \leq t.$$

Schließlich setzt man

$$h(x) = H_t.$$

Wir zeigen, daß h kollisionsresistent ist, wenn g kollisionsresistent ist. Wir beweisen dazu, daß man aus einer Kollision von h eine Kollision von g bestimmen kann.

Sei also (x, x') eine Kollision von h. Ferner seien $x_1, \ldots, x_t$, $x'_1, \ldots, x'_{t'}$ die zugehörigen Folgen von Blöcken der Länge r, die so wie oben beschrieben konstruiert sind. Die entsprechenden Folgen von Hashwerten seien $H_0, \ldots, H_t$, $H'_0, \ldots, H'_{t'}$.

Weil (x, x') eine Kollision ist, gilt $H_t = H'_{t'}$. Sei $t \leq t'$. Wir vergleichen jetzt H_{t-i} mit $H'_{t'-i}$ für $i = 1, 2, \ldots$. Angenommen, wir finden ein $i < t$ mit

$$H_{t-i} = H'_{t'-i}$$

und

$$H_{t-i-1} \neq H'_{t'-i-1}.$$

Dann gilt

$$H_{t-i-1} \circ x_{t-i} \neq H'_{t'-i-1} \circ x'_{t'-i}$$

und

$$g(H_{t-i-1} \circ x_{t-i}) = H_{t-i} = H'_{t'-i} = g(H'_{t'-i-1} \circ x'_{t'-i}).$$

Dies ist eine Kollision von g.

Sei nun angenommen, daß

$$H_{t-i} = H'_{t'-i} \quad 0 \leq i \leq t.$$

Unten zeigen wir, daß es einen Index i gibt mit $0 \leq i \leq t-1$ und

$$x_{t-i} \neq x'_{t'-i}.$$

Daraus folgt

$$H_{t-i-1} \circ x_{t-i} \neq H'_{t'-i-1} \circ x'_{t'-i}$$

und

$$g(H_{t-i-1} \circ x_{t-i}) = H_{t-i} = H'_{t'-i} = g(H'_{t'-i-1} \circ x'_{t'-i}).$$

Also ist wieder eine Kollision von g gefunden.

Wir zeigen jetzt, daß es einen Index i gibt mit $0 \leq i \leq t-1$ und

$$x_{t-i} \neq x'_{t'-i}.$$

Werden für die Darstellung der Länge von x weniger Wörter gebraucht als für die Darstellung der Länge von x', dann gibt es einen Index i, für den x_{t-i} nur aus Nullen besteht (das ist der String, der zwischen die Darstellung von x und seiner Länge geschrieben wurde) und für den $x'_{t'-i}$ eine führende 1 enthält (weil alle Wörter, die in der Darstellung der Länge von x' vorkommen, mit 1 beginnen).

Werden für die Darstellung der Längen von x und x' gleich viele Wörter gebraucht, aber sind diese Längen verschieden, dann gibt es einen Index i derart, daß x_{t-i} und $x'_{t'-i}$ in der Darstellung der Länge von x bzw. x' vorkommen und verschieden sind.

Sind die Längen von x und x' aber gleich, dann gibt es einen Index i mit $x_{t-i} \neq x'_{t'-i}$, weil x und x' verschieden sind.

Wir haben also gezeigt, wie man eine Kollision der Kompressionsfunktion aus einer Kollision der Hashfunktion gewinnt. Weil der Begriff "kollisionsresistent" aber nicht formal definiert wurde, haben wir dieses Ergebnis auch nicht als mathematischen Satz formuliert.

12.5 SHA-1

SHA-1 [71] ist heute ein sehr häufig benutzte Hashfunktion. Sie wird zum Beispiel im Digital Signature Standard (DSS) [31] verwendet. In diesem Abschnitt beschreiben wir SHA-1.

Sei $x \in \{0,1\}^*$. Sei die Länge $|x|$ von x, also die Anzahl der Bits in x, kleiner als 2^{64}. Der Hashwert von x wird folgendermaßen berechnet.

Im ersten Schritt wird x so ergänzt, daß die Länge von x ein Vielfaches von 512 ist. Dies geht so:

1. An x wird eine 1 angehängt: $x \leftarrow x \circ 1$.
2. An x werden so viele Nullen angehängt, daß $|x| = k \cdot 512 - 64$ ist. Die Anzahl der angehängten Nullen ist minimal mit dieser Eigenschaft.
3. Die Länge des originalen x wird als 64-Bit-Zahl geschrieben und an x angehängt.

Beispiel 12.5.1. Die Nachricht x sei

$$01100001\ 01100010\ 01100011\ 01100100\ 01100101.$$

Nach dem ersten Schritt der Ergänzung ist x

$$01100001\ 01100010\ 01100011\ 01100100\ 01100101\ 1.$$

Die Länge von x ist jetzt 41. Im zweiten Schritt werden 407 Nullen an x angehängt. Dann ist die Länge von x nämlich $448 = 512 - 64$. In Hexadezimalschreibweise ist x also

61626364 65800000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000

Die Länge des originalen x war 40. Als 64-Bit-Zahl geschrieben ist diese Länge

$$00000000\ 00000028.$$

Daher ist x schließlich

61626364	65800000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000028

Bei der Berechnung des Hashwertes werden die Funktionen

$$f_t : \{0,1\}^{32} \times \{0,1\}^{32} \times \{0,1\}^{32} \to \{0,1\}^{3}2$$

verwendet, die folgendermaßen spezifiziert ist

$$f_t(B,C,D) = \begin{cases} (B \wedge C) \vee (\neg B \wedge D) & \text{für } 0 \leq t \leq 19 \\ B \oplus C \oplus D & \text{für } 20 \leq t \leq 39 \\ (B \wedge C) \vee (B \wedge D) \vee (C \wedge D) & \text{für } 40 \leq t \leq 59 \\ B \oplus C \oplus D & \text{für } 60 \leq t \leq 79. \end{cases}$$

Hierbei bezeichnet $\wedge$ das bitweise logische "und", $\vee$ das bitweise logische "oder" und $\oplus$ das bitweise logische "xor". Außerdem werden die Konstanten

$$K_t = \begin{cases} 5A827999 & \text{für } 0 \leq t \leq 19 \\ 6ED9EBA1 & \text{für } 20 \leq t \leq 39 \\ 8F1BBCDC & \text{für } 40 \leq t \leq 59 \\ CA62C1D6 & \text{für } 60 \leq t \leq 79. \end{cases}$$

benutzt.

Der Hashwert wird jetzt so berechnet. Sei x ein nach den beschriebenen Regeln erweiterter Bitstring, dessen Länge durch 512 teilbar ist. Die Folge der 512-Bit-Wörter sei

$$x = M_1 M_2 M_3 \dots M_n.$$

Zuerst werden folgende 32-Bit-Wörter initialisiert: $H_0 = 67452301$, $H_1 = EFCDAB89$, $H_2 = 98BADCFE$, $H_3 = 10325476$, $H_4 = C3D2E1F0$.

Danach wird die folgende Prozedur für $i = 1, 2, \dots, n$ ausgeführt. Darin bedeutet $S^k(w)$ den zirkulären Links-Shift eines 16-Bitwortes w um k Bits. Außerdem bedeutet + die Addition zweier durch 16-Bit-Wörter dargestellter Zahlen mod 2^{16}.

1. Schreibe M_i als Folge $M_i = W_0 W_1 \dots W_{15}$ von 32-Bit-Wörtern.
2. Für $t = 16, 17, \dots, 79$ setze $W_t = S^1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16})$.
3. Setze $A = H_0$, $B = H_1$, $C = H_2$, $D = H_3$ und $E = H_4$.
4. Für $t = 0, 1, \dots, 79$ setze $T = S^5(A) + f_t(B,C,D) + E + W_t + K_t$, $E = D$, $D = C$, $C = S^{36}(B)$, $B = A$, $A = T$.
5. Setze $H_0 = H_0 + A$, $H_1 = H_1 + B$, $H_2 = H_2 + C$, $H_3 = H_3 + D$, $H_4 = H_4 + E$.

Der Hashwert ist

$$\text{SHA-1}(x) = H_0 H_1 H_2 H_3 H_4.$$

12.6 Andere Hashfunktionen

Neben SHA-1 werden in der Praxis auch andere Hashfunktionen benutzt, die ähnlich wie in Abschnitt 12.4 konstruiert sind. Modifikationen dieser Konstruktion beschleunigen die Auswertung. Tabelle 12.1 aus [52] enthält einige Charakteristika praktisch verwendeter Hashfunktionen.

Hashfunktion	Blocklänge	Relative Geschwindigkeit
MD4	128	1.00
MD5	128	0.68
RIPEMD-128	128	0.39
SHA-1	160	0.28
RIPEMD-160	160	0.24

Tabelle 12.1. Spezielle Hashfunktionen

Die Hashfunktion MD4 kann nicht mehr als kollisionsresistent gelten, weil Dobbertin [26] durch Berechnung von 2^{20} Hashwerten eine Kollision gefunden hat. Das Konstruktionsprinzip von MD4 wird aber in allen aufgeführten Hashfunktionen verwendet.

Auch MD5 muß mit Vorsicht verwendet werden, weil Dobbertin [25] gezeigt hat, dass die verwendete Kompressionsfunktion nicht kollisionsfrei ist. Alle aufgeführten Hashfunktionen sind aber sehr effizient. Eine genaue Beschreibung von RIPEMD-128, RIPEMD-160 und SHA-1 findet sich im Standard ISO/IEC 10118.

12.7 Eine arithmetische Kompressionsfunktion

Wir haben bereits erwähnt, daß nachweislich kollisionsresistente Kompressionsfunktionen nicht bekannt sind. Es gibt aber eine Kompressionsfunktion, die jedenfalls dann kollisionsresistent ist, wenn es schwer ist, diskrete Logarithmen in $(\mathbb{Z}/p\mathbb{Z})^*$ zu berechnen. Sie wurde von Chaum, van Heijst und Pfitzmann erfunden. Wir erläutern diese Kompressionsfunktion.

Sei p eine Primzahl, $q = (p-1)/2$ ebenfalls eine Primzahl und a eine Primitivwurzel mod p und b zufällig gewählt in $\{1, 2, \ldots, p-1\}$. Betrachte folgende Funktion:

$$h : \{0, 1, \ldots, q-1\}^2 \to \{1, \ldots, p-1\}, \quad (x_1, x_2) \mapsto a^{x_1} b^{x_2} \bmod p \qquad (12.1)$$

Dies ist zwar keine Kompressionsfunktion wie in Abschnitt 12.1. Aber weil $q = (p-1)/2$ ist, bildet die Funktion Bitstrings (x_1, x_2), deren binäre Länge ungefähr doppelt so groß ist wie die binäre Länge von p, auf Bitstrings ab, die nicht länger sind als die binäre Länge von p. Man kann aus h leicht eine Kompressionsfunktion im Sinne von Abschnitt 12.1 konstruieren.

Beispiel 12.7.1. Sei $q = 11$, $p = 23$, $a = 5$, $b = 4$. Dann ist $h(5, 10) = 5^5 \cdot 4^{10} \bmod 23 = 20 \cdot 6 \bmod 23 = 5$.

Eine Kollision von h ist ein Paar $(x, x') \in \{0, 1, \ldots, q-1\}^2 \times \{0, 1, \ldots, q-1\}^2$ mit $x \neq x'$ und $h(x) = h(x')$. Wir zeigen nun, daß jeder, der leicht eine Kollision von h finden kann, genauso leicht den diskreten Logarithmus von b zur Basis a modulo p ausrechnen kann.

Sei also (x, x') eine Kollision von h, $x = (x_1, x_2)$, $x' = (x_3, x_4)$, $x_i \in \{0, 1, \ldots, q-1\}$, $1 \leq i \leq 4$. Dann gilt

$$a^{x_1} b^{x_2} \equiv a^{x_3} b^{x_4} \bmod p$$

woraus

$$a^{x_1 - x_3} \equiv b^{x_4 - x_2} \bmod p$$

folgt. Bezeichne mit y den diskreten Logarithmus von b zur Basis a modulo p. Dann hat man also

$$a^{x_1 - x_3} \equiv a^{y(x_4 - x_2)} \bmod p.$$

Da a eine Primitivwurzel modulo p ist, impliziert diese Kongruenz

$$x_1 - x_3 \equiv y(x_4 - x_2) \bmod (p-1) = 2q. \tag{12.2}$$

Diese Kongruenz hat eine Lösung y, nämlich den diskreten Logarithmus von b zur Basis a. Das ist nur möglich, wenn $d = \gcd(x_4 - x_2, p-1)$ ein Teiler von $x_1 - x_3$ ist (siehe Übung 3.23.11). Nach Wahl von x_2 und x_4 ist $|x_4 - x_2| < q$. Da $p - 1 = 2q$ ist, folgt daraus

$$d \in \{1, 2\}.$$

Ist $d = 1$, so hat (12.2) eine eindeutige Lösung modulo $p-1$. Man kann also sofort den diskreten Logarithmus als kleinste nicht negative Lösung dieser Kongruenz bestimmen. Ist $d = 2$, so hat die Kongruenz zwei verschiedene Lösungen mod $p-1$ und man kann durch Ausprobieren herausfinden, welche die richtige ist.

Die Kompressionsfunktion aus (12.1) ist also kollisionsresistent, solange die Berechnung diskreter Logarithmen schwierig ist. Die Kollisionsresistenz wurde damit auf ein bekanntes Problem der Zahlentheorie reduziert. Man kann daher der Meinung sein, daß h sicherer ist als andere Kompressionsfunktionen. Weil die Auswertung von h aber modulare Exponentiationen erfordert, ist h auch sehr ineffizient und daher nur von theoretischem Interesse.

12.8 Message Authentication Codes

Kryptographische Hashfunktionen erlauben es zu überprüfen, ob eine Datei verändert wurde. Sie erlauben es aber nicht festzustellen, von wem eine Nachricht kommt. Will man die Authentizität einer Nachricht überprüfbar machen, so kann man parametrisierte Hashfunktionen verwenden.

Definition 12.8.1. *Eine* parametrisierte Hashfunktion *ist eine Familie* $\{h_k : k \in \mathcal{K}\}$ *von Hashfunktionen. Hierbei ist* $\mathcal{K}$ *eine Menge. Sie heißt* Schlüsselraum *von* h.

Eine parametrisierte Hashfunktion heißt auch *Message Authentication Code*, kurz MAC.

Beispiel 12.8.2. Sei

$$g : \{0,1\}^* \to \{0,1\}^4$$

eine Hashfunktion. Dann kann man daraus folgendermaßen einen MAC mit Schlüsselraum $\{0,1\}^4$ machen:

$$h_k : \{0,1\}^* \to \{0,1\}^4, \quad x \mapsto g(x) \oplus k.$$

Das folgende Beispiel zeigt, wie man MACs benutzen kann.

Beispiel 12.8.3. Die Professorin Alice sendet per Email an das Prüfungsamt eine Liste der Matrikelnummern aller Studenten, die den Schein zur Vorlesung "Einführung in die Kryptographie" erhalten haben. Diese Liste braucht Alice nicht geheimzuhalten. Sie hängt ja sogar öffentlich aus. Aber das Prüfungsamt muß sicher sein, daß die Nachricht tatsächlich von Alice kommt. Dazu wird ein MAC $\{h_k : k \in \mathcal{K}\}$ benutzt. Alice und das Prüfungsamt tauschen einen geheimen Schlüssel $k \in \mathcal{K}$ aus. Mit ihrer Liste x schickt Alice auch den Hashwert $y = h_k(x)$ an das Prüfungsamt. Bob, der Sachbearbeiter im Prüfungsamt, kann seinerseits den Hashwert $y' = h_k(x')$ der erhaltenen Nachricht x' berechnen. Er akzeptiert die Nachricht, wenn $y = y'$ ist.

Damit das Verfahren, das in Beispiel 12.8.3 beschrieben wurde, sicher ist, muß der MAC *fälschungsresistent* sein. Das bedeutet, daß es unmöglich ist, ohne Kenntnis von k ein Paar $(x, h_k(x))$ zu generieren.

Ein Standardverfahren, MACs zu konstruieren, besteht darin, den Eingabetext x mit einem symmetrischen Verfahren im CBC-Mode zu verschlüsseln, alle Schlüsseltextblöcke bis auf den letzten wegzuwerfen und den letzten als Hashwert zu verwenden. Auf weitere Details wollen wir hier verzichten und verweisen auf [52].

12.9 Übungen

Übung 12.9.1. Konstruieren sie eine Funktion, die eine Einwegfunktion ist, falls das Faktorisierungsproblem für natürliche Zahlen schwer zu lösen ist.

Übung 12.9.2. Für eine Permutation π in S_3 sei e_π die Bitpermutation für Bitstrings der Länge 3. Bestimmen Sie für jedes $\pi \in S_3$ die Anzahl der Kollisionen der Kompressionsfunktion $h_\pi(x) = e_\pi(x) \oplus x$.

Übung 12.9.3. Betrachten Sie die Hashfunktion $h : \{0,1\}^* \to \{0,1\}^*$, $k \mapsto \lfloor 10000(k(1+\sqrt{5})/2) \bmod 1) \rfloor$, wobei die Strings k mit den durch sie dargestellten natürlichen Zahlen identifiziert werden und $r \bmod 1 = r - \lfloor r \rfloor$ ist für eine positive reelle Zahl r.

1. Bestimmen Sie die die maximale Länge der Bilder.
2. Geben Sie eine Kollision dieser Hashfunktion an.

Übung 12.9.4. Erläutern Sie die Konstruktion einer Hashfunktion aus einer Kompressionsfunktion aus Abschnitt 12.4 für $r = 1$.

13. Digitale Signaturen

13.1 Idee

Elektronische Dokumente will man nicht nur verschlüsseln, sondern auch signieren können. Das macht man mit digitalen Signaturen. Diese Signaturen haben eine ähnliche Funktion wie gewöhnliche Unterschriften.

Wenn Alice ein Dokument handschriftlich signiert, kann jeder, der das Dokument sieht und die Unterschrift von Alice kennt, verifizieren, daß das Dokument tatsächlich von Alice unterschrieben wurde. Alice kann später nicht mehr bestreiten, das Dokument signiert zu haben. Die Unterschrift kann z.B. als Beweis in Gerichtsverfahren herangezogen werden, daß Alice das Dokument zur Kenntnis genommen und ihm zugestimmt hat.

Auch viele elektronische Dokumente müssen signiert werden. Das gilt z.B. für Kaufverträge im Internet, elektronische Banktransaktionen und verbindliche Emails. Digitale Signaturen müssen aber anders realisiert werden als handschriftliche. Man kann nicht einfach eine Unterschrift unter das elektronische Dokument setzen, weil diese Unterschrift leicht kopiert werden kann. Bei handschriftlichen Signaturen ist das schwieriger. Werden sie kopiert, kann das nachgewiesen werden.

Die Idee, die der Realisierung digitaler Signaturen zugrunde liegt, ist eng mit der Public-Key-Verschlüsselung verwandt. Will Alice elektronisch signieren, braucht sie dazu einen privaten, geheimen Schlüssel d und einen öffentlichen Schlüssel e. Der private Schlüssel ist sicher gespeichert, z.B. auf einer Chipkarte. Der öffentliche Schlüssel liegt in einem Verzeichnis, zu dem jeder Zugang hat, das aber nicht verändert werden kann.

Wenn Alice ein Dokument m signieren will, berechnet sie aus dem Dokument und ihrem privaten Schlüssel d die digitale Signatur $s(d, m)$ des Dokumentes. Unter Verwendung des öffentlichen Schlüssels kann dann jeder verifizieren, daß die Signatur korrekt ist.

Wir diskutieren in den folgenden Abschnitten zuerst die Sicherheit von Signaturverfahren. Dann beschreiben wir verschiedene Realisierungen digitaler Signaturen. Die Verfahren werden jeweils in drei Schritten beschrieben: Schlüsselerzeugung, Signatur, Verifikation. Anschließend wird die Effizienz und die Sicherheit diskutiert. Die Sicherheitsdiskussion erläutert die Schwierigkeit bekannter Angriffe. Für weitergehende Sicherheitsdiskussionen, wie sie in Abschnitt 13.2 angedeutet wurden, wird auf [7] verwiesen.

Wir bezeichnen mit Σ ein Alphabet.

13.2 Sicherheit

In Abschnitt 9.2 wurde die Sicherheit von Public-Key-Verschlüsselungsverfahren diskutiert. Wir behandeln die Sicherheit von Algorithmen für digitale Signaturen analog.

13.2.1 Sicherheit des privaten Schlüssels

Ein Signaturverfahren kann nur sicher sein, wenn es unmöglich ist, in vertretbarer Zeit aus den öffentlich verfügbaren Informationen, also insbesondere aus den öffentlichen Schlüsseln, die privaten Schlüssel der Nutzer zu berechnen. Bei den heute bekannten Signaturverfahren wird das dadurch gewährleistet, daß gewisse Berechnungsprobleme der Zahlentheorie schwer zu lösen sind. Welche Berechnungsprobleme das sind, werden wir bei der Beschreibung der einzelnen Signaturverfahren erklären. Ob diese zahlentheoretischen Probleme immer schwierig bleiben, ist nicht bekannt. Eine gute Idee kann ein bewährtes Signaturverfahren unsicher machen. Es ist zum Beispiel bekannt, daß Quantencomputer alle gängigen Signaturverfahren unsicher machen (siehe [74]). Es ist nur nicht bekannt, ob und wann solche Computer wirklich gebaut werden können. Es ist daher unbedingt nötig, Sicherheitsinfrastrukturen so zu konstruieren, daß die verwendeten kryptographischen Basistechniken leicht ausgetauscht werden können. Das wird z.B. in dem Projekt FlexiPKI [32] versucht.

13.2.2 No-Message-Attacks

Es muß nicht das Ziel eines Angreifers sein, geheime Signaturschlüssel zu finden. Der Angreifer kann versuchen, ohne Kenntnis des geheimen Signaturschlüssels eine gültige Signatur einer Nachricht zu berechnen, für das der legitime Signierer keine Signatur erstellt hat. Man nennt das eine *existentielle Fälschung*. Er geht dazu so vor:

1. Der Angreifer besorgt sich den Verifikationsschlüssel von Alice.
2. Unter Verwendung des Verifikationsschlüssels berechnet der Angreifer eine Nachricht x und eine gültige Alice-Signatur s für x.

Der Angreifer kann das Dokument, für das er eine Signatur berechnet, also in Abhängigkeit vom öffentlichen Schlüssel berechnen. Er braucht es nicht vorzuwählen. Das Ergebnis ist eine gültige Signatur für irgendeinen gültigen Text. Ein solcher Angriff heißt No-Message-Attack im Gegensatz zu den Chosen-Message-Attacks, die im nächsten Abschnitt beschrieben werden.

Natürlich kann ein Angreifer einfach eine Signatur raten. Mit einer sehr geringen Wahrscheinlichkeit ist er dann erfolgreich. Ein Signaturverfahren ist sicher gegen No-Message-Attacks, wenn kein Angreifer den beschriebenen Angriff mit nicht vernachlässigbarer Wahrscheinlichkeit erfolgreich durchführen kann.

13.2.3 Chosen-Message-Attacks

Sicherheit gegen No-Message-Attacks genügt nicht. Es ist nämlich möglich, dass der Angreifer schon andere gültige Signaturen kennt und diese verwendet, um neue Signaturen zu erzeugen. Solche Angriffe werden in den Abschnitten 13.3.4 und 13.5.6 beschrieben. Es ist sogar manchmal möglich, daß der Angreifer sich zunächst selbst gewählte Dokumente signieren läßt bevor er eine Signatur fälscht. Das wird im nächsten Beispiel erläutert.

Beispiel 13.2.1. Ein Webserver will nur legitimierten Nutzern Zugang gewähren. Meldet sich ein Nutzer an, generiert der Webserver eine Zufallszahl und fordert den Nutzer auf, diese Zufallszahl zu signieren. Ist die Signatur gültig, wird der Zugang gewährt. Andernfalls nicht. Ein Angreifer kann sich als Webserver ausgeben und sich selbst gewählte Dokumente signieren lassen. Die Signaturen kann er entweder direkt verwenden oder er benutzt sie, um neue Signaturen zu erzeugen.

Ein solcher Angriff funktioniert so:

1. Der Angreifer besorgt sich den Verifikationsschlüssel von Alice.
2. Unter Verwendung des Verifikationsschlüssels berechnet der Angreifer ein Dokument x und eine gültige Alice-Signatur s für x. Während der Berechnung von x und s sind kann sich der Angreifer jederzeit Nachrichten seiner Wahl von Alice signieren lassen. Die Nachrichten müssen nur verschieden von x sein.

Dieser Angriff heißt Chosen-Message-Attack. Der Angreifer kann nämlich die Auswahl der Dokumente, die Alice für ihn signiert, an die Berechnung, die er durchführt, anpassen. Auch hier kann ein Angreifer versuchen, eine passende Signatur zu raten. Er kann also immer mit einer geringen Wahrscheinlichkeit erfolgreich sein. Ein Signaturverfahren ist sicher gegen Adaptive-Chosen-Message-Attacks, wenn kein Angreifer den beschriebenen Angriff mit nicht vernachlässigbarer Wahrscheinlichkeit erfolgreich durchführen kann.

13.3 RSA-Signaturen

In Abschnitt 9.3 wurde das älteste Public-Key-Verschlüsselungsverfahren beschrieben, das RSA-Verfahren. Dieses Verfahren kann man auch zur Erzeugung digitaler Signaturen verwenden. Die Idee ist ganz einfach. Alice signiert

ein Dokument m, indem sie ihr Entschlüsselungsverfahren auf das Dokument anwendet, also $s = m^d \bmod n$ berechnet, wobei n der RSA-Modul und d der Entschlüsselungsexponent ist. Bob verifiziert die Signatur, indem er darauf das Verschlüsselungsverfahren anwendet, also $s^e \bmod n$ berechnet. Ergibt sich daraus das Dokument m, so ist die Signatur verifiziert. Es ist nämlich nicht bekannt, wie man ein s berechnen kann, für das $m = s^e \bmod n$ gilt, wenn man den privaten Schlüssel d nicht kennt. Dies ist das Prinzip der RSA-Signatur, die jetzt im Detail beschrieben wird.

13.3.1 Schlüsselerzeugung

Die Schlüsselerzeugung funktioniert genauso wie beim RSA-Verschlüsselungsverfahren. Alice wählt also zufällig zwei große Primzahlen p, q und einen Exponenten e mit $1 < e < (p-1)(q-1)$ und $\gcd(e,(p-1)(q-1)) = 1$. Sie berechnet $n = pq$ und $d \in \mathbb{Z}$ mit $1 < d < (p-1)(q-1)$ und $de \equiv 1 \bmod (p-1)(q-1)$. Ihr öffentlicher Schlüssel ist (n,e) und ihr geheimer Schlüssel ist d.

13.3.2 Erzeugung der Signatur

Wir erklären, wie Alice eine Zahl $m \in \{0,1,\ldots,n-1\}$ signiert. Eine solche Zahl m kann z.B. eine kurze Nachricht bedeuten. Später werden längere Nachrichten durch eine Hashfunktion auf Zahlen in der Menge $\{0,1,\ldots,n-1\}$ abgebildet und danach signiert (siehe Abschnitt 13.3.6).

Um die Zahl m zu signieren, berechnet Alice den Wert

$$s = m^d \bmod n. \tag{13.1}$$

Die Signatur ist s. Diese Signiermethode ist noch verbesserungsbedürftig, weil sie Gefahren birgt. Diese Gefahren und Vorbeugungsmaßnahmen werden unten beschrieben.

13.3.3 Verifikation

Bob will die Signatur s verifizieren. Er besorgt sich den öffentlichen Schlüssel (n,e) von Alice und berechnet

$$m = s^e \bmod n. \tag{13.2}$$

Daß dies stimmt, wenn die Signatur s korrekt gebildet wurde, folgt aus Theorem 9.3.4. Was hat Bob durch die Verifikation erreicht? Er kennt jetzt den Text m. Da er ihn aus der Signatur s gewonnen hat, weiß er, daß s die Signatur von m ist. Er braucht m vorher nicht zu kennen. Er ist sich auch sicher, daß Alice die Signatur s erzeugt hat. Die Signatur s ist nämlich die RSA-Entschlüsselung von m. Solange das RSA-Verschlüsselungsverfahren sicher ist, kann niemand zu der vorgegebenen Nachricht m die Signatur s ohne Kenntnis des privaten Schlüssels d berechnen.

Diese Verifikation kann jeder durchführen, der den öffentlichen Schlüssel von Alice kennt. Will Bob etwa einem Richter gegenüber beweisen, daß Alice den Text m wirklich signiert hat, braucht er nur die Signatur s und den öffentlichen Schlüssel e von Alice vorzulegen. Der Richter kann dann die Verifikation $m = s^e \bmod n$ selber vornehmen.

Beispiel 13.3.1. Alice wählt $p = 11$, $q = 23$, $e = 3$. Daraus ergibt sich $n = 253$, $d = 147$. Der öffentliche Schlüssel von Alice ist $(253, 3)$. Ihr geheimer Schlüssel ist 147.

Alice will an einem Geldautomaten Geld abheben und den Betrag von 111 DM signieren. Dazu berechnet sie $s = 111^{147} \bmod 253 = 89$. Der Geldautomat erhält $s = 89$ und berechnet $m = s^3 \bmod 253 = 111$. Damit weiß der Geldautomat, daß Alice 111 DM abheben will und er kann das auch Dritten gegenüber beweisen.

13.3.4 Angriffe

So, wie die Erzeugung der RSA-Signatur bisher beschrieben wurde, bestehen eine Reihe von Gefahren.

Zu Beginn der Verifikation besorgt sich Bob den öffentlichen Schlüssel (n, e) von Alice. Wenn es dem Angreifer Oskar gelingt, Bob seinen eigenen öffentlichen Schlüssel als den Schlüssel von Alice unterzuschieben, kann er danach Signaturen erzeugen, die Bob als Signaturen von Alice anerkennt. Es ist also wichtig, daß Bob den authentischen öffentlichen Schlüssel von Alice hat. Er muß sich von der Authentizität des öffentlichen Schlüssels von Alice überzeugen können. Dazu werden Trustcenter verwendet, die später beschrieben werden.

Ein anderer Angriff funktioniert so: Oskar wählt eine Zahl $s \in \{0, \ldots, n-1\}$. Dann behauptet er, s sei eine RSA-Signatur von Alice. Wer diese Signatur verifizieren will, berechnet $m = s^e \bmod n$ und glaubt, Alice habe m signiert. Wenn m ein sinnvoller Text ist, wurde Alice damit die Signatur dieses sinnvollen Textes untergeschoben. Das ist eine No-Message-Attack.

Beispiel 13.3.2. Wie in Beispiel 13.3.1 wählt Alice $p = 11$, $q = 23$, $e = 3$. Daraus ergibt sich $n = 253$, $d = 147$. Der öffentliche Schlüssel von Alice ist $(253, 3)$. Ihr geheimer Schlüssel ist 147.

Oskar möchte vom Konto von Alice Geld abheben. Er schickt einfach $s = 123$ an den Geldautomaten. Der Geldautomat berechnet $m = 123^3 \bmod 253 = 52$. Er glaubt, daß Alice 117 DM abheben will. Tatsächlich hat Alice die 117 DM aber nie unterschrieben. Oskar hat ja nur eine zufällige Unterschrift gewählt.

Eine weitere Gefahr beim Signieren mit RSA kommt daher, daß das RSA-Verfahren multiplikativ ist. Sind m_1, $m_2 \in \{0, \ldots, n-1\}$ und sind $s_1 = m_1^d \bmod n$ und $s_2 = m_2^d \bmod n$ die Signaturen von m_1 und m_2, dann ist

$$s = s_1 s_2 \bmod n = (m_1 m_2)^d \bmod n$$

die Signatur von $m = m_1 m_2$. Aus zwei gültigen Signaturen kann man also leicht eine dritte gültige Signatur generieren. Nutzt ein Angreifer die Multiplikativität aus, kann er mit einer Chosen-Message-Attack jede Signatur fälschen. Soll nämlich die Nachricht $m \in \{0, \ldots, n-1\}$ signiert werden, dann wählt der Angreifer eine von m verschiedene Nachricht $m_1 \in \{0, \ldots, n-1\}$ mit $\gcd(m_1, n) = 1$. Dann berechnet er

$$m_2 = m m_1^{-1} \bmod n,$$

wobei m_1^{-1} das Inverse von m_1 mod n ist. Der Angreifer läßt sich die beiden Nachrichten m_1 und m_2 signieren. Er erhält die Signaturen s_1 und s_2 und kann daraus die Signatur $s = s_1 s_2 \bmod n$ von m berechnen. Das RSA-Signaturverfahren, so wie es bis jetzt beschrieben wurde, ist also nicht sicher gegen Chosen-Message-Angriffe.

In den folgenden Abschnitten beschreiben wir Vorkehrungen gegen die beschriebenen Gefahren.

13.3.5 Signatur von Texten mit Redundanz

Zwei der im vorigen Abschnitt beschriebenen Angriffe werden unmöglich, wenn nur Texte $m \in \{0, 1, \ldots, n-1\}$ signiert werden, deren Binärdarstellung von der Form $w \circ w$ ist mit $w \in \{0,1\}^*$. Die Binärdarstellung besteht also aus zwei gleichen Hälften. Der wirklich signierte Text ist die erste Hälfte, nämlich w. Aber signiert wird $w \circ w$. Bei der Verifikation wird $m = s^e \bmod n$ bestimmt, und dann wird geprüft, ob der signierte Text die Form $w \circ w$ hat. Wenn nicht, wird die Signatur zurückgewiesen.

Werden nur Texte von der Form $w \circ w$ signiert, kann der Angreifer Oskar die beschriebene existentielle Fälschung nicht mehr anwenden. Er müßte nämlich eine Signatur $s \in \{0, 1, \ldots, n-1\}$ auswählen, für die die Binärentwicklung von $m = s^e \bmod n$ die Form $w \circ w$ hat. Es ist unbekannt, wie man eine solche Signatur s ohne Kenntnis des privaten Schlüssels bestimmen kann.

Auch die Multiplikativität von RSA kann nicht mehr ausgenutzt werden. Es ist nämlich äußerst unwahrscheinlich, daß $m = m_1 m_2 \bmod n$ eine Binärentwicklung von der Form $w \circ w$ hat, wenn das für beide Faktoren der Fall ist.

Die Funktion

$$R : \{0,1\}^* \to \{0,1\}^*, w \mapsto R(w) = w \circ w,$$

die zur Erzeugung der speziellen Struktur verwendet wurde, heißt *Redundanzfunktion*. Es können natürlich auch andere Redundanzfunktionen verwendet werden.

13.3.6 Signatur mit Hashwert

Bisher wurde beschrieben, wie Texte m, deren binäre Länge nicht größer ist als die binäre Länge des RSA-Moduls, mit dem RSA-Verfahren signiert werden. Der signierte Text wird bei der Verifikation aus der Signatur ermittelt.

Wenn Alice einen beliebig langen Text x signieren will, verwendet sie eine öffentlich bekannte, kollisionsresistente Hashfunktion

$$h : \{0,1\}^* \to \{0,\ldots,n-1\}.$$

Weil h kollisionsresistent ist, muß h auch Einwegfunktion sein. In der Praxis konstruiert man h mit einer gängigen kollisionsresistenten Hashfunktion, die z.B. einen 160-Bit-String erzeugt. Darauf wendet man eine Expansionsfunktion an. Dies ist in dem Standard PKCS #1 genau festgelegt (siehe [60]).

Die Signatur eines Textes x ist

$$s = h(x)^d \bmod n.$$

Aus dieser Signatur läßt sich nur der Hashwert des Textes rekonstruieren, aber nicht der Text selbst. Daher wird zur Verifikation der Text x benötigt. Nachdem Alice die Signatur s des Dokumentes x erzeugt hat, sendet sie diese zusammen mit x an Bob. Bob berechnet $m = s^e \bmod n$ und vergleicht diese Zahl mit dem Hashwert $h(x)$. Weil die Hashfunktion h öffentlich bekannt ist, kann Bob $h(x)$ berechnen. Stimmen m und $h(x)$ überein, so ist die Signatur gültig, andernfalls nicht.

Dieses Verfahren macht die existentielle Fälschung aus Abschnitt 13.3.4 unmöglich. Angenommen, der Fälscher Oskar wählt eine Signatur s. Dann muß er auch einen Text x produzieren, und muß (x, s) an Bob schicken. Bob berechnet $m = s^e \bmod n$ und prüft, ob $m = h(x)$ ist. Oskar muß also x so bestimmen, daß $h(x) = m$ gilt. Der Text x ist also Urbild von m und m liegt durch s fest. Oskar muß also ein Urbild x von m unter h bestimmen. Das kann er nicht, weil h eine Einwegfunktion ist.

Auch die Ausnutzung der Multiplikativität von RSA (siehe Abschnitt 13.3.4) wird unmöglich. Weil h eine Einwegfunktion ist, ist es nämlich unmöglich, einen Text x zu finden mit $h(x) = m = m_1 m_2 \mod n$.

Schließlich kann Oskar den Text x, der von Alice signiert wurde, auch nicht durch einen anderen Text x' mit der gleichen Signatur ersetzen. Das Paar (x, x') wäre dann nämlich eine Kollision von h, und h wurde als kollisionsresistent angenommen.

13.3.7 Wahl von p und q

Wer den öffentlichen RSA-Modul faktorisieren kann, ist in der Lage, den geheimen Exponenten d zu bestimmen und damit RSA-Signaturen von Alice zu fälschen. Daher müssen p und q so gewählt werden, daß n nicht zerlegt werden kann. Die Wahl von p und q wurde schon für das RSA-Verschlüsselungsverfahren in Abschnitt 9.3.6 beschrieben.

13.3.8 Sichere Verwendung

Eine Variante des RSA-Signaturverfahrens, die unter geeigneten Voraussetzungen sicher gegen Chosen-Message-Attacks ist, findet man in [8].

13.4 Signaturen aus Public-Key-Verfahren

Die Konstruktion des RSA-Signaturverfahrens läßt sich mit jedem deterministischen Public-Key-Verschlüsselungsverfahren nachahmen, sofern dieses eine Zusatzbedingung erfüllt. Seien E und D die Ver- und Entschlüsselungsfunktionen und seien e und d der öffentliche und der geheime Schlüssel. Dann muß für jeden Klartext m die Gleichung

$$m = E(D(m, d), e)$$

gelten. Die Rolle von Ver- und Entschlüsselung muß also vertauschbar sein. Diese Bedingung ist für das RSA-Verfahren erfüllt, weil

$$(m^d)^e \equiv (m^e)^d \equiv m \bmod n$$

gilt.

Verwendet man ein Public-Key-Verfahren, das obige Bedingung erfüllt, so ist $s = D(m, d)$ die Signatur eines Textes m. Bei der Verifikation wird $m = E(D(m, d), e)$ bestimmt. Bei der Konstruktion des Verfahrens muß man wie beim RSA-Verfahren eine Redundanz- oder eine Hashfunktion verwenden. Wie dies genau zu geschehen hat, ist z.B. in der Norm ISO/IEC 9796 [39] festgelegt.

Neben dem RSA-Verfahren kann man z.B. auch das Rabin-Verschlüsselungsverfahren benutzen, um ein Signaturverfahren zu konstruieren. Dies geschieht in den Übungen.

13.5 ElGamal-Signatur

Wie das ElGamal-Verschlüsselungsverfahren (siehe Abschnitt 9.6) bezieht auch das ElGamal-Signaturverfahren seine Sicherheit aus der Schwierigkeit, diskrete Logarithmen in $(\mathbb{Z}/p\mathbb{Z})^*$ zu berechnen, wobei p eine Primzahl ist. Da im ElGamal-Verschlüsselungsverfahren die Verschlüsselung und die Entschlüsselung nicht einfach vertauschbar sind, kann man daraus nicht direkt ein Signaturverfahren machen, jedenfalls nicht so, wie in Abschnitt 13.4 beschrieben. Das ElGamal-Signaturverfahren muß daher anders konstruiert werden als das entsprechende Verschlüsselungsverfahren.

13.5.1 Schlüsselerzeugung

Die Schlüsselerzeugung ist genauso wie im Verschlüsselungsverfahren (siehe Abschnitt 9.6.1). Alice generiert eine zufällige große Primzahl p und eine Primitivwurzel g mod p. Alice wählt a zufällig in der Menge $\{1, 2, \ldots, p-2\}$. Sie berechnet $A = g^a \bmod p$. Ihr privater Schlüssel ist a. Ihr öffentlicher Schlüssel ist (p, g, A).

13.5.2 Erzeugung der Signatur

Alice signiert einen Text $x \in \{0,1\}^*$. Sie benutzt eine öffentlich bekannte, kollisionsresistente Hashfunktion

$$h : \{0,1\}^* \to \{1, 2, \ldots, p-2\}.$$

Alice wählt eine Zufallszahl $k \in \{1, 2, \ldots, p-2\}$, die zu $p-1$ teilerfremd ist. Sie berechnet

$$r = g^k \bmod p, \quad s = k^{-1}(h(x) - ar) \bmod (p-1). \tag{13.3}$$

Hierin bedeutet k^{-1} das Inverse von k modulo $p-1$. Die Signatur ist das Paar (r, s). Weil eine Hashfunktion benutzt wurde, benötigt ein Verifizierer neben der Signatur auch den Text x. Er kann x nicht aus s ermitteln.

13.5.3 Verifikation

Bob, der Verifizierer, verschafft sich den öffentlichen Schlüssel (p, g, A) von Alice. Genauso wie beim RSA-Signaturverfahren muß er sich von der Authentizität des öffentlichen Schlüssels von Alice überzeugen. Er verifiziert, daß

$$1 \le r \le p-1$$

ist. Falls diese Bedingung nicht erfüllt ist, wird die Signatur zurückgewiesen. Andernfalls überprüft Bob, ob die Kongruenz

$$A^r r^s \equiv g^{h(x)} \bmod p \tag{13.4}$$

erfüllt ist. Wenn ja, akzeptiert Bob die Signatur. Sonst nicht.

Wir zeigen, daß die Verifikation funktioniert. Falls s gemäß (13.3) konstruiert ist, folgt

$$A^r r^s \equiv g^{ar} g^{kk^{-1}(h(x)-ar)} \equiv g^{h(x)} \bmod p \tag{13.5}$$

wie gewünscht. Ist umgekehrt (13.4) für ein Paar (r, s) erfüllt, und ist k der diskrete Logarithmus von r zur Basis g, so gilt nach Korollar 3.9.3

$$g^{ar+ks} \equiv g^{h(x)} \bmod p.$$

Weil g eine Primitivwurzel mod p ist, folgt daraus

$$ar + ks \equiv h(x) \bmod p - 1.$$

Ist k zu $p-1$ teilerfremd, folgt daraus (13.3). Es gibt dann keine andere Art, die Signatur s zu konstruieren.

Beispiel 13.5.1. Wie in Beispiel 9.6.1 wählt Alice $p = 23$, $g = 7$, $a = 6$ und berechnet $A = g^a \bmod p = 4$. Ihr öffentlicher Schlüssel ist dann $(p = 23, g = 7, A = 4)$. Ihr geheimer Schlüssel ist $a = 6$.

Alice will ein Dokument x mit Hashwert $h(x) = 7$ signieren. Sie wählt $k = 5$ und erhält $r = 17$. Das Inverse von k mod $(p - 1 = 22)$ ist $k^{-1} = 9$. Daher ist $s = k^{-1}(h(x) - ar) \bmod (p - 1) = 9 * (7 - 6 * 17) \bmod 22 = 3$. Die Signatur ist $(17, 3)$.

Bob will diese Signatur verifizieren. Er berechnet $A^r r^s \bmod p = 4^{17} * 17^3 \bmod 23 = 5$. Er berechnet auch $g^{h(x)} \bmod p = 7^7 \bmod 23 = 5$. Damit ist die Signatur verifiziert.

13.5.4 Die Wahl von p

Wer diskrete Logarithmen mod p berechnen kann, ist in der Lage, den geheimen Schlüssel a von Alice zu ermitteln und damit Signaturen zu fälschen. Dies ist die einzige bekannte allgemeine Methode, ElGamal-Signaturen zu erzeugen. Man muß also p so groß wählen, daß die Berechnung diskreter Logarithmen nicht möglich ist. Nach heutigem Kenntnisstand wählt man die Primzahl p so, daß sie wenigstens 768 Bits, für lange Gültigkeiten der Signaturen besser 1024 Bits lang ist. Es ist außerdem nötig, p so zu wählen, daß die bekannten DL-Algorithmen kein leichtes Spiel haben. Man muß z.B. vermeiden, daß $p - 1$ nur kleine Primfaktoren hat. Sonst können diskrete Logarithmen mod p nämlich mit dem Algorithmus von Pohlig-Hellman (siehe Abschnitt 11.5) berechnet werden. Man muß auch vermeiden, daß p für das Zahlkörpersieb besonders geeignet ist. Dies ist der Fall, wenn es ein irreduzibles Polynom vom Grad 5 gibt, das sehr kleine Koeffizienten und eine Nullstelle mod p hat. Wie bereits in Abschnitt 9.3.6 beschrieben, ist es am besten, p zufällig zu wählen.

Es gibt aber eine spezielle Situation, die vermieden werden muß. Wenn $p \equiv 3 \bmod 4$, die Primitivwurzel g ein Teiler von $p-1$ ist und wenn die Berechnung diskreter Logarithmen in der Untergruppe von $(\mathbb{Z}/p\mathbb{Z})^*$ der Ordnung g möglich ist, dann kann das ElGamal-Signaturverfahren gebrochen werden (siehe Übung 13.10.5). Diskrete Logarithmen in dieser Untergruppe kann man jedenfalls dann berechnen (mit dem Algorithmus von Pohlig-Hellman-Shanks-Pollard), wenn g nicht zu groß ist (siehe Abschnitt 11.5). Da die Primzahl p häufig so gewählt wird, daß $p - 1 = 2q$ für eine ungerade Primzahl q gilt, ist die Bedingung $p \equiv 3 \bmod 4$ oft erfüllt. Um obige Attacke zu vermeiden, wählt man also eine Primitivwurzel g, die $p - 1$ nicht teilt.

13.5.5 Die Wahl von k

Wir zeigen, daß aus Sicherheitsgründen für jede neue Signatur ein neuer Exponent k gewählt werden muß. Dies ist ja bei zufälliger Wahl von k garantiert.

Angenommen, die Signaturen s_1 und s_2 der Texte x_1 und x_2 werden mit demselben Exponenten k erzeugt. Dann ist der Wert $r = g^k \bmod p$ für beide Signaturen gleich. Also gilt

$$s_1 - s_2 \equiv k^{-1}(h(x_1) - h(x_2)) \bmod (p-1).$$

Hieraus kann man die Zahl k bestimmen, wenn $h(x_1) - h(x_2)$ invertierbar modulo $p-1$ ist. Aus $k, s_1, r, h(x_1)$ kann man den geheimen Schlüssel a von Alice berechnen. Es ist nämlich

$$s_1 = k^{-1}(h(x_1) - ar) \bmod (p-1)$$

und daher

$$a \equiv r^{-1}(h(x_1) - ks_1) \bmod (p-1).$$

13.5.6 Existentielle Fälschung

Für die Sicherheit des Verfahrens ist es erforderlich, daß tatsächlich eine Hashfunktion verwendet wird und nicht die Nachricht x direkt signiert wird.

Wenn die Nachricht x ohne Hashfunktion signiert wird, lautet die Verifikationskongruenz

$$A^r r^s \equiv g^x \bmod p.$$

Wir zeigen, wie man r, s, x wählen kann, damit diese Kongruenz erfüllt ist. Man wählt zwei ganze Zahlen u, v mit $\gcd(v, p-1) = 1$. Dann setzt man

$$r = g^u A^v \bmod p, \quad s = -rv^{-1} \bmod (p-1), \quad x = su \bmod (p-1).$$

Mit diesen Werten gilt die Kongruenz

$$A^r r^s \equiv A^r g^{su} A^{sv} \equiv A^r g^{su} A^{-r} \equiv g^x \bmod p.$$

Das ist die Verifikationskongruenz.

Bei Verwendung einer kryptographischen Hashfunktion kann man auf die beschriebene Weise nur Signaturen von Hashwerten erzeugen. Man kann aber dazu nicht den passenden Klartext zurückgewinnen, wenn die Hashfunktion eine Einwegfunktion ist.

Wie beim RSA-Verfahren kann man das beschriebene Problem auch durch Einführung von Redundanz lösen.

Auch die Bedingung $1 \le r \le p-1$ ist wichtig, um existentiellen Betrug zu vermeiden. Wird diese Größenbeschränkung nicht von r gefordert, so kann man aus bekannten Signaturen neue Signaturen konstruieren. Sei (r, s) die

Signatur einer Nachricht x. Sei x' ein weiterer Text, der signiert werden soll. Oskar berechnet

$$u = h(x')h(x)^{-1} \bmod (p-1).$$

Hierbei wird vorausgesetzt, daß $h(x)$ invertierbar ist mod $p-1$. Oskar berechnet weiter

$$s' = su \bmod (p-1)$$

und mit Hilfe des chinesischen Restsatzes ein r' mit

$$r' \equiv ru \bmod (p-1), \quad r' \equiv r \bmod p. \tag{13.6}$$

Die Signatur von x' ist (r', s'). Wir zeigen, daß die Verifikation mit dieser Signatur funktioniert. Tatsächlich gilt

$$A^{r'}(r')^{s'} \equiv A^{ru}r^{su} \equiv g^{u(ar+ks)} \equiv g^{h(x')} \bmod p.$$

Wir zeigen auch, daß $r' \geq p$ gilt, also den Test $1 \leq r' \leq p-1$ verletzt. Einerseits gilt

$$1 \leq r \leq p-1, \quad r \equiv r' \bmod p. \tag{13.7}$$

Andererseits ist

$$r' \equiv ru \not\equiv r \bmod p-1. \tag{13.8}$$

Das liegt daran, daß $u \equiv h(x')h(x)^{-1} \not\equiv 1 \bmod p-1$ gilt, weil h kollisionsresistent ist. Aus (13.8) folgt $r \neq r'$ und aus (13.7) folgt also $r' \geq p$.

13.5.7 Effizienz

Die Erzeugung einer ElGamal-Signatur erfordert eine Anwendung des erweiterten euklidischen Algorithmus zur Berechnung von $k^{-1} \bmod p-1$ und eine modulare Exponentiation mod p zur Berechnung von $r = g^k \bmod p$. Beide Berechnungen können sogar als Vorberechnungen durchgeführt werden. Sie hängen nicht von dem zu signierenden Text ab. Führt man eine solche Vorberechnung durch, muß man die Resultate aber sicher speichern können. Die aktuelle Signatur erfordert dann nur noch zwei modulare Multiplikationen und ist damit extrem schnell.

Die Verifikation einer Signatur erfordert drei modulare Exponentiationen. Das ist deutlich teurer als die Verifikation einer RSA-Signatur. Man kann die Verifikation aber beschleunigen, wenn man die Kongruenz

$$g^{-h(x)}A^r r^s \equiv 1 \bmod p$$

verifiziert und die modularen Exponentiationen auf der linken Seite simultan durchführt. Dies wird in Abschnitt 3.13 erläutert. Aus Satz 3.13.1 folgt, daß die Verifikation höchstens $5+t$ Multiplikationen und $t-1$ Quadrierungen mod p erfordert, wobei t die binäre Länge von p ist. Das ist nur unwesentlich mehr als eine einzige Potenzierung.

13.5.8 Sichere Verwendung

Eine Variante des ElGamal-Signaturverfahrens, die unter geeigneten Voraussetzungen sicher gegen Chosen-Message-Attacks ist, findet man in [61].

13.5.9 Verallgemeinerung

Wie das ElGamal-Verschlüsselungsverfahren läßt sich auch das ElGamal-Signaturverfahren in beliebigen zyklischen Gruppen implementieren, deren Ordnung bekannt ist. Das ist ein großer Vorteil des Verfahrens. Die Implementierung des Verfahrens kann aus obiger Beschreibung leicht abgeleitet werden. Natürlich müssen dieselben Sicherheitsvorkehrungen getroffen werden wie beim ElGamal-Signaturverfahren in $(\mathbb{Z}/p\mathbb{Z})^*$.

13.6 Der Digital Signature Algorithm (DSA)

Der Digital Signature Algorithm (DSA) wurde 1991 vom US-amerikanischen National Institute of Standards and Technology (NIST) vorgeschlagen und später vom NIST zum Standard erklärt. Der DSA ist eine effizientere Variante des ElGamal-Verfahrens. Im DSA wird die Anzahl der modularen Exponentiationen bei der Verifikation von drei auf zwei reduziert und die Länge der Exponenten ist nur 160 Bit. Außerdem ist die Parameterwahl viel genauer vorgeschrieben als beim ElGamal-Verfahren.

13.6.1 Schlüsselerzeugung

Alice erzeugt eine Primzahl q mit

$$2^{159} < q < 2^{160}.$$

Die Binärentwicklung dieser Primzahl hat also genau die Länge 160. Alice wählt als nächstes eine große Primzahl p mit folgenden Eigenschaften

- $2^{511+64t} < p < 2^{512+64t}$ für ein $t \in \{0, 1, \ldots, 8\}$,
- die zuerst gewählte Primzahl q ist ein Teiler von $p - 1$.

Die binäre Länge von p liegt also zwischen 512 und 1024 und sie ist ein Vielfaches von 64. Die Binärentwicklung von p besteht damit aus 8 bis 16 Bitstrings der Länge 64. Die Bedingung $q \mid (p-1)$ impliziert, daß die Gruppe $(\mathbb{Z}/p\mathbb{Z})^*$ Elemente der Ordnung q besitzt (siehe Theorem 3.21.1).

Als nächstes wählt Alice eine Primitivwurzel $x \bmod p$ und berechnet

$$g = x^{(p-1)/q} \bmod p.$$

Die Ordnung von $g + p\mathbb{Z}$ ist also q. Zuletzt wählt Alice eine Zahl a zufällig in der Menge $\{1, 2, \ldots, q-1\}$ und berechnet

$$A = g^a \bmod p.$$

Der öffentliche Schlüssel von Alice ist (p, q, g, A). Ihr geheimer Schlüssel ist a. Man beachte, daß $A+p\mathbb{Z}$ zu der von $g+p\mathbb{Z}$ erzeugten Untergruppe (der Ordnung q) gehört. Die Untergruppe hat ungefähr 2^{160} Elemente. Die Ermittlung des geheimen Schlüssels erfordert die Berechnung diskreter Logarithmen in dieser Untergruppe. Wir werden im Abschnitt über die Sicherheit des Verfahrens darauf eingehen, wie schwierig die Berechnung diskreter Logarithmen in dieser Untergruppe ist.

13.6.2 Erzeugung der Signatur

Alice will den Text x signieren. Sie verwendet eine öffentlich bekannte kollisionsresistente Hashfunktion

$$h : \{0,1\}^* \to \{1, 2, \ldots, q-1\}.$$

Sie wählt eine Zufallszahl $k \in \{1, 2, \ldots, q-1\}$, berechnet

$$r = (g^k \bmod p) \bmod q \tag{13.9}$$

und setzt

$$s = k^{-1}(h(x) + ar) \bmod q. \tag{13.10}$$

Hierbei ist k^{-1} das Inverse von k modulo q. Die Signatur ist (r, s).

13.6.3 Verifikation

Bob will die Signatur (r, s) des Textes x verifizieren. Er beschafft sich den authentischen öffentlichen Schlüssel (p, q, g, A) von Alice und die öffentlich bekannte Hashfunktion h. Bob verifiziert, daß

$$1 \le r \le q-1 \text{ und } 1 \le s \le q-1 \tag{13.11}$$

gilt. Ist eine dieser Bedingungen verletzt, ist die Signatur ungültig. Bob weist sie zurück. Andernfalls verifiziert Bob, daß

$$r = ((g^{(s^{-1}h(x)) \bmod q} A^{(rs^{-1}) \bmod q}) \bmod p) \bmod q \tag{13.12}$$

gilt. Ist die Signatur gemäß (13.9) und (13.10) korrekt konstruiert, so ist (13.12) tatsächlich erfüllt. Dann gilt nämlich

$$g^{(s^{-1}h(x)) \bmod q} A^{(rs^{-1}) \bmod q} \equiv g^{s^{-1}(h(x)+ra)} \equiv g^k \bmod p,$$

woraus (13.12) unmittelbar folgt.

13.6.4 Effizienz

Das DSA-Verfahren ist sehr eng mit dem ElGamal-Signaturverfahren verwandt. Wie im ElGamal-Signaturverfahren kann man die Erzeugung von Signaturen durch Vorberechnungen wesentlich beschleunigen.

Die Verifikation profitiert von zwei Effizienzsteigerungen. Sie benötigt nur noch zwei modulare Exponentiationen mod p, während im ElGamal-Verfahren drei Exponentiationen nötig waren. Dies ist aber nicht so bedeutend angesichts der Möglichkeit, die Verifikation durch simultane Exponentiation auszuführen (siehe die Abschnitte 13.5.7 und 3.13). Bedeutender ist, daß die Exponenten in allen modularen Exponentiationen nur 160 Bit lang sind, während im ElGamal-Verfahren die Exponenten genauso lang sind wie der Modul p, also wenigstens 512 Bits. Diese Verkürzung beschleunigt die Berechnungen natürlich erheblich.

13.6.5 Sicherheit

Wie im ElGamal-Verfahren muß für jede neue Signatur ein neues k gewählt werden (siehe Abschnitt 13.5.5). Außerdem ist die Verwendung einer Hashfunktion und die Überprüfung der Bedingungen in (13.11) unbedingt nötig. Wird keine Hashfunktion verwendet oder die erste Bedingung nicht überprüft, ergibt sich die Möglichkeit der existentiellen Fälschung wie in Abschnitt 13.5.6 beschrieben.

Wenn Oskar diskrete Logarithmen in der von $g + p\mathbb{Z}$ erzeugten Untergruppe H von $(\mathbb{Z}/p\mathbb{Z})^*$ berechnen kann, so ist er in der Lage, den geheimen Schlüssel a von Alice zu bestimmen und damit systematisch Signaturen zu fälschen. Das ist bis heute auch die einzige bekannte Möglichkeit, DSA-Signaturen zu fälschen. Einen wesentlichen Effizienzvorteil bezieht das DSA-Verfahren daraus, daß die Berechnungen nicht mehr in der gesamten Gruppe $(\mathbb{Z}/p\mathbb{Z})^*$ ausgeführt werden, sondern in der wesentlich kleineren Untergruppe H. Es stellt sich also die Frage, ob dieses DL-Problem einfacher ist als das allgemeine DL-Problem.

Grundsätzlich sind zwei Möglichkeiten bekannt, diskrete Logarithmen zu berechnen.

Man kann Index-Calculus-Verfahren in $\mathbb{Z}/p\mathbb{Z}$ anwenden (siehe Abschnitt 11.6). Es ist aber nicht bekannt, wie solche Verfahren einen Vorteil daraus ziehen können, daß ein diskreter Logarithmus in einer Untergruppe zu berechnen ist. Tatsächlich ist der Berechnungsaufwand genauso groß, als wenn man einen diskreten Logarithmus berechnen würde, dessen Basis eine Primitivwurzel modulo p ist.

Man kann auch generische Verfahren verwenden, die in allen endlichen abelschen Gruppen funktionieren. Die brauchen aber wesentlich länger als Index-Calculus-Verfahren. Die besten bekannten Verfahren von Shanks (siehe Abschnitt 11.3) oder Pollard (siehe Abschnitt 11.4) benötigen mehr als $\sqrt{q}$

Operationen in $(\mathbb{Z}/p\mathbb{Z})^*$, um diskrete Logarithmen in der Untergruppe H der Ordnung q zu berechnen. Da $q > 2^{159}$ ist, sind dies also wenigstens 2^{79} Operationen, was heutzutage noch unmöglich erscheint.

13.7 Das Merkle-Signaturverfahren

Bis jetzt wurden Signaturverfahren beschrieben, deren Sicherheit auf der Schwierigkeit beruht, natürliche Zahlen in ihre Primfaktoren zu zerlegen oder diskrete Logarithmen in geeigneten Gruppen zu berechnen. Es ist aber unklar, ob diese Probleme in der Zukunft schwierig bleiben. So hat 1994 Peter Shor [75] bewiesen, dass Quantencomputer in Polynomzeit natürliche Zahlen in ihre Primfaktoren zerlegen können und in allen relevanten Gruppen diskrete Logarithmen berechnen können. Bis heute sind aber noch keine hinreichend großen Quantencomputer realisierbar. Erste Prototypen existieren. Mit einem solchen Prototypen konnte die Zahl 15 in ihre Primfaktoren zerlegt werden [81]. Physiker arbeiten mit Hochdruck daran, Technologien zu entwickeln, die die Konstruktion großer Quantencomputer erlauben. Viele Physiker sind zuversichtlich, dass dies erfolgreich sein wird. Aufgrund der großen praktischen Bedeutung von Signaturverfahren, zum Beispiel für die Authentisierung von Software-Updates in mobilen Computern, ist es daher nötig, über Alternativen zu den heute gebräuchlichen Signaturverfahren nachzudenken. Eine solche Alternative ist das Merkle-Signaturverfahren. Es wurde 1979, also gleichzeitig mit dem RSA-Verfahren, von Ralph Merkle in seiner Dissertation vorgestellt. Die Sicherheit des Merkle-Signaturverfahren beruht lediglich auf der Kollisionsresistenz einer kryptographischen Hashfunktion. Eine solche Hashfunktion benötigt jedes Signaturverfahren, das statt großer Dokumente deutlich kleinerer Hashwerte signiert. Die Sicherheitsvoraussetzungen des Merkle-Verfahrens sind also minimal. Da das Merkle-Verfahren jede kryptographischen Hashfunktion verwenden kann, entsteht eine enorme Flexibilität. Sollte eine kryptographischen Hashfunktion unsicher werden, kann man sie durch eine andere ersetzen und erhält eine neue Instanz des Merkle-Signaturverfahren.

Trotz dieser Flexibilität konnte sich das Merkle-Verfahren in der Praxis nicht durchsetzen. Das liegt daran, dass das Merkle-Verfahren gegenüber dem RSA-Verfahren und den anderen heute verwendeten Signaturverfahren erhebliche Effizienznachteile hat. Neuere Forschungen haben aber zu sehr effizienten Varianten des Merkle-Verfahrens geführt. Dies wird in Abschnitt 13.9.6 beschrieben. Aufgrund dieser Verbesserungen und aufgrund des technologischen Fortschritts sind die effizienten Varianten des Merkle-Verfahren heute sehr wichtigste Kandidaten für zukünftige Signatur-Verfahren.

Das Merkle-Verfahren verwendet ein Einmal-Signaturverfahren (One-Time Signature Scheme, OTS). Bei einem Einmal-Signaturverfahren benutzt der Signierer einen geheimen Signierschlüssel, um ein Dokument zu

signieren. Dieser Signierschlüssel kann aber nur für eine Signatur verwendet werden. Benutzt man ihn mehrfach, wird das Einmal-Signaturverfahren unsicher. Der Verifizierer benutzt den entsprechenden Verifikationsschlüssel, um die Signatur zu verifizieren. Für jede neue Signatur ist also ein neuer Signierschlüssel und ein neuer Verifikationsschlüssel nötig. Das macht Einmal-Signaturverfahren sehr unpraktisch. Würde zum Beispiel ein Webserver ein solches Verfahren verwenden, um sich seinen Benutzern gegenüber zu authentisieren, dann müsste er eine Unzahl von Signierschlüsseln verwenden; für jeden Authentisierungsvorgang einen neuen. Die Nutzer müssten alle Verifikationsschlüssel kennen.

Die Idee von Merkle ist, einen Hash-Baum zu verwenden, um die Gültigkeit von sehr vielen Verifikationsschlüsseln auf die Gültigkeit eines einzigen öffentlichen Schlüssels zurückzuführen. In dem Webserver-Szenario müssen die Nutzer also nur noch einen einzigen öffentlichen Schlüssel speichern. Weitere Verbesserungen, die in Abschnitt 13.9.6 angedeutet werden, erlauben es auch dem Signierer, nur einen einzigen Signierschlüssel zu speichern.

Der nächste Abschnitt beschreibt zunächst eine Einmal-Signaturverfahren. Danach erläutert er die Reduktion vieler Verifikationsschlüssel auf einen einzigen öffentlichen Schlüssel, nämlich die Wurzel eines Hash-Baums. Schließlich fasst er noch die Optimierungen des Merkle-Verfahrens kurz zusammen.

13.8 Das Lamport-Diffie Einmal-Signaturverfahren

Dieser Abschnitt beschreibt das Lamport-Diffie-Einmal-Signaturverfahren [44]. Dieses Verfahren verwendet eine kryptographische Hashfunktion

$$H : \{0,1\}^* \to \{0,1\}^n, n \in \mathbb{N}.$$

13.8.1 Schlüsselerzeugung

Der Signaturschlüssel ist ein String

$$x = \big(x(0,1), x(1,1), x(0,2), x(1,2), \ldots, x(0,n), x(1,n)\big) \in \{0,1\}^{2n}.$$

Der zugehörige Verifikationsschlüssel ist der String

$$\begin{aligned} y &= \big(y(0,1), y(1,1), y(0,2), y(1,2), \ldots, y(0,n), y(1,n)\big) \\ &= \big(H(x(0,1)), H(x(1,1)), H(x(0,2)), H(x(1,2)), \ldots, H(x(0,n), H(x(1,n))\big). \end{aligned}$$

13.8.2 Erzeugung der Signatur

Der Signierer berechnet die Signatur eines Dokuments $d \in \{0,1\}^*$ folgendermaßen. Zunächst bestimmt er den Hashwert $H(d) = (H_1, \ldots, H_n)$. Die Signatur ist dann einfach die Folge

$$s = (s_1, \ldots, s_n) = (x(H_1, 1)), \ldots, x(H_n, n)).$$

In der Signatur wird also der geheime Signaturschlüssel teilweise veröffentlicht. Die Signatur ist eine Folge von Bitstrings der Länge n. Der i-te Bitstring ist $x(0, i)$, wenn das i-te Bit im Hashwert des signierten Dokuments 0 ist und $x(1, i)$ andernfalls.

13.8.3 Verifikation

Auch der Verifizierer berechnet den Hashwert $H(d) = (H_1, \ldots, H_n)$. Er kennt den Verifikationsschlüssel y. Er akzeptiert die Signatur, wenn

$$(H(s_1), \ldots, H(s_n)) = (y(H_1, 1), \ldots, y(H_n, n))$$

und weist sie andernfalls zurück.

Beispiel 13.8.1. Wir nehmen an, dass die Hashwerte im Merkle-Verfahren die Länge 3 haben. Die Hashfunktion H arbeitet so: Sie nimmt die letzten drei Bits des Dokuments, das gehasht wird und kehrt die Reihenfolge um. Es ist also

$$H(111111111011) = 110, \quad H(0000000001) = 100.$$

Das ist zwar überhaupt nicht kollisionsresistent. Aber Hashfunktionen, die auf Hashwerte der Länge 3 abbilden können nie kollisionsresistent sein. Dieses Beispiel dient nur der Illustration. Der Signierschlüssel sei

$$(x(0,1), x(1,1), x(0,2), x(1,2), x(0,3), x(1,3)) = \begin{pmatrix} 1\,0\,1\,0\,0\,1 \\ 1\,1\,1\,1\,1\,0 \\ 1\,0\,0\,0\,1\,1 \end{pmatrix}.$$

Der Verifikationsschlüssel ist dann

$$(y(0,1), y(1,1), y(0,2), y(1,2), y(0,3), y(1,3)) = \begin{pmatrix} 1\,0\,0\,0\,1\,1 \\ 1\,1\,1\,1\,1\,0 \\ 1\,0\,1\,0\,0\,1 \end{pmatrix}.$$

Signiert wird das Dokument $d = (11100101001100)$. Sein Hashwert ist $H(d) = (H_1, H_2, H_3) = (001)$. Die Signatur ist

$$\begin{aligned} s &= (s_1, s_2, s_3) = (x(H_1, 1), x(H_2, 2), x(H_3, 3)) \\ &= (x(0,1), x(0,2), x(1,3)) = \begin{pmatrix} 1\,1\,1 \\ 1\,1\,0 \\ 1\,0\,1 \end{pmatrix}. \end{aligned}$$

Um diese Signatur zu verifizieren, berechnet der Verifizierer seinerseits den Hashwert $H(d) = (H_1, H_2, H_3) = (001)$. Er prüft, ob

$$\begin{aligned}\big(H(s_1), H(s_2), H(s_3)\big) &= \big(y(H_1, 1), y(H_2, 2), y(H_3, 3)\big) \\ &= \big(y(0, 1), y(0, 2), y(1, 3)\big)\end{aligned} \tag{13.13}$$

gilt. Der Verizierer berechnet also

$$(H(s_1), H(s_2), H(s_3)) = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}.$$

Der Verifizierer stellt fest, dass (13.13) gilt und akzeptiert die Signatur.

13.8.4 Sicherheit

Angenommen, der Angreifer Oskar möchte die Signatur eines Dokumentes mit Hashwert $(H_1, \ldots, H_n)$ fälschen. Dann muss er für alle Strings $y(H_i, i)$, $1 \leq i \leq n$, passende $s_i \in \{0,1\}^n$ finden mit der Eigenschaft $H(s_i) = y(H_i, i)$, $1 \leq i \leq n$. Er muss also die Urbilder der $y(H_i, i)$ unter der Hashfunktion H finden. Wenn die Funktion H eine Einwegfunktion ist, wenn es also unmöglich ist, in vertretbarer Zeit Urbilder dieser Funktion zu berechnen, dann ist das Signaturverfahren sicher. Es ist aber wichtig, dass jeder Signierschlüssel nur einmal verwendet wird. Bei jeder Signatur werden nämlich n Strings des geheimen Schlüssels preisgegeben. Sie können in einer neuen Signatur wieder verwendet werden, wenn die entsprechenden Bits im Hashwert des zu signierenden Dokuments den richtigen Wert haben.

13.9 Das Merkle-Verfahren

Dieser Abschnitt beschreibt das Merkle-Signaturverfahren aus [53]. Es verwendet einen Hashbaum, um die Gültigkeit vieler Verifikationsschlüssel auf die Gültigkeit eines einzigen öffentlichen Schlüssels, der Wurzel des Hashbaumes, zurückzuführen.

13.9.1 Initialisierung

Bei der Initialisierung des Merkle-Verfahrens wird eine Hashfunktion

$$H : \{0,1\}^* \to \{0,1\}^n$$

festgelegt und ein Einmal-Signaturverfahren gewählt. Das kann das Lamport-Diffie-Verfahren aus dem vorigen Abschnitt oder irgendein anderes Einmal-Signaturverfahren sein. Aber es kann auch jedes andere Signaturverfahren verwendet werden, zum Beispiel das RSA-Verfahren.

Dann wird festgelegt, wie viele Signaturen mit einem einzigen öffentlichen Schlüssel verifiziert werden sollen. Dazu wird eine natürliche Zahl h gewählt. Die Anzahl der verifizierbaren Signaturen ist $N = 2^h$.

Beispiel 13.9.1. Wir verwenden die Hashfunktion H, die folgendermaßen arbeitet: Der Hashwert sind die drei letzten Bits der Binärdarstellung der Quersumme der Dezimalzahl, die durch den zu hashenden String dargestellt wird. Ist diese Binärdarstellung zu kurz, werden führende Nullen ergänzt. Sei $d = 11000000100001$. Die entsprechende Dezimalzahl ist 12321. Die Quersumme dieser Zahl ist 9. Die Binärentwicklung von 9 ist 1001. Der Hashwert ist also $H(d) = 001$. Das Einmal-Signaturverfahren ist das von Lamport-Diffie. Außerdem legen wir fest, dass mit einem öffentlichen Schlüssel 4 Signaturen verifiziert werden sollen. Die Tiefe des Hashbaumes ist also 2.

13.9.2 Schlüsselerzeugung

Der Signierer wählt N Schlüsselpaare (x_i, y_i), $0 \leq i < N$, des verwendeten Einmal-Signaturverfahrens. Dabei ist jeweils x_i der Signierschlüssel und y_i der Verifikationsschlüssel, $0 \leq i < N$. Als nächstes konstruiert der Signierer den Merkle-Hashbaum. Es handelt sich um einen binären Baum. Die Blätter dieses Baums sind die Hashwerte $H(y_i)$, $0 \leq i < N$, der Verifikationsschlüssel. Jeder Knoten im Baum, der kein Blatt ist, ist der Hashwert $H(k_l||k_r)$ seiner beiden Kinder k_l und k_r. Dabei ist k_l das linke Kind und k_r das rechte Kind und $||$ bezeichnet die Verkettung zweier Bitsrings. Der Merkle-Hashbaum wird in Abbildung 13.1 gezeigt.

Der geheime Schlüssel ist die Folge $(x_0, \ldots, x_{N-1})$ der gewählten Signierschlüssel. Der öffentliche Schlüssel ist die Wurzel R des Merkle-Hashbaumes.

Beispiel 13.9.2. Wir setzen das Beispiel 13.9.1 fort. Wir wählen also vier Paare (x_i, y_i), $0 \leq i < 4$. Dabei ist x_i jeweils ein Lamport-Diffie-Signierschlüssel und y_i ist der zugehörige Verifikationsschlüssel. Jedes x_i und jedes y_i besteht also aus sechs Bitstrings der Länge 3. Die werden hier nicht explizit angegeben. Als nächstes wird der Hashbaum berechnet. Die Knoten dieses Baums werden mit $H_{i,j}$ bezeichnet. Dabei ist j die Tiefe des Knotens $H_{i,j}$ im Baum. Die Blätter des Baums seien

$$H_{0,2} = 110, H_{1,2} = 001, H_{2,2} = 010, H_{3,2} = 101.$$

dann sind die Knoten auf Tiefe 1

$$H_{0,1} = H(H_{0,2}||H_{1,2}) = H(110001) = 101,$$
$$H_{1,1} = H(H_{2,2}||H_{3,2}) = H(010101) = 011.$$

Die Wurzel ist

$$R = H_{0,0} = H(H_{0,1}||H_{1,1}) = H(101011) = 111.$$

Der geheime Schlüssel ist die Folge (x_0, x_1, x_2, x_3) der vier Lamport-Diffie-Signierschlüssel. Der öffentliche Schlüssel ist die Wurzel $R = 111$ des Hashbaumes.

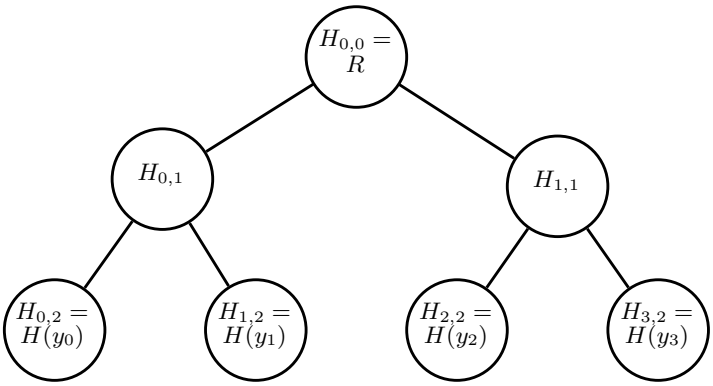


Abb. 13.1. Merkle-Hashbaum der Höhe $h = 2$.

13.9.3 Erzeugung der Signatur

Ein Dokument d soll signiert werden, und dies ist die i-te Signatur, die der Signierer ausstellt. Dann verwendet er den Signierschlüssel x_i und berechnet damit die Einmal-Signatur s des Dokuments. Danach berechnet der Signierer einen Authentisierungspfad, der es dem Verifizierer erlaubt, die Gültigkeit des Verifikationsschlüssels y_i auf die Gültigkeit des öffentlichen Schlüssels R zurückzuführen. Der Authentisierungspfad für den Verifikationsschlüssel y_i ist eine Folge $(a_h, \ldots, a_1)$ von Knoten im Merkle-Hashbaum. Dieser Pfad ist durch folgende Eigenschaft charakterisiert. Bezeichne mit $(b_h, \ldots, b_1, b_0)$ den Pfad im Merkle-Hashbaum vom Blatt $H(y_i)$ zur Wurzel R. Es ist also $b_h = H(y_i)$ und $b_0 = R$. Dann ist a_i, $h \geq i \geq 1$, der Knoten mit demselben Vater wie b_i. Dies wird in Abbildung 13.2 illustriert. Die Signatur von d ist

$$S = (s, y_i, i, a_h, \ldots, a_1).$$

Beispiel 13.9.3. Wir setzen das Beispiel 13.9.2 fort. Signiert werden soll das Dokument d. Dazu wird der vierte Signierschlüssel x_3 verwendet. Die EinmalsSignatur bezeichnen wir mit s. Wir bestimmen den Authentisierungspfad. Der Pfad vom Blatt $H(y_3) = H_{3,2}$ zur Wurzel R im Hashbaum ist $(b_2, b_1, b_0) = (H_{3,2}, H_{1,1}, H_{0,0})$. Der Authentisierungspfad ist $(a_2, a_1) = (H_{2,2}, H_{0,1})$. Die Signatur ist $(s, y_3, 3, a_2, a_1)$.

13.9.4 Verifikation

Der Verifizierer erhält das Dokument d und die Signatur $(s, y_i, i, a_h, \ldots, a_1)$. Zunächst verifiziert er die Signatur s unter Verwendung des Verifikationsschlüssels y_i. Wenn die Verifikation fehlschlägt, wird die Signatur zurückgewiesen. Ist die Verifikation erfolgreich, überprüft der Verifizierer die Gültigkeit des Verifikationsschlüssels y_i. Dazu verwendet er die Zahl i und den

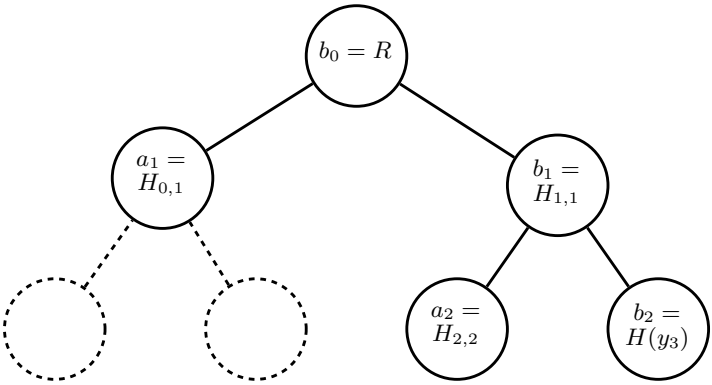


Abb. 13.2. Authentisierungspfad für das 4. Blatt.

Authentisierungspfad $(a_h, \ldots, a_1)$. Er berechnet $b_h = H(y_i)$. Er weiß, dass die Knoten a_h und b_h im Merkle-Hashbaum denselben Vater haben. Er weiß nur noch nicht, ob a_h der linke oder der rechte Nachbar von b_h ist. Das liest er an i ab. Ist i ungerade, dann ist a_h der linke Nachbar von b_h. Andernfalls ist a_h der rechte Nachbar von b_h. Im ersten Fall bestimmt der Verifizierer den Knoten $b_{h-1} = H(a_h||b_h)$ im Pfad $(b_h, \ldots, b_1, b_0)$ vom Blatt $b_h = H(y_i)$ zur Wurzel $b_0 = R$ im Merkle-Hashbaum. Im zweiten Fall berechnet er $b_{h-1} = H(b_h||a_h)$. Die weiteren Blätter b_i, $h - 2 \geq i \geq 1$, werden analog berechnet. Dazu benötigt der Verifizierer die Binärentwicklung $i_0 \cdots i_h$ von i. Angenommen, der Verifizierer hat b_j berechnet, $h \geq j > 0$. Dann kann er b_{j-1} so berechnen. Ist $i_j = 1$, so ist a_j der linke Nachbar von b_j, und es ist $b_{j-1} = H(a_j||b_j)$. Ist aber $i = 0$, dann ist a_j der rechte Nachbar von b_j und es gilt $b_{j-1} = H(b_j||a_j)$. Der Verifizierer akzeptiert die Signatur, wenn $b_0 = R$ ist und weist sie andernfalls zurück. Das Verifikationsverfahren ist in Abbildung 13.3 illustriert.

Beispiel 13.9.4. Wir setzen das Beispiel 13.9.3 fort. Der Verifizierer kennt das Dokument d und die Signatur $(s, y_3, 3, a_2 = 010, a_1 = 101)$. Er verifiziert die Signatur mit dem Verifikationsschlüssel y_3. Ist das erfolgreich, validiert er den Verifikationsschlüssel. Dazu bestimmt er die Binärentwicklung 11 von 3. Dann berechnet er den Knoten $b_2 = H(y_3) = 101$. Da das letzte Bit in der Binärentwicklung von $i = 3$ Eins ist, gilt $b_1 = H(a_2||b_2) = H(010101) = 011$. Das erste Bit in der Binärdarstellung von $i = 3$ ist auch Eins. Darum ist $b_0 = H(a_1||b_1) = H(101011) = 111$ Das ist tatsächlich der öffentliche Schlüssel. Also ist der öffentliche Schlüssel validiert und die Signatur wird akzeptiert.

13.9.5 Sicherheit

Wir zeigen, dass das Merkle-Verfahren sicher ist, solange das Einmal-Signaturverfahren sicher und die verwendete Hashfunktion kollisionsresistent ist.

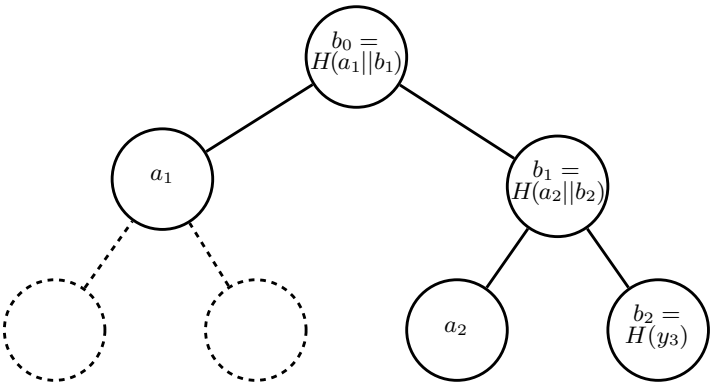


Abb. 13.3. Validierung eines Verifikationsschlüssels.

Der Angreifer Oskar möchte eine gültige Signatur $(s, y, i, a_h, \ldots, a_1)$ eines Dokuments d fälschen. Oskar kennt den i-ten Signaturschlüssel nicht. Wenn das verwendete Einmal-Signaturverfahren sicher ist, kann Oskar keine Einmal-Signatur von d erzeugen, die mit dem Verifikationsschlüssel y_i verifizierbar ist. Oskar muss also den Verifikationsschlüssel ersetzen und dazu einen Authentisierungspfad $(a'_h, \ldots, a'_1)$ fälschen. Wir zeigen, dass die Verifikation des gefälschten Authentisierungspfades eine Kollision der Hashfunktion H liefert. Bei der Verifikation der gefälschten Signatur entsteht eine Folge $b'_h, \ldots, b'_1, b'_0$ von Hashwerten mit $b'_h = H(y)$ und $b'_0 = R$. Bei der Verifikation der echten Signatur entsteht der Pfad $b_h, \ldots, b_1, b_0$ vom Blatt $H(y_i)$ zur Wurzel R. Sind diese beiden Folgen gleich, dann ist insbesondere $H(y) = b'_h = b_h = H(y_i)$. Da aber die beiden Verifikationsschlüssel y und y_i verschieden sind, ist eine Kollision der Hashfunktion H gefunden. Angenommen, die beiden Folgen $b_h, \ldots, b_0$ und $b'_h, \ldots, b'_0$ sind verschieden. Es gilt aber $b_0 = R = b'_0$. Darum gibt es einen minimalen Index $j > 0$ mit $b_j \neq b'_j$ und $b_{j-1} = b'_{j-1}$. Ist das j-te Bit in der Binärentwicklung von i Null, dann gilt $H(a_j||b_j) = b_{j-1} = b'_{j-1} = H(a'_j||b'_j)$. Andernfalls gilt $H(b_j||a_j) = b_{j-1} = b'_{j-1} = H(b'_j||a'_j)$. In beiden Fällen ist eine Kollision der Hashfunktion gefunden. Solange die Hashfunktion kollisionsresistent ist, kann Oskar den Authentisierungspfad also nicht fälschen.

13.9.6 Verbesserungen

Das ursprüngliche Merkle-Signaturverfahren in Kombination mit dem Lamport-Diffie-Einmal-Signaturverfahren hat eine Reihe von Nachteilen. In den letzten Jahren sind aber viele Verbesserungen des Merkle-Verfahrens vorgeschlagen worden, die dieses Verfahren zu einem konkurrenzfähigen Signaturverfahren machen. Wir werden die Probleme und ihre Lösungen jetzt kurz beschreiben und die entsprechende Literatur angeben.

Der Signierschlüssel im Merkle-Verfahren ist zu lang. Man kann ihn durch einen Zufallswert ersetzen und daraus die Schlüssel mit eineem Pseudizufallszahlengenerator erzeugen. Dies wurde von Coronado [19] vorgeschlagen.

Im Lamport-Diffie-Einmal-Signaturverfahren wird jedes Bit einzeln signiert, und die Signatur jedes Bits ist ein Hashwert. Dadurch werden die Merkle-Signaturen sehr lang. Es sind aber in [53, 27] verschiedene Vorschläge gemacht worden, Bits simultan zu signieren. Das macht die Signaturen deutlich kleiner. Aber je mehr Bits simultan signiert werden, desto mehr Hashwerte müssen bei der Signatur und der Verifikation berechnet werden. Die Anzahl der Hashwertberechnungen steigt dabei exponentiell mit der Anzahl der simultan signierten Bits. Daher ist die Möglichkeit, die Signatur auf diese Weise zu verkürzen ohne zu ineffizient zu werden, begrenzt.

Ein weiteres Effizienzproblem besteht darin, beim Signieren den Authentisierungspfad zu berechnen. Es ist natürlich möglich, dass der Signierer den ganzen Hashbaum speichert. Wenn die Höhe dieses Baums aber größer wird, verbraucht der Hashbaum zu viel Platz. Dann kann der Algorithmus von Szydlo [80] verwendet werden. Dieser Algorithmus setzt die Kenntnis des Hashbaumes nicht voraus und benötigt nur Speicherplatz für $3h$ Knoten, wobei h die Höhe des Hashbaumes ist.

Auch wenn der Baum bei Anwendung des Sydlo-Algorithmus nicht gespeichert werden muss, um Authentisierungspfade zu berechnen, so muss er doch für die Erzeugung des öffentlichen Schlüssels vollständig berechnet werden. Ist die Höhe des Baums größer als 20, so wird das sehr langsam. Darum hat Coronado [19] vorgeschlagen, statt eines Baums mehrere Bäume zu verwenden. Die können nach und nach berechnet werden. Diese Idee wird von Dahmen und Vuillaume in [20] weiterentwickelt.

13.10 Übungen

Übung 13.10.1. Berechnen Sie die RSA-Signatur (ohne Hashfunktion) von $m = 11111$ mit RSA-Modul $n = 28829$ und dem kleinstmöglichen öffentlichen Exponenten e.

Übung 13.10.2. Stellt die Low-Exponent-Attacke oder die Common-Modulus-Attacke ein Sicherheitsproblem für RSA-Signaturen dar?

Übung 13.10.3. Wie kann man aus dem Rabin-Verschlüsselungsverfahren ein Signaturverfahren machen? Beschreiben Sie die Funktionsweise eines Rabin-Signaturverfahrens und diskutieren Sie seine Sicherheit und Effizienz.

Übung 13.10.4. Berechnen Sie die Rabin-Signatur (ohne Hashfunktion) von $m = 11111$ mit dem Rabin-Modul $n = 28829$.

Übung 13.10.5. Im ElGamal-Signaturverfahren werde die Primzahl p, $p \equiv 1 \pmod 4$ und die Primitivwurzel $g \bmod p$ benutzt. Angenommen, $p-1 = gq$ mit $q \in \mathbb{Z}$ und g hat nur kleine Primfaktoren. Sei A der öffentliche Schlüssel von Alice.

1. Zeigen Sie, dass sich eine Lösung z der Kongruenz $A^q = g^{qz} \bmod p$ effizient finden läßt.
2. Sei x ein Dokument und sei h der Hashwert von x. Zeigen Sie, dass $(q, (p-3)(h-qz)/2)$ eine gültige Signatur von x ist.

Übung 13.10.6. Sei $p = 130$. Berechnen Sie einen gültigen privaten Schlüssel a und den entsprechenden öffentlichen Schlüssel (p, g, A) für das ElGamal-Signaturverfahren.

Übung 13.10.7. Sei $p = 2237$ und $g = 2$. Der geheime Schlüssel von Alice ist $a = 1234$. Der Hashwert einer Nachricht m sei $h(m) = 111$. Berechnen Sie die ElGamal-Signatur mit $k = 2323$ und verifizieren Sie sie.

Übung 13.10.8. Angenommen, im ElGamal-Signaturverfahren ist die Bedingung $1 \leq r \leq p-1$ nicht gefordert. Verwenden Sie die existentielle Fälschung aus Abschnitt 13.5.6, um eine ElGamal-Signatur eines Dokumentes x' mit Hashwert $h(x') = 99$ aus der Signatur in Beispiel 13.10.7 zu konstruieren.

Übung 13.10.9. Es gelten dieselben Bezeichnungen wie in Übung 13.10.7. Alice verwendet das DSA-Verfahren, wobei q der größte Primteiler von $p-1$ ist. Sie verwendet aber $k = 25$. Wie lautet die entsprechende DSA-Signatur? Verifizieren Sie die Signatur.

Übung 13.10.10. Erläutern Sie die existentielle Fälschung aus Abschnitt 13.5.6 für das DSA-Verfahren.

Übung 13.10.11. Wie lautet die Verifikationskongruenz, wenn im ElGamal-Signaturverfahren s zu $s = (ar + kh(x)) \bmod p-1$ berechnet wird?

Übung 13.10.12. Modifizieren Sie die ElGamal-Signatur so, daß die Verifikation nur zwei Exponentiationen benötigt.

14. Andere Gruppen

Wie in den Abschnitten 9.5.4 und 13.5.9 beschrieben wurde, kann das ElGamal-Verschlüsselungs- und Signaturverfahren nicht nur in der primen Restklassengruppe modulo einer Primzahl,genauso sondern auch in anderen Gruppen realisiert werden, in denen das Problem, diskrete Logarithmen zu berechnen, sehr schwer ist. Es sind einige Gruppen vorgeschlagen worden, die wir hier kurz beschreiben. Für ausführlichere Beschreibungen verweisen wir aber auf die Literatur.

14.1 Endliche Körper

Bis jetzt wurden die ElGamal-Verfahren in der Einheitengruppe eines endlichen Körpers von Primzahlordnung beschrieben. In diesem Abschnitt beschreiben wir, wie man die ElGamal-Verfahren auch in anderen endlichen Körpern realisieren kann.

Sei p eine Primzahl und n eine natürliche Zahl. Wir haben aber in Theorem 3.21.1 gezeigt, daß die Einheitengruppe des endlichen Körpers $\mathrm{GF}(p^n)$ zyklisch ist. Die Ordnung dieser Einheitengruppe ist $p^n - 1$. Wenn diese Ordnung nur kleine Primfaktoren hat, kann man das Pohlig-Hellmann-Verfahren anwenden und effizient diskrete Logarithmen berechnen (siehe Abschnitt 11.5). Wenn dies nicht der Fall ist, kann man Index-Calculus-Algorithmen anwenden. Für festes n und wachsende Charakteristik p verwendet man das Zahlkörpersieb. Für feste Charakteristik und wachsenden Grad n benutzt man das Funktionenkörpersieb [69]. Beide haben die Laufzeit $L_q[1/3, c+o(1)]$ (siehe Abschnitt 10.4), wobei c eine Konstante und $q = p^n$ ist. Werden p und n simultan vergrößert, so ist die Laufzeit immer noch $L_q[1/2, c + o(1)]$.

14.2 Elliptische Kurven

Elliptische Kurven kann man über beliebigen Körpern definieren. Für die Kryptographie interessant sind elliptische Kurven über endlichen Körpern, speziell über Primkörpern. Der Einfachheit halber beschreiben wir hier nur elliptische Kurven über Primkörpern. Mehr Informationen über elliptische

Kurven und ihre kryptographische Anwendung findet man in [43], [51] und [12].

14.2.1 Definition

Sei p eine Primzahl $p > 3$. Seien $a, b \in \mathrm{GF}(p)$. Betrachte die Gleichung

$$y^2 z = x^3 + axz^2 + bz^3. \tag{14.1}$$

Die *Diskriminante* dieser Gleichung ist

$$\Delta = -16(4a^3 + 27b^2). \tag{14.2}$$

Wir nehmen an, daß die Diskriminante Δ nicht Null ist. Ist $(x, y, z) \in \mathrm{GF}(p)^3$ eine Lösung dieser Gleichung, so ist für alle $c \in \mathrm{GF}(p)$ auch $c(x, y, z)$ eine solche Lösung. Zwei Lösungen (x, y, z) und (x', y', z') heißen *äquivalent*, wenn es ein von Null verschiedenes $c \in \mathrm{GF}(p)$ gibt mit $(x, y, z) = c(x', y', z')$. Dies definiert eine Äquivalenzrelation auf der Menge aller Lösungen von (14.1). Die Äquivalenzklasse von (x, y, z) wird mit $(x : y : z)$ bezeichnet. Die Elliptische Kurve $E(p; a, b)$ ist definiert als die Menge aller Äquivalenzklassen von Lösungen dieser Gleichung, die nicht $(0 : 0 : 0)$ sind. Jedes Element dieser Menge heißt *Punkt auf der Kurve*.

Wir vereinfachen die Beschreibung der elliptischen Kurve. Ist (x', y', z') eine Lösung von (14.1) und ist $z' \neq 0$, dann gibt es in $(x' : y' : z')$ genau einen Vertreter $(x, y, 1)$. Dabei ist (x, y) eine Lösung der Gleichung

$$y^2 = x^3 + ax + b. \tag{14.3}$$

Ist umgekehrt $(x, y) \in \mathrm{GF}(p)^2$ eine Lösung von (14.3), dann ist $(x, y, 1)$ eine Lösung von (14.1). Außerdem gibt es genau eine Äquivalenzklasse von Lösungen (x, y, z) mit $z = 0$. Ist nämlich $z = 0$, dann ist auch $x = 0$. Die Äquivalenzklasse ist $(0 : 1 : 0)$. Damit ist die elliptische Kurve

$$E(p; a, b) = \{(x : y : 1) : y^2 = x^3 + ax + b\} \cup \{(0 : 1 : 0)\}.$$

Oft schreibt man auch (x, y) für $(x : y : 1)$ und $\mathcal{O}$ für $(0 : 1 : 0)$. Damit ist dann

$$E(p; a, b) = \{(x, y) : y^2 = x^3 + ax + b\} \cup \{\mathcal{O}\}.$$

Beispiel 14.2.1. Wir arbeiten im Körper GF(11). Die Elemente des Körpers stellen wir durch ihre kleinsten nicht negativen Vertreter dar. Über diesem Körper betrachten wir die Gleichung

$$y^2 = x^3 + x + 6. \tag{14.4}$$

Es ist $a = 1$ und $b = 6$. Ihre Diskriminante ist $\Delta = -16 * (4 + 27 * 6^2) = 4$. Also definiert Gleichung (14.4) eine elliptische Kurve über GF(11). Sie ist

$$\begin{aligned} E(11; 1, 6) = \{&\mathcal{O}, (2, 4), (2, 7), (3, 5), (3, 6), (5, 2), (5, 9), (7, 2), \\ &(7, 9), (8, 3), (8, 8), (10, 2), (10, 9)\}. \end{aligned}$$

14.2.2 Gruppenstruktur

Sei p eine Primzahl, $p > 3$, $a, b \in \mathrm{GF}(p)$ und sei $E(p; a, b)$ eine elliptische Kurve. Wir definieren die Addition von Punkten auf dieser Kurve.

Für jeden Punkt P auf der Kurve setzt man

$$P + \mathcal{O} = \mathcal{O} + P = P.$$

Der Punkt $\mathcal{O}$ ist also das neutrale Element der Addition.

Sei $P = (x, y)$ ein von $\mathcal{O}$ verschiedener Punkt der Kurve. Dann ist $-P = (x, -y)$ und man setzt $P + (-P) = \mathcal{O}$.

Seien P_1, P_2 Punkte der Kurve, die beide von $\mathcal{O}$ verschieden sind und für die $P_2 \neq -P_1$ gilt. Sei $P_i = (x_i, y_i)$, $i = 1, 2$. Dann berechnet man

$$P_1 + P_2 = (x_3, y_3)$$

folgendermaßen: Setze

$$\lambda = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1}, \text{ falls } P \neq Q, \\ \frac{3x_1^2 + a}{2y_1}, \text{ falls } P = Q \end{cases}$$

und

$$x_3 = \lambda^2 - x_1 - x_2, y_3 = \lambda(x_1 - x_3) - y_1.$$

Man kann zeigen, daß $E(p; a, b)$ mit dieser Addition eine abelsche Gruppe ist.

Beispiel 14.2.2. Wir verwenden die Kurve aus Beispiel 14.2.1 und berechnen die Punktsumme $(2, 4) + (2, 7)$. Da $(2, 7) = -(2, 4)$ ist, folgt $(2, 4) + (2, 7) = \mathcal{O}$. Als nächstes berechnen wir $(2, 4) + (3, 5)$. Wir erhalten $\lambda = 1$ und $x_3 = -4 = 7$, $y_3 = 2$. Also ist $(2, 4) + (3, 5) = (7, 2)$. Als letztes gilt $(2, 4) + (2, 4) = (5, 9)$, wie der Leser leicht verifizieren kann.

14.2.3 Kryptographisch sichere Kurven

Sei p eine Primzahl, $p > 3$, $a, b \in \mathrm{GF}(p)$ und sei $E(p; a, b)$ eine elliptische Kurve. In der Gruppe $E(p; a, b)$ kann man das Diffie-Hellman-Schlüsselaustauschverfahren (siehe Abschnitt 9.5) und die ElGamal-Verfahren zur Verschlüsselung und Signatur (siehe Abschnitte 9.5.4 und 13.5.9) implementieren.

Damit diese Verfahren sicher sind, muß es schwierig sein, in $E(p; a, b)$ diskrete Logarithmen zu berechnen. Der schnellste bekannte Algorithmus zur DL-Berechnung auf beliebigen elliptischen Kurven ist der Pohlig-Hellman-Algorithmus (siehe Abschnitt 11.5). Für spezielle Kurven, sogenannte *supersinguläre* und *anomale* Kurven, sind schnellere Algorithmen bekannt.

Man geht heute davon aus, daß eine Kurve $E(p; a, b)$, die die gleiche Sicherheit wie 1024-Bit RSA-Systeme bietet, weder supersingulär noch anomal ist, etwa 2^{163} Punkte hat und daß die Punktanzahl der Kurve zur Verhinderung von Pohlig-Hellman-Attacken einen Primfaktor $q \geq 2^{160}$ hat. Wir beschreiben kurz, wie man solche Kurven findet.

Die Anzahl der Punkte auf der Kurve $E(p; a, b)$ ergibt sich aus folgendem Satz.

Theorem 14.2.3 (Hasse). *Für die Ordnung der Gruppe* $E(p; a, b)$ *gilt* $|E(p; a, b)| = p + 1 - t$ *mit* $|t| \leq 2\sqrt{p}$.

Das Theorem von Hasse garantiert, daß die elliptische Kurve $E(p; a, b)$ ungefähr p Punkte hat. Um eine Kurve mit etwa 2^{163} Punkten zu erhalten, braucht man $p \approx 2^{163}$. Liegt p fest, wählt man die Koeffizienten a und b zufällig und bestimmt die Ordnung der Punktgruppe. Dies ist in Polynomzeit möglich, aber der Algorithmus zur Berechnung der Ordnung braucht pro Kurve einige Minuten. Ist die Kurve supersingulär, anomal oder hat sie keinen Primfaktor $q \geq 2^{160}$, so verwirft man sie und wählt neue Koeffizienten a und b. Andernfalls wird die Kurve als kryptographisch sicher akzeptiert.

Zu dem beschriebenen Auswahlverfahren für kryptographisch sichere Kurven gibt es eine Alternative: die Erzeugung von Kurven mit komplexer Multiplikation (siehe [51], [65]). In dieser Methode wird zuerst die Ordnung der Punktgruppe, aber nicht die Kurve selbst erzeugt. Das geht wesentlich schneller als die Bestimmung der Punktordnung einer zufälligen Kurve. An der Gruppenordnung läßt sich ablesen, ob die Kurve kryptographisch geeignet ist. Erst wenn eine kryptographisch sichere Ordnung gefunden ist, wird die zugehörige Kurve erzeugt. Die Erzeugung der Kurve ist aufwendig. Mit dieser Methode kann man nur eine kleine Teilmenge aller möglichen Kurven erzeugen.

14.2.4 Vorteile von EC-Kryptographie

Die Verwendung elliptischer Kurven für kryptographische Anwendungen kann mehrere Gründe haben.

Public-Key-Kryptographie mit elliptischen Kurven ist die wichtigste bis jetzt bekannte Alternative zu RSA-basierten Verfahren. Solche Alternativen sind dringend nötig, da niemand die Sicherheit von RSA garantieren kann.

Ein zweiter Grund für die Verwendung von EC-Kryptosystemen besteht darin, daß sie Effizienzvorteile gegenüber RSA-Verfahren bieten. Während nämlich RSA-Verfahren modulare Arithmetik mit 1024-Bit-Zahlen verwenden, begnügen sich EC-Verfahren mit 163-Bit-Zahlen. Zwar ist die Arithmetik auf elliptischen Kurven aufwendiger als die in primen Restklassengruppen. Das wird aber durch die geringere Länge der verwendeten Zahlen kompensiert. Dadurch ist es z.B. möglich, EC-Kryptographie auf Smart-Cards ohne Koprozessor zu implementieren. Solche Smart-Cards sind wesentlich billiger als Chipkarten mit Koprozessor.

14.3 Quadratische Formen

Es ist auch möglich, Klassengruppen binärer quadratischer Formen oder, allgemeiner, Klassengruppen algebraischer Zahlkörper zur Implementierung kryptographischer Verfahren zu benutzen (siehe [17] und [18]).

Klassengruppen weisen einige Unterschiede zu den anderen Gruppen auf, die bis jetzt beschrieben wurden. Die Ordnung der Einheitengruppe des endlichen Körpers $\mathrm{GF}(p^n)$ ist $p^n - 1$. Die Ordnung der Punktgruppe einer elliptischen Kurve über einem endlichen Körper kann in Polynomzeit bestimmt werden. Dagegen sind keine effizienten Algorithmen zur Bestimmung der Ordnung der Klassengruppe eines algebraischen Zahlkörpers bekannt. Die bekannten Algorithmen benötigen genauso viel Zeit zur Bestimmung der Gruppenordnung wie zur Lösung des DL-Problems. Sie haben subexponentielle Laufzeit für festen Grad des Zahlkörpers. Der zweite Unterschied: Kryptographische Anwendungen in Einheitengruppen endlicher Körper und Punktgruppen elliptischer Kurven beruhen darauf, daß in diesen Gruppen das DL-Problem schwer ist. In Klassengruppen gibt es ein weiteres Problem: zu entscheiden, ob zwei Gruppenelemente gleich sind. In den Einheitengruppen endlicher Körper und in Punktgruppen elliptischer Kurven ist der Gleichheitstest trivial. In Klassengruppen gibt es aber i.a. keine eindeutige, sondern nur eine höchst mehrdeutige Darstellung der Gruppenelemente. Darum ist der Gleichheitstest schwierig. Je kleiner die Klassengruppe ist, umso schwieriger ist es, Gleichheit von Gruppenelementen zu entscheiden. Es kommt sogar häufig vor, daß die Klassengruppe nur ein Element hat. Dann liegt die Basis für die Sicherheit kryptographischer Verfahren nur in der Schwierigkeit des Gleichheitstests.

Es wird zur Zeit in meiner Arbeitsgruppe erforscht, wie man Klassengruppen am besten in der Kryptographie verwenden kann.

14.4 Übungen

Übung 14.4.1. Konstruieren Sie den endlichen Körper GF(9) samt seiner Additions- und Multiplikationstabelle.

Übung 14.4.2.
1. Konstruieren Sie GF(125) und bestimmen Sie ein erzeugendes Element der multiplikativen Gruppe $\mathrm{GF}(125)^*$.
2. Bestimmen Sie einen gültigen öffentlichen und privaten Schlüssel für das ElGamal-Sigaturverfahren in $\mathrm{GF}(125)^*$.

Übung 14.4.3. Wieviele Punkte hat die elliptische Kurve $y^2 = x^3 + x + 1$ über GF(7)? Ist die Punktgruppe zyklisch? Wenn ja, bestimmen Sie einen Erzeuger.

Übung 14.4.4. Sei p eine Primzahl, $p \equiv 3 \bmod 4$ und sei E eine elliptische Kurve über $\mathrm{GF}(p)$. Finden Sie einen Polynomzeitalgorithmus, der für $x \in \mathrm{GF}(p)$ einen Punkt (x, y) auf E konstruiert, falls ein solcher Punkt existiert. Hinweis: Benutzen Sie Übung 3.23.21. Verwenden Sie den Algorithmus, um einen Punkt $(2, y)$ auf $E(111119; 1, 1)$ zu finden.

15. Identifikation

In den vorigen Kapiteln wurden zwei wichtige kryptographische Basismechanismen erklärt: Verschlüsselung und digitale Signatur. In diesem Kapitel geht es um eine dritte grundlegende Technik, die Identifikation.

15.1 Anwendungen

Wir geben zuerst zwei Beispiele für Situationen, in denen Identifikation nötig ist:

Beispiel 15.1.1. Alice will über das Internet von ihrer Bank erfahren, wieviel Geld noch auf ihrem Konto ist. Dann muß sie sich der Bank gegenüber identifizieren. Sie muß also beweisen, daß tatsächlich Alice diese Anfrage stellt und nicht ein Betrüger.

Beispiel 15.1.2. Bob arbeitet in einer Universität und verwendet dort einen Unix-Computer. Bob steht in dem Benutzer-Verzeichnis des Rechners. Wenn Bob morgens zur Arbeit kommt, muß er sich bei dem Computer anmelden und dabei beweisen, daß er ihn tatsächlich benutzen darf. Dazu demonstriert Bob dem Rechner, daß er tatsächlich Bob ist. Der schaut in der Benutzerliste nach, findet Bob dort und gewährt ihm Zugang. Voraussetzung für eine erfolgreiche Anmeldung ist also, daß Bob sich dem Netz gegenüber identifiziert. Meist geschieht das mit Paßwörtern. Wir werden diese Identifikationsmethode und ihre Probleme in Abschnitt 15.2 diskutieren.

Identifikation ist in vielen Anwendungen nötig. Typischerweise geht es dabei um die Überprüfung einer Zugangsberechtigung, die an eine bestimmte Identität gebunden ist. Verfahren, die Identifikation ermöglichen, nennt man *Identifikationsprotokolle*. In einem Identifikationsprotokoll demonstriert der *Beweiser* dem *Verifizierer*, daß der Verifizierer gerade mit dem Beweiser kommuniziert oder kurz vorher mit ihm kommuniziert hat. Es ist also wichtig, daß die Identifikation in Realzeit erfolgt.

In diesem Kapitel werden verschiedene Identifikationsprotokolle besprochen.

15.2 Paßwörter

Der Zugang zu Unix- und NT-Systemen funktioniert normalerweise mit Paßwörtern. Das Paßwort w wird vom Benutzer ausgewählt und ist nur ihm bekannt. Im Rechner wird der Funktionswert $f(w)$ von w unter einer Einwegfunktion f gespeichert. Wenn der Benutzer Zugang wünscht, gibt er seinen Namen und sein Paßwort ein. Der Rechner bestimmt den Funktionswert $f(w)$ und vergleicht ihn mit dem Wert, der bei dem Benutzer in der Liste gespeichert ist. Wenn die beiden Werte übereinstimmen, dann erhält der Benutzer Zugang, andernfalls nicht.

Paßwortschutz wird aber nicht nur beim Zugang zu NT- oder Unix-Rechnern verwendet. Auch WWW-Seiten oder geheime Public-Key-Schlüssel werden häufig durch Paßwörter geschützt.

Weil nicht das Paßwort w selbst, sondern nur der Funktionswert $f(w)$ auf dem Rechner abgelegt ist, und weil f eine Einwegfunktion ist, nützt es nichts, die Paßwortdatei auszuspionieren. Wenn man nämlich $f(w)$ gefunden hat, kann man daraus w nicht ermitteln. Trotzdem hat das beschriebene Paßwortverfahren einige Sicherheitslücken.

Eine erste besteht darin, daß sich der Benutzer sein Paßwort merken muß. Also muß er das Paßwort so wählen, daß er es nicht vergißt. Häufig werden Namen von Verwandten benutzt. Das eröffnet folgende Angriffsmöglichkeit: Der Angreifer bildet für alle Wörter w aus einem Wörterbuch den Funktionswert $f(w)$. Er vergleicht diese Funktionswerte mit den abgespeicherten Funktionswerten. Sobald ein berechneter mit einem abgespeicherten Wert übereinstimmt, hat der Angreifer ein geheimes Paßwort gefunden. Um diesen Angriff zu verhindern, muß man Paßwörter wählen, die nicht in Wörterbüchern vorkommen. Es empfiehlt sich, Sonderzeichen wie z.B. \$ oder # in den Paßwörtern zu verwenden. Damit wird es aber schwieriger, sich die Paßwörter zu merken. Man kann die Paßwörter auch auf Chipkarten speichern. Anstatt sein Paßwort einzugeben, steckt der Benutzer seine Chipkarte in ein Lesegerät, das mit dem Rechner verbunden ist. Der Rechner liest das Paßwort von der Chipkarte und startet mit diesem Paßwort den Identifikationsvorgang. Jetzt kann das Paßwort beliebig kompliziert sein. Niemand braucht es sich zu merken. Im Gegenteil. Es ist besser, wenn der Benutzer sein Paßwort gar nicht kennt. Dann kann er es auch nicht verraten.

Ein Angreifer kann auch versuchen, das geheime Paßwort durch Abhören einer Leitung zu finden. Gelingt ihm das, so kennt er das Paßwort und kann sich danach selbst erfolgreich anmelden. Dieser Angriff ist besonders dann möglich, wenn die Entfernung zwischen dem Anmelder und dem Rechner, auf dem er sich anmelden will, groß ist. Das ist z.B. bei Anmeldungen auf Internet-Seiten der Fall.

Schließlich kann der Angreifer versuchen, das Paßwortverzeichnis zu ändern und sich selbst in das Paßwortverzeichnis einzutragen. Das Paßwortver-

zeichnis muß also sicher vor unberechtigten Schreibzugriffen geschützt werden.

15.3 Einmal-Paßwörter

Im vorigen Abschnitt wurde gezeigt, daß die Verwendung von feststehenden Paßwörtern problematisch ist, weil sie abgehört werden können. Dies ist bei Einmalpaßwörtern nutzlos. Ein Einmalpaßwort wird nämlich nur für einen Anmeldevorgang benutzt und danach nicht mehr.

Die einfachste Methode, das zu realisieren, besteht darin, daß der Beweiser eine Liste $w_1, w_2, \ldots, w_n$ geheimer Paßwörter und der Verifizierer die Liste der Funktionswerte $f(w_1), \ldots, f(w_n)$ hat. Zur Identifikation werden dann der Reihe nach die Paßwörter $w_1, w_2, \ldots, w_n$ benutzt. Die Schwierigkeit dieses Verfahrens besteht darin, daß der Beweiser alle Paßwörter schon von vornherein kennen muß. Daher können sie auch vorher ausspioniert werden.

Eine Alternative besteht darin, daß Beweiser und Verifizieren die Kenntnis einer geheimen Funktion f und eines Startwertes w teilen. Die Paßwörter sind $w_i = f^i(w)$, $i \geq 0$. Bei diesem Verfahren müssen Beweiser und Verifizierer keine Paßwörter a priori kennen. Das Paßwort w_i kann als $w_i = f(w_{i-1})$ auf beiden Seiten zum Zeitpunkt der Identifikation berechnet werden. Es muß aber dafür gesorgt sein, daß die geheime Einwegfunktion wirklich geheim bleibt.

15.4 Challenge-Response-Identifikation

Bei den bis jetzt beschriebenen Identifikationsverfahren kann ein Angreifer lange vor seinem Betrugsversuch in den Besitz eines geheimen Paßworts kommen, das er dann zu irgendeinem späteren Zeitpunkt einsetzen kann. Das funktioniert sogar bei Einmalpaßwörtern.

Bei Challenge-Response-Verfahren ist das anders. Will sich Alice Bob gegenüber identifizieren, bekommt sie von Bob eine Aufgabe (Challenge). Die Aufgabe kann sie nur lösen, weil sie ein Geheimnis kennt. Die Lösung schickt sie als Antwort (Response) an Bob. Bob verifiziert die Lösung und erkennt die Identität von Alice an, wenn die Lösung korrekt ist. Andernfalls erkennt er Alices Identität nicht an.

15.4.1 Verwendung von symmetrischer Kryptographie

Ein einfaches Challenge-Response-Verfahren benutzt ein symmetrisches Verschlüsselungsverfahren. Der Einfachheit halber nehmen wir an, daß bei diesem Verfahren die Schlüssel zum Ver- und Entschlüsseln gleich sind. Alice

und Bob kennen beide einen geheimen Schlüssel k. Will Alice sich Bob gegenüber identifizieren, teilt sie das Bob mit. Bob erzeugt eine Zufallszahl r und schickt sie an Alice. Alice verschlüsselt die Zufallszahl, bildet also $c = E_k(r)$ und schickt den Schlüsseltext c an Bob. Der entschlüsselt c, berechnet also $r' = D_k(c)$. Dann vergleicht er, ob $r = r'$ gilt. Wenn ja, ist Alices Identität anerkannt. Andernfalls nicht.

Der Nachteil dieses Verfahrens ist, daß auch Bob den geheimen Schlüssel kennt, der es Alice ermöglicht, sich zu identifizieren. Wird dieses Identifikationsverfahren also als Zugangsschutz zu einem Rechnernetz verwendet, liegen die geheimen Schlüssel im Netz. Sie müssen vor unberechtigtem Zugriff geschützt werden.

15.4.2 Verwendung von Public-Key-Kryptographie

Challenge-Response-Protokolle kann man auch unter Verwendung von Signaturverfahren realisieren. Will sich Alice bei Bob identifizieren, erhält sie von Bob eine Zufallszahl, die Challenge, und signiert diese. Sie schickt die Signatur an Bob (Response). Bob verifiziert die Signatur.

Bei diesem Verfahren verwendet Bob nur den öffentlichen Schlüssel von Alice. Er kennt Alices privaten Schlüssel nicht. Das ist sicherer als die Verwendung von symmetrischen Verfahren. Trotzdem muß der öffentliche Schlüssel von Alice geschützt werden, zwar nicht vor Lesezugriffen, aber vor Veränderung. Gelingt es nämlich einem Angreifer, Alices öffentlichen Schlüssel gegen seinen eigenen auszutauschen, kann er sich auch erfolgreich als Alice identifizieren.

15.4.3 Zero-Knowledge-Beweise

Challenge-Response-Protokolle beruhen darauf, daß der Beweiser dem Verifizierer demonstrieren kann, daß er Kenntnis von einem Geheimnis hat. Bei der Verwendung von symmetrischen Verschlüsselungsverfahren kennt der Verifizierer das Geheimnis auch. Benutzt man dagegen Signaturverfahren, kennt der Verifizierer das Geheimnis nicht. Das ist natürlich sicherer.

Man kann auch spezielle Identifikationsverfahren benutzen. Als Beispiel beschreiben wir Zero-Knowledge-Beweise (ZK-Beweise). Der Beweiser kennt ein Geheimnis, der Verifizierer nicht. In dem Protokoll überzeugt der Beweiser den Verifizierer davon, daß er das Geheimnis kennt. Ein Betrüger kann das nicht. Diese Eigenschaften haben alle anderen Public-Key-Identifikationsverfahren auch. Zero-Knowledge-Beweise haben aber eine wichtige zusätzliche Eigenschaft. Man kann mathematisch beweisen, daß der Verifizierer nichts über das Geheimnis erfährt, obwohl er am Ende des Identifikationsprotokolls von der Identität des Beweisers überzeugt ist. Dieser mathematische Beweis verwendet die Sprache der Komplexitätstheorie. Wir werden dies hier nicht

entwickeln. Wir werden statt dessen Zero-Knowledge-Beweise an einem Beispiel, dem *Fiat-Shamir-Verfahren*, erläutern.

Bob, der Beweiser, wählt wie im RSA-Verfahren zwei Primzahlen p und q und berechnet $n = pq$. Dann wählt er eine Zahl s zufällig und gleichverteilt aus der Menge der zu n primen Zahlen in $\{1, \ldots, n-1\}$ und berechnet $v = s^2 \bmod n$. Bobs öffentlicher Schlüssel ist (v, n). Sein geheimer Schlüssel ist s, eine Quadratwurzel von v mod n.

In einem Zero-Knowledge-Beweis wird Bob der Verifiziererin Alice beweisen, daß er eine Quadratwurzel s von v mod n kennt. Das Identifikationsprotokoll funktioniert so:

1. **Committment** Bob wählt zufällig und gleichverteilt $r \in \{1, 2, \ldots, n-1\}$ und berechnet $x = r^2 \bmod n$. Das Ergebnis x sendet Bob an die Verifiziererin Alice.
2. **Challenge** Alice wählt zufällig und gleichverteilt eine Zahl $e \in \{0, 1\}$ und sendet sie an Bob. x.
3. **Response für** $e = 0$ Wenn Bob den Wert $e = 0$ erhält, dann schickt er die Zufallszahl r an Alice und Alice verifiziert, daß $r^2 \equiv x \bmod n$ istx.
4. **Response für** $e = 1$ Wenn Bob den Wert $e = 1$ erhält, dann schickt er die Zahl $y = rs \bmod n$ an Alice und Alice verifiziert, daß $y^2 \equiv xv \bmod n$ ist.

Beispiel 15.4.1. Es sei $n = 391 = 17 * 23$. Der geheime Schlüssel von Bob ist $s = 123$. Der öffentliche Schlüssel von Bob ist also $(271, 391)$. In einem Identifikationsprotokoll will Bob Alice beweisen, daß er eine Quadratwurzel von v mod n kennt. Er wählt $r = 271$ und schickt $x = r^2 \bmod n = 324$ an Alice. Alice schickt die Challenge $e = 1$ an Bob. Er schickt die Response $y = rs \bmod n = 98$ zurück. Alice verifiziert $220 = y^2 \equiv vx \bmod n$.

Wir analysieren dieses Protokoll.

Wenn Bob das Geheimnis, die Quadratwurzel s aus v mod n, kennt, dann kann er beide möglichen Fragen von Alice, nämlich die nach r und die nach y ,richtig beantworten. Man sagt, daß das Protokoll *vollständig* ist.

Wir haben bereits im Abschnitt 9.4.5 gesehen, daß jeder, der Quadratwurzeln aus Zahlen mod n berechnen kann, auch den Modul n faktorisieren kann. Daher ist das Geheimnis s sicher, solange Faktorisieren schwer ist, was heutzutage noch der Fall ist. Der Betrüger Oskar kann also das Geheimnis s nicht bestimmen.

Aber was passiert, wenn Oskar trotzdem das Protokoll mit Alice durchführt? Dann kann er jedenfalls nicht beide Fragen, nach r und nach y, richtig beantworten. Würde Oskar Zahlen r und y kennen, für die $r^2 \equiv x \bmod n$ und $y^2 \equiv xv \bmod n$ gilt, dann könnte er die Quadratwurzel $s = yr^{-1} \bmod n$ aus v mod n berechnen. Aber eine solche Quadratwurzel kennt Oskar nicht. Oskar kann aber x so bestimmen, daß er entweder ein korrektes r oder ein richtiges y angeben kann. Will er die Frage nach r richtig beantworten, dann bestimmt er x als Quadrat einer ihm bekannten Zahl r mod n. Will er aber ein korrektes y

zurücksenden, dann wählt er y vor und bestimmt x zu y^2v^{-1} mod n. In jedem Fall wird Oskar eine der beiden Fragen falsch beantworten, und weil er x übermitteln muß, bevor er weiß, ob er r oder y angeben muß, wird Alice seine falsche Identität mit der Wahrscheinlichkeit $1/2$ auffallen. Alice stellt nämlich jede der beiden Fragen mit der Wahrscheinlichkeit $1/2$. Damit Alice sicher ist, daß der Beweiser wirklich Bob ist, wird das Identifikationsprotokoll mit neuen Werten für r und e wiederholt. Die Wahrscheinlichkeit dafür, daß Oskar in k aufeinanderfolgenden Iterationen des Protokolls erfolgreich betrügen kann, ist $1/2^k$. Nach k erfolgreichen Versuchen ist also Alice mit Wahrscheinlichkeit $1-1/2^k$ davon überzeugt, daß Bob tatsächlich Bob ist. Daher nennt man das Protokoll *korrekt*.

Wir zeigen, daß das Protokoll die *Zero-Knowledge-Eigenschaft* hat. Zuerst erläutern wir, was das bedeutet.

Angenommen, Alice wollte betrügen. Welches Betrugsziel könnte sie haben? Alice kann zum Beispiel versuchen, Bobs Geheimnis, also eine Quadratwurzel aus v modulo n, zu ermitteln. Wir gehen davon aus, daß dafür die Kenntnis von n und v nicht genügt, wenn n hinreichen groß und richtig gewählt ist. Aber vielleicht kann Alice die Informationen aus dem Protokoll ja benutzen, um eine solche Quadratwurzel zu finden. Alice könnte auch das Ziel haben, auch ohne Kenntnis des Geheimnisses andere davon zu überzeugen, daß sie Bob ist. Auch weitere Betrugsziele sind denkbar. Es ist unmöglich, alle diese Betrugsziele zu kennen. Aber es muß ausgeschlossen sein, daß die Informationen, die Alice im Protokoll erhält, einen Betrug erleichtern. Die Zero-Knowledge-Eigenschaft sorgt dafür. Sie garantiert nämlich, daß Alice die im Protokoll ausgetauschten Nachrichten auch ohne Beteiligung von Bob so simulieren kann, daß die Verteilung auf den simulierten Nachrichte nicht von der Verteilung auf den echten Nachrichten unterschieden werden kann. Es muß aber nicht nur die Verteilung auf den Nachrichten im Original-Protokoll simulierbar sein. Wenn Alice etwas Betrügerisches mit dem Protokoll tun will, wählt sie die Challenge e möglicherweise anders als gleichverteilt aus, um einen Vorteil zu erhalten. Sie kann die Auswahl von dem Committment x abhängig machen. Auch bei einer solchen Auswahl muß die entstehende Verteilung simulierbar sein, damit klar ist, daß Alice ein keinem Fall etwas unter Verwendung der Protokollinformation machen kann, was sie nicht auch allein könnte.

Wir erklären die Simulation. Im Fiat-Shamir-Protokoll entsteht in jedem Durchlauf ein Nachrichtentripel (x, e, y). Dabei ist x ein gleichverteilt zufälliges Quadrat modulo n in $\{0, \ldots, n-1\}$, e ist eine Zahl in $\{0, 1\}$, die gemäß einer von Alice bestimmten Verteilung (,die auch von x abhängen darf,) ausgewählt wird und y ist eine gleichverteilt zufällige Quadratwurzel von xs^e in $\{0, \ldots, n-1\}$. Tripel (x, e, y) mit derselben Verteilung können auf folgende Weise simuliert werden. Der Simulator wählt eine gleichverteilt zufällige Zahl f in $\{0, 1\}$ und eine gleichverteilt zufällige Zahl $y \in \{0, \ldots, n-1\}$. Dann berechnet der Simulator $x = y^2 s^{-f}$ mod n, wobei s^{-f} das Inverse von s^f mo-

dulo n ist. Schließlich wählt der Simulator gemäß der von Alice ausgesuchten Verteilung eine Zahl e in $\{0,1\}$ aus. Wenn e und f übereinstimmen, gibt der Simulator das Tripel (x,e,y) aus. Dann gilt nämlich $x = y^2 s^{-f} \bmod n$. Andernfalls verwirft der Simulator das Tripel (x,e,y).

Die Erfolgswahrscheinlichkeit für den Simulator ist $1/2$ und der Simulator produziert offensichtlich dieselbe Verteilung wie das Originalprotokoll. Die Zahl x ist ein Zufallsquadrat modulo n in $\{0,\ldots,n-1\}$, die Zahl e wird genauso gewählt wie im Originalprotokoll und die Zahl y ist eine zufällige Quadratwurzel von xs^e modulo n in $\{0,1,\ldots,n-1\}$.

Im allgemeinen fordert man für die Zero-Knowledge-Eigenschaft nur, daß die Verteilung auf den Nachrichten im simulierten Protokoll von der Verteilung der Nachrichten im Originalprotokoll in Polynomzeit nicht zu unterscheiden ist. Dies wird im Rahmen der Komplexitätstheorie mathematisch präzisiert. Wir verweisen auf [33].

15.5 Übungen

Übung 15.5.1. Sei p eine Primzahl, g eine Primitivwurzel mod p, $a \in \{0,1,\ldots,p-2\}$ und $A = g^a \bmod p$. Beschreiben Sie einen Zero-Knowledge-Beweis dafür, daß Alice den diskreten Logarithmus a von A mod p zur Basis g kennt. Der ZK-Beweis ist analog zum dem in Abschnitt 15.4.3 beschriebenen.

Übung 15.5.2. Im Fiat-Shamir-Verfahren sei $n = 143$, $v = 82$, $x = 53$ und $e = 1$. Bestimmen Sie eine gültige Antwort, die die Kenntnis einer Quadratwurzel von v mod n beweist.

Übung 15.5.3 (Feige-Fiat-Shamir-Protokoll). Eine Weiterentwicklung des Fiat-Shamir-Verfahrens ist das Feige-Fiat-Shamir-Protokoll. Eine vereinfachte Version sieht so aus: Alice benutzt einen RSA-Modul n. Sie wählt $s_1,\ldots,s_k$ gleichverteilt zufällig aus $\{1,\ldots,n-1\}^k$ und berechnet $v_i = s_i^2 \bmod n$, $1 \le i \le k$. Ihr öffentlicher Schlüssel ist $(n,v_1,\ldots,v_k)$. Ihr geheimer Schlüssel ist $(s_1,\ldots,s_k)$. Um Bob von ihrer Identität zu überzeugen, wählt Alice $r \in \{1,\ldots,n-1\}$ zufällig, berechnet $x = r^2 \bmod n$ und schickt x an Bob. Bob wählt gleichverteilt zufällig $(e_1,\ldots,e_k) \in \{0,1\}^k$ und schickt diesen Vektor an Alice (Challenge). Alice schickt $y = r\prod_{i=1}^k s_i^{e_i}$ an Bob (Response). Bob verifiziert $y^2 \equiv x\prod_{i=1}^k v_i^{e_i} \bmod p$. Mit welcher Wahrscheinlichkeit kann ein Betrüger in einem Durchgang dieses Protokolls betrügen?

Übung 15.5.4. Modifizieren Sie das Verfahren aus Übung 15.5.3 so, daß seine Sicherheit auf der Schwierigkeit beruht, diskrete Logarithmen zu berechnen.

Übung 15.5.5 (Signatur aus Identifikation). Machen Sie aus dem Protokoll aus Übung 15.5.3 ein Signatur-Verfahren. Die Idee besteht darin, bei Signatur der Nachricht m die Challenge $(e_1, \ldots, e_k)$ durch den Hashwert $h(x \circ m)$ zu ersetzen.

16. Secret Sharing

In Public-Key-Infrastrukturen ist es oft nützlich, private Schlüssel von Teilnehmern rekonstruieren zu können. Wenn nämlich ein Benutzer die Chipkarte mit seinem geheimen Schlüssel verliert, kann er seine verschlüsselt gespeicherten Daten nicht mehr entschlüsseln. Aus Sicherheitsgründen ist es aber wichtig, dass nicht ein einzelner die Möglichkeit hat, geheime Schlüssel zu rekonstruieren. Es ist besser, wenn bei der Rekonstruktion von privaten Schlüsseln mehrere Personen zusammenarbeiten müssen. Die können sich dann gegenseitig kontrollieren. Die Wahrscheinlichkeit sinkt, daß Unberechtigte Zugang zu geheimen Schlüsseln bekommen. In diesem Kapitel wird eine Technik vorgestellt, dieses Problem zu lösen, das *Secret-Sharing.* .

16.1 Prinzip

Wir beschreiben, was Secret-Sharing-Techniken leisten. Seien n und t natürliche Zahlen. In einem (n,t)-Secret-Sharing-Protokoll wird ein Geheimnis auf n Personen aufgeteilt. Jeder hat einen Teil (Share) des Geheimnisses. Wenn sich t dieser Personen zusammentun, können sie das Geheimnis rekonstruieren. Wenn sich aber weniger als t dieser Teilgeheimnisträger zusammentun, können sie keine relevante Information über das Geheimnis erhalten.

16.2 Das Shamir-Secret-Sharing-Protokoll

Seien $n, t \in \mathbb{N}$, $t \leq n$. Wir beschreiben das (n,t)-Secret-Sharing-Protokoll von Shamir [72]. Es verwendet eine Primzahl p und beruht auf folgendem Lemma.

Lemma 16.2.1. *Seien $\ell, t \in \mathbb{N}$, $\ell \leq t$. Weiter seien $x_i, y_i \in \mathbb{Z}/p\mathbb{Z}$, $1 \leq i \leq \ell$, wobei die x_i paarweise verschieden sind. Dann gibt es genau $p^{t-\ell}$ Polynome $b \in (\mathbb{Z}/p\mathbb{Z})[X]$ vom Grad $\leq t-1$ mit $b(x_i) = y_i$, $1 \leq i \leq \ell$.*

Beweis. Das Lagrange-Interpolationsverfahren liefert das Polynom

$$b(X) = \sum_{i=1}^{\ell} y_i \prod_{j=1, j\neq i}^{\ell} \frac{x_j - X}{x_j - x_i}, \tag{16.1}$$

das $b(x_i) = y_i$, $1 \le i \le \ell$ erfüllt. Jetzt muß noch die Anzahl solcher Polynome bestimmt werden.

Sei $b \in (\mathbb{Z}/p\mathbb{Z})[X]$ ein solches Polynom. Schreibe

$$b(X) = \sum_{j=0}^{t-1} b_j X^j, \;\; b_j \in \mathbb{Z}/p\mathbb{Z}, 0 \le j \le t-1.$$

Aus $b(x_i) = y_i$, $1 \le i \le \ell$ erhält man das lineare Gleichungssystem

$$\begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{t-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{t-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_\ell & x_\ell^2 & \cdots & x_\ell^{t-1} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{t-1} \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_\ell \end{pmatrix}. \tag{16.2}$$

Die Teil-Koeffizientenmatrix

$$U = \begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{\ell-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{\ell-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_\ell & x_\ell^2 & \cdots & x_\ell^{\ell-1} \end{pmatrix}$$

ist eine *Vandermonde-Matrix*. Ihre Determinante ist

$$\det U = \prod_{1 \le i < j \le \ell} (x_j - x_i).$$

Weil die x_i nach Voraussetzung paarweise verschieden sind, ist diese Determinante ungleich Null. Der Rang von U ist also ℓ. Daher hat der Kern der Koeffizientenmatrix des linearen Gleichungssystems (16.2) den Rang $t - \ell$ und die Anzahl der Lösungen ist $p^{t-\ell}$. □

Jetzt können wir das Protokoll beschreiben.

16.2.1 Initialisierung

Der Dealer wählt eine Primzahl p, $p \ge n + 1$ und paarweise und von Null verschiedene Elemente $x_i \in \mathbb{Z}/p\mathbb{Z}$, $1 \le i \le n$. Die Elemente von $\mathbb{Z}/p\mathbb{Z}$ werden zum Beispiel durch ihre kleinsten nicht negativen Vertreter dargestellt. Die x_i werden veröffentlicht.

16.2.2 Verteilung der Geheimnisteile

Der Dealer will ein Geheimnis $s \in \mathbb{Z}/p\mathbb{Z}$ verteilen.

1. Er wählt geheime Elemente $a_j \in \mathbb{Z}/p\mathbb{Z}$, $1 \leq j \leq t-1$ und konstruiert daraus das Polynom
$$a(X) = s + \sum_{j=1}^{t-1} a_j X^j. \tag{16.3}$$
Es ist vom Grad $\leq t-1$.
2. Der Dealer berechnet die Geheimnisteile $y_i = a(x_i)$, $1 \leq i \leq n$.
3. Der Dealer gibt dem i-ten Geheimnisträger den Geheimnisteil y_i, $1 \leq i \leq n$.

Das Geheimnis ist also der konstante Term $a(0)$ des Polynoms $a(X)$.

Beispiel 16.2.2. Sei $n = 5$, $t = 3$. Der Dealer wählt $p = 17$, $x_i = i$, $1 \leq i \leq 5$.

Das Geheimnis sei $s = 3$. Der Dealer wählt die geheimen Koeffizienten $a_i = 13 + i$, $1 \leq i \leq 2$. Damit ist also
$$a(X) = 15X^2 + 14X + 3. \tag{16.4}$$
Die Geheimnisteile sind damit $y_1 = a(1) = 15$, $y_2 = a(2) = 6$, $y_3 = a(3) = 10$, $y_4 = a(4) = 10$, $y_5 = a(5) = 6$.

16.2.3 Rekonstruktion des Geheimnisses

Angenommen, t Geheimnisträger arbeiten zusammen. Ihre Geheimnisteile seien $y_i = a(x_i)$, $1 \leq i \leq t$. Dabei ist $a(X)$ das Polynom aus (16.3). Dies kann man durch Umnummerierung der Geheimnisteile immer erreichen. Jetzt gilt
$$a(X) = \sum_{i=1}^{t} y_i \prod_{j=1, j\neq i}^{t} \frac{x_j - X}{x_j - x_i}. \tag{16.5}$$
Dieses Polynom erfüllt nämlich $a(x_i) = y_i$, $1 \leq i \leq t$ und nach Lemma 16.2.1 gibt es genau ein solches Polynom vom Grad höchstens $t-1$. Daher ist
$$s = a(0) = \sum_{i=1}^{t} y_i \prod_{j=1, j\neq i}^{t} \frac{x_j}{x_j - x_i}. \tag{16.6}$$
Die Formel (16.6) wird von den Geheimnisträgern benutzt, um das Geheimnis zu konstruieren.

Beispiel 16.2.3. Wir setzen das Beispiel 16.2.2 fort.

Die ersten drei Geheimnisträger rekonstruieren das Geheimnis. Die Lagrange-Interpolationsformel ergibt
$$a(0) = 15\frac{6}{2} + 6\frac{3}{-1} + 10\frac{2}{2} \bmod 17 = 3. \tag{16.7}$$

16.2.4 Sicherheit

Angenommen, weniger als t Geheimnisträger versuchen gemeinsam, das Geheimnis s zu ermitteln. Ihre Anzahl sei m , $m < t$. Ihre Geheimnisteile seien $y_i, 1 \leq i \leq m$. Dies wird durch Umnummerierung der Geheimnisteile erreicht. Die Geheimnisträger wissen, dass das Geheimnis der konstante Term eines Polynoms $a \in \mathbb{Z}_p[X]$ vom Grad $\leq t-1$, das $a(x_i) = y_i$, $1 \leq i \leq m$ erfüllt. Aus Lemma 16.2.1 erhält man das folgende Resultat.

Lemma 16.2.4. *Für jedes $s' \in \mathbb{Z}/p\mathbb{Z}$ gibt es genau p^{t-m-1} Polynome $a'(X) \in (\mathbb{Z}/p\mathbb{Z})[X]$ vom Grad $\leq t-1$ mit $a'(0) = s'$ und $a'(x_i) = y_i$, $1 \leq i \leq m$.*

Beweis. Da die x_i paarweise und von Null verschieden sind, folgt die Behauptung aus Lemma 16.2.1 mit $\ell = m+1$. □

Lemma 16.2.4 zeigt, dass die m Geheimnisträger keine Information über das Geheimnis bekommen, weil alle möglichen konstanten Terme gleich wahrscheinlich sind.

16.3 Übungen

Übung 16.3.1. Rekonstruieren Sie das Geheimnis in Beispiel 16.2.2 aus den letzten drei Geheimnisanteilen.

Übung 16.3.2. Sei $n = 4$, $t = 2$, $p = 11$, $s = 3$, $a_1 = 2$. Konstruieren Sie $a(X)$ und die Geheimnisanteile y_i, $1 \leq i \leq 4$.

17. Public-Key-Infrastrukturen

Ein großer Vorteil asymmetrischer Kryptoverfahren besteht darin, daß die Schlüsselverwaltung einfacher ist als bei symmetrischen Verschlüsselungsverfahren. Die Schlüssel, die zum Verschlüsseln gebraucht werden, müssen nicht geheimgehalten werden, sondern sie können öffentlich sein. Die privaten Schlüssel müssen aber auch in Public-Key-Systemen geheim bleiben. Außerdem müssen die öffentlichen Schlüssel vor Fälschung und Mißbrauch geschützt werden. Die Verteilung und Speicherung der öffentlichen und privaten Schlüssel geschieht in einer sogenannten *Public-Key-Infrastruktur* (PKI).
In diesem Kapitel werden einige Organisationsprinzipien von Public-Key-Infrastrukturen erläutert.

17.1 Persönliche Sicherheitsumgebung

17.1.1 Bedeutung

Will Bob mit einem Public-Key-System Signaturen erzeugen oder verschlüsselte Nachrichten entschlüsseln, braucht er einen privaten Schlüssel. Dieser Schlüssel muß geheim bleiben, weil jeder, der ihn kennt, Bobs Signatur fälschen oder geheime Nachrichten an Bob entschlüsseln kann. Die Umgebung, in der Bob seinen geheimen Schlüssel ablegt, wird *persönliche Sicherheitsumgebung* oder *Personal Security Environment (PSE)* genannt. Der private Schlüssel darf die PSE nicht verlassen, weil er sonst in eine unsichere Umgebung käme. Darum hat die PSE noch weitere Aufgaben. In der PSE werden Chiffretexte entschlüsselt, weil dazu der private Schlüssel nötig ist. Aus demselben Grund werden in der PSE Dokumente signiert.

Oft werden in der PSE auch die privaten Schlüssel erzeugt. Werden sie anderswo erzeugt, dann sind sie wenigstens dem Erzeuger bekannt. Andererseits ist es bei der Schlüsselerzeugung besonders wichtig, daß keine technischen Fehler gemacht werden. Die Schlüsselerzeugung im RSA-Verfahren erfordert die zufällige Wahl zweier Primzahlen p und q. Ist der Zufallszahlengenerator der PSE schwach, so lassen sich die verwendeten Primzahlen vielleicht rekonstruieren. Das kann dafür sprechen, die Schlüssel in einer vertrauenswürdigen Stelle zu erzeugen, wo der jeweils beste bekannte Zufallszahlengenerator verwendet wird.

17.1.2 Implementierung

Je sensibler die Dokumente sind, die verschlüsselt oder signiert werden, desto sicherer muß die PSE ausgelegt sein. Eine einfache PSE ist ein durch ein Paßwort geschützter Speicherbereich auf der Festplatte von Bobs Workstation. Dieses Paßwort kann zum Beispiel dazu verwendet werde, die Information in der PSE zu entschlüsseln. Die Sicherheit dieser PSE hängt sehr stark von der Sicherheit des verwendeten Betriebssystems ab. Man kann argumentieren, daß Betriebssysteme ohnehin hohe Sicherheit gewährleisten müssen und daher auch die PSE schützen können. Betriebssysteme müssen zum Beispiel verhindern, daß sich ein Unbefugter Administrator-Rechte verschafft. Andererseits ist bekannt, daß man mit entsprechendem Aufwand viele Schutzfunktionen des Betriebssystems umgehen kann und eine Software-PSE daher nicht sicher ist.

Für sehr sicherheitskritische Anwendungen reicht der Schutz durch das Betriebssystem nicht. Sicherer ist es, die PSE auf einer Chipkarte unterzubringen. Bob kann seine Karte immer bei sich haben. Wenn die Karte im Leser steckt, erlaubt sie nur sehr wenige Zugriffe von außen. Die Manipulation ihrer Hard- oder Software ist extrem schwierig (trotzdem hat es auch auf Chipkarten in der Vergangenheit immer wieder erfolgreiche Angriffe gegeben). Leider sind Berechnungen auf Chipkarten sehr langsam. Große Datenmengen lassen sich auf einer Chipkarte nicht in vertretbarer Zeit ver- oder entschlüsseln. Man verschlüsselt daher mit dem öffentlichen Schlüssel des Kommunikationspartners nur Sitzungsschlüssel, die dann zur Verschlüsselung von Dokumenten verwendet werden und in verschlüsselter Form den Schlüsseltexten angehängt werden (siehe Abschnitt 9.1). Der Sitzungsschlüssel wird auf der Chipkarte entschlüsselt. Die Entschlüsselung des gesamten Textes erfolgt dann im PC oder in der Workstation.

17.1.3 Darstellungsproblem

Selbst wenn Alice zum Signieren eine Chipkarte verwendet, treten ernste Sicherheitsprobleme auf: Alice will ein Dokument signieren. Sie startet dazu auf ihrem PC ein Signierprogramm. Dieses Programm schickt das zu signierende Dokument oder seinen Hashwert an die Chipkarte. Dort wird die Signatur berechnet. Der Gegner Oskar kann aber (mit einigem Aufwand) den PC von Alice so manipulieren, daß ein anderes Dokument, als Alice glaubt, an die Chipkarte geschickt und dort signiert wird. Alice sieht ja nicht, was signiert wird, weil die Chipkarte keine Anzeige hat. Sie sieht nur das Dokument von dem sie glaubt, daß es signiert werde. Es ist also möglich, daß Alice Dokumente signiert, die sehr nachteilig für sie sind. Das ist als *Darstellungsproblem* für Signaturen bekannt. Je wichtiger die Dokumente sind, deren digitale Signatur anerkannt wird, desto dramatischer ist das Darstellungsproblem. Das Problem ist gelöst, wenn Alice sieht, was sie signiert. Es ist aber noch unklar,

wie man das in einer sicheren Umgebung am besten realisiert. Es gibt z.B. Versuche, Mobiltelefone zum Signieren zu verwenden. Das Mobiltelefon ist dann die PSE. Der zu signierende Text wird vollständig an das Mobiltelefon übertragen, dort angezeigt, danach gehasht und signiert. Aber das Display eines Mobiltelefons ist sehr klein. Man kann also nur kurze Texte, die signiert werden, anzeigen. Von der Lösung des Darstellungsproblems hängt es ab, ob digitale Signaturen in wirklich sicherheitskritischen Situationen eingesetzt werden können.

17.2 Zertifizierungsstellen

In Public-Key-Systemen ist es nicht nur wichtig, daß Alice ihre privaten Schlüssel schützen kann. Benutzt sie den öffentlichen Schlüssel von Bob, muß sie sicher sein, daß sie tatsächlich Bobs öffentlichen Schlüssel hat. Gelingt es nämlich Oskar, Bobs öffentlichen Schlüssel gegen seinen eigenen auszutauschen, dann kann er Nachrichten entschlüsseln, die für Bob bestimmt waren, und er kann in Bobs Namen digitale Signaturen erzeugen.

Eine Möglichkeit, zu garantieren, daß Teilnehmer in einem Public-Key-System die richtigen öffentlichen Schlüssel der anderen Teilnehmer erhalten, besteht darin, jedem Teilnehmer eine sogenannten *Certification-Authority* (CA) zuzuordnen, der er vertraut. Die CA bürgt mit ihrer Signatur für die Korrektheit und Gültigkeit der öffentlichen Schlüssel der Teilnehmer. Da jeder Teilnehmer den öffentlichen Schlüssel der CA kennt, kann er die Signaturen der CA verifizieren. Wir erklären im folgenden, wie das im einzelnen funktioniert.

17.2.1 Registrierung

Wenn Bob neuer Benutzer des Public-Key-Systems wird, läßt er sich bei der ihm zugeordneten CA registrieren. Er teilt der CA seinen Namen und andere erforderliche persönliche Daten mit. Die CA muß Bobs Informationen verifizieren. Am einfachsten ist es, wenn Bob persönlich zu der CA geht und seinen Ausweis vorlegt. Die CA weist Bob einen geeigneten Benutzernamen zu, der sich von allen anderen Benutzernamen unterscheidet. Unter diesem Namen wird Bob zum Beispiel Signaturen erzeugen. Wenn Bob nicht will, daß sein wirklicher Name bekannt wird, kann er auch ein Pseudonym verwenden. Dann ist nur der CA Bobs wirkliche Identität bekannt.

17.2.2 Schlüsselerzeugung

Bobs öffentliche und private Schlüssel werden in seiner PSE oder von seiner CA erzeugt. Es ist vorteilhaft, wenn Bob seine privaten Schlüssel nicht kennt. Dann kann er sie auch nicht preisgeben. Geschieht die Schlüsselerzeugung in

Bobs PSE, so werden Bobs private Schlüssel in der PSE gespeichert. Die öffentlichen Schlüssel werden der CA mitgeteilt. Werden die Schlüssel von der CA erzeugt, so müssen die privaten Schlüssel auf Bobs PSE gelangen. Die Übertragung der Schlüssel muß natürlich entsprechend gesichert sein.

Für jeden Zweck, etwa Signieren, Verschlüsseln und Authentifizieren, muß ein eigenes Schlüsselpaar erzeugt werden, weil sonst die Sicherheit des Systems gefährdet ist. Das illustriert das folgende Beispiel.

Beispiel 17.2.1. Wenn Alice für Challenge-Response-Authentikation und Signatur dasselbe Schlüsselpaar verwendet, dann können ihre Signaturen folgendermaßen gefälscht werden: Oskar gibt vor, er wolle die Identität von Alice prüfen. Als Challenge schickt er Alice den Hashwert $h(m)$ eines Textes m. Alice signiert diesen Hashwert im Glauben, es handele sich um eine Zufallszahl. Der signierte Hashwert ist aber die Signatur des Textes m. Alice hat also, ohne es zu merken, ein Dokument signiert, das ihr von Oskar vorgelegt wurde.

17.2.3 Zertifizierung

Um eine verifizierbare Verbindung zwischen Bob und seinen öffentlichen Schlüsseln herzustellen, erstellt die CA ein *Zertifikat*. Dieses Zertifikat ist ein von der CA signierter String, der mindestens folgende Informationen enthält:

1. Bobs Benutzernamen oder sein Pseudonym,
2. die zugeordneten öffentlichen Schlüssel,
3. die Bezeichnung der Algorithmen, mit denen die öffentlichen Schlüssel von Bob benutzt werden können,
4. die laufende Nummer des Zertifikats,
5. Beginn und Ende der Gültigkeit des Zertifikats,
6. den Namen der CA,
7. Angaben, ob die Nutzung der Schlüssel auf bestimmte Anwendungen beschränkt ist.

Das Zertifikat wird zusammen mit dem Namen des Benutzers in ein Verzeichnis (Directory) eingetragen. Nur die CA darf in dieses Directory schreiben oder Einträge aus diesem Directory löschen, aber die Benutzer, die der CA zugeordnet sind, können das Directory lesen.

17.2.4 Archivierung

Einige Schlüssel, die in Public-Key-Systemen verwendet werden, müssen archiviert werden. Die Dauer der Aufbewahrung hängt von der Verwendung der Schlüssel ab.

Öffentliche Signaturschlüssel müssen solange aufbewahrt werden, wie die entsprechenden Signaturen noch verifizierbar sein sollen. Signaturen in elektronischen Grundbüchern müssen zum Beispiel viele Jahrzehnte gültig sein.

Für Signaturschlüssel werden Zertifikate gespeichert, damit später ihre Authentizität noch verifiziert werden kann. Private Entschlüsselungsschlüssel müssen solange aufbewahrt werden, wie die entsprechenden Chiffretexte noch entschlüsselbar sein sollen, etwa wenn sie verschlüsselt gespeichert sind. Private Schlüssel werden aber nicht von der CA, sondern in der PSE der Benutzer archiviert. Für Authentikationsschlüssel, private Signaturschlüssel und öffentliche Verschlüsselungsschlüssel ist keine Archivierung erforderlich. Sie werden nur solange benötigt, wie sie zur Authentikation, zum Signieren oder zum Verschlüsseln gebraucht werden.

17.2.5 Personalisierung des PSE

Nach erfolgreicher Registrierung, Schlüsselerzeugung und Zertifizierung überträgt die CA die für den Teilnehmer relevanten Daten in seine PSE. Das sind insbesondere Bobs private Schlüssel, sofern sie von der CA erzeugt wurden. Außerdem kann Bobs Zertifikat und der öffentliche Schlüssel der CA in der PSE gespeichert werden.

17.2.6 Verzeichnisdienst

Bei der CA gibt es ein Verzeichnis (Directory) der von der CA erzeugten Zertifikate mit den entsprechenden Teilnehmernamen. Will Alice die öffentlichen Schlüssel von Bob erfahren, so fragt sie bei dem Directory ihrer CA an, ob Bob dort registriert ist. Wenn ja, erhält sie Bobs Zertifikat. Alice kennt ja den öffentlichen Schlüssel ihrer CA. Sie kann verifizieren, daß das Zertifikat von ihrer CA kommt und sie vertraut ihrer CA. Also hat sie den authentischen öffentlichen Schlüssel von Bob. Wenn Bob nicht bei der CA von Alice registriert ist, kann Alice versuchen, ein Zertifikat für Bob auf Umwegen zu erhalten. Das wird in Abschnitt 17.3 noch genauer beschrieben.

Zertifikate, die Alice häufig braucht, kann sie in ihre PSE aufnehmen. Sie muß aber die Gültigkeit der Zertifikate regelmäßig überprüfen (siehe Abschnitt 17.2.8).

Wenn eine CA für eine große Zahl von Benutzern zuständig ist, kann es bei den Anfragen an die CA Engpässe geben. Eine Möglichkeit, die Effizienz der Directory-Anfragen zu verbessern, besteht darin, das Directory zu replizieren und die Benutzer auf die verschiedenen Kopien aufzuteilen.

Beispiel 17.2.2. Eine Firma möchte alle 50 000 Mitarbeiter zu Teilnehmern einer PKI machen. Die Firma ist über fünf Länder verteilt. Sie möchte nur eine CA in England betreiben. Um die Anfrage beim Directory der CA effizienter zu gestalten, verteilt sie fünf Kopien des Directories auf die fünf Länder und führt selbst das Original. Die Kopien werden zweimal am Tag auf den neuesten Stand gebracht.

17.2.7 Schlüssel-Update

Alle Schlüssel, die in einem Public-Key-System benutzt werden, haben eine bestimmte Gültigkeitsdauer. Rechtzeitig, bevor ein Schlüssel seine Gültigkeit verliert, muß er durch einen neuen Schlüssel ersetzt werden. Dieser neue Schlüssel wird nach seiner Erzeugung von CA und Benutzer ausgetauscht. Der Austausch muß so ablaufen, daß der neue Schlüssel selbst dann nicht kompromittiert wird, wenn der alte Schlüssel bekannt wird.

Folgendes Verfahren, den Schlüssel zu erneuern, ist unsicher: Kurz bevor das Schlüsselpaar von Bob ungültig wird, erzeugt seine CA ein neues Schlüsselpaar für Bob. Sie verschlüsselt den neuen privaten Schlüssel mit Bobs altem öffentlichem Schlüssel und sendet ihn an Bob. Wenn Oskar diese Kommunikation protokolliert und wenn es ihm später gelingt, Bobs alten privaten Schlüssel zu finden, dann kann er auch den neuen privaten Schlüssel bestimmen. In diesem Fall hängt die Sicherheit des neuen Schlüssels von der Sicherheit des alten Schlüssels ab, und es ist sinnlos, einen neuen Schlüssel auszutauschen.

Statt dessen kann man Varianten des Diffie-Hellman-Schlüsselaustauschs verwenden, die die Man-In-The-Middle-Attack unmöglich machen (siehe Abschnitt 9.5).

17.2.8 Rückruf von Zertifikaten

Manchmal muß die CA ein Zertifikat für ungültig erklären, obwohl seine Gültigkeitsdauer noch nicht abgelaufen ist.

Beispiel 17.2.3. Bob hat seine Chipkarte bei einer Bootsfahrt verloren. Die Smartcard liegt jetzt auf dem Meeresgrund. Auf der Smartcard ist Bobs privater Signaturschlüssel. Den kann er jetzt nicht mehr zum Signieren gebrauchen, weil der Schlüssel nur auf der Chipkarte steht und sonst nirgendwo. Daher ist auch das Zertifikat, das den entsprechenden öffentlichen Schlüssel enthält, nicht mehr gültig. Es muß von der CA für ungültig erklärt werden.

Die ungültig gewordenen Zertifikate schreibt die CA in eine Liste, die *Certificate Revocation List* (CRL). Diese ist Teil des Directories der CA. Ein Eintrag in der CRL enthält die Seriennummer des Zertifikats, den Zeitpunkt, zu dem das Zertifikat ungültig wurde und möglicherweise andere Informationen, wie z.B. die Gründe für die Ungültigkeit. Der Eintrag wird von der CA signiert.

17.2.9 Zugriff auf ungültige Schlüssel

Wird ein ungültiger, aber archivierter Schlüssel benötigt, wird dieser vom Archiv bereitgestellt werden. Der Benutzer, der diesen Schlüssel erhalten möchte, muß seine Berechtigung gegebenenfalls nachweisen.

Beispiel 17.2.4. Von der CA werden die Signaturschlüssel jeden Monat gewechselt. Bob signiert einen Auftrag an Alice, bestreitet aber drei Monate später, den Auftrag erteilt zu haben. Alice möchte beweisen, daß der Auftrag tatsächlich erteilt wurde. Sie braucht dafür den öffentlichen Schlüssel, der aber schon seit zwei Monaten ungültig ist und darum nicht mehr im Directory der CA steht.

17.3 Zertifikatsketten

Wenn Bob und Alice nicht zu derselben CA gehören, kann Alice den öffentlichen Schlüssel von Bob nicht einfach aus dem Directory ihrer CA erfahren. Sie kann den öffentlichen Schlüssel von Bob aber indirekt erhalten.

Beispiel 17.3.1. Alice ist bei der CA für Hessen registriert. Bob ist bei der CA für das Saarland registriert. Alice kennt also den öffentlichen Schlüssel der CA Hessen, aber nicht den öffentlichen Schlüssel der CA des Saarlandes. Von ihrer CA erhält Alice ein Zertifikat für den öffentlichen Schlüssel der CA des Saarlandes. Aus dem Directory der CA des Saarlandes erhält Alice ein Zertifikat für den öffentlichen Schlüssel von Bob. Unter Verwendung des öffentlichen Schlüssels ihrer CA kann Alice verifizieren, daß der öffentliche Schlüssel der CA des Saarlandes korrekt ist. Damit ist Alice also im Besitz des öffentlichen Schlüssels der CA des Saarlandes. Diesen Schlüssel kann sie nun verwenden, um das Zertifikat für Bobs öffentlichen Schlüssel zu verifizieren.

Wie in Beispiel 17.3.1 beschrieben, kann Alice eine *Zertifikatskette* verwenden, um den authentischen öffentlichen Schlüssel von Bob zu erhalten, selbst wenn Bob zu einer anderen CA gehört. Formal kann man eine solche Kette so beschreiben: Für eine Zertifizierungsstelle CA und einen Teilnehmer U bezeichne mit $\mathrm{CA}\{U\}$ ein Zertifikat, das den Namen von U an den öffentlichen Schlüssel von U bindet. Dabei kann U entweder ein Benutzer oder eine CA sein. Eine Zertifikatskette, die für Alice den öffentlichen Schlüssel von Bob zertifiziert, ist dann eine Folge

$$\mathrm{CA}_1\{\mathrm{CA}_2\}, \mathrm{CA}_2\{\mathrm{CA}_3\}, \ldots, \mathrm{CA}_{k-1}\{\mathrm{CA}_k\}, \mathrm{CA}_k\{\mathrm{Bob}\}.$$

Dabei ist CA_1 die CA, bei der Alice registriert ist. Alice verifiziert diese Zertifikatskette, indem sie mit dem öffentlichen Schlüssel von CA_1 das erste Zertifikat prüft und dabei den öffentlichen Schlüssel von CA_2 erhält, mit dem öffentlichen Schlüssel von CA_2 das Zertifikat für CA_3 prüft usw. und zum Schluß mit dem öffentlichen Schlüssel von CA_k das Zertifikat für Bob prüft.

Dieses Verfahren setzt voraus, daß Vertrauen transitiv ist, d.h. wenn sich U_1 auf U_2 verläßt und U_2 auf U_3, dann traut U_1 auch U_3.

Lösungen der Übungsaufgaben

Übung 2.12.1 Sei $z = \lfloor \alpha \rfloor = \max\{x \in \mathbb{Z} : x \leq \alpha\}$. Dann ist $\alpha - z \geq 0$. Außerdem ist $\alpha - z < 1$, denn wäre $\alpha - z \geq 1$, dann wäre $\alpha - (z+1) \geq 0$ im Widerspruch zur Maximalität von α. Insgesamt ist also $0 \leq \alpha - z < 1$ oder $\alpha - 1 < z \leq \alpha$. Da es aber in diesem Intervall nur eine ganze Zahl gibt, ist z diese eindeutig bestimmte Zahl.

Übung 2.12.3 Die Teiler von 195 sind ± 1, ± 3, ± 5, ± 13, ± 15, ± 39, ± 65, ± 195.

Übung 2.12.5 $1243 \bmod 45 = 28$, $-1243 \bmod 45 = 17$.

Übung 2.12.7 Angenommen, m teilt die Differenz $b - a$. Sei $a = q_a m + r_a$ mit $0 \leq r_a < m$ und sei $b = q_b m + r_b$ mit $0 \leq r_b < m$. Dann ist $r_a = a \bmod m$ und $r_b = b \bmod m$. Außerdem ist

$$b - a = (q_b - q_a)m + (r_b - r_a). \tag{17.1}$$

Weil m ein Teiler von $b - a$ ist, folgt aus (17.1), daß m auch ein Teiler von $r_b - r_a$ ist. Weil aber $0 \leq r_b, r_a < m$ ist, gilt

$$-m < r_b - r_a < m.$$

Weil m ein Teiler von $r_b - r_a$ ist, folgt daraus $r_b - r_a = 0$, also $a \bmod m = b \bmod m$.

Sei umgekehrt $a \bmod m = b \bmod m$. Wir benutzen dieselben Bezeichnungen wie oben und erhalten $b - a = (q_b - q_a)m$. Also ist m ein Teiler von $b - a$.

Übung 2.12.8 Es gilt $225 = 128 + 64 + 32 + 1 = 2^7 + 2^6 + 2^5 + 2^0$. Also ist 11100001 die Binärdarstellung von 225. Die Hexadezimaldarstellung gewinnt man daraus, indem man von hinten nach vorn die Binärdarstellung in Blöcke der Länge vier aufteilt und diese als Ziffern interpretiert. Wir bekommen also 1110 0001, d.h. $14 * 16 + 1$. Die Ziffern im Hexadezimalsystem sind $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F$. Damit ist E1 die Hexadezimaldarstellung von 225.

Übung 2.12.11 Wir müssen zeigen, daß es positive Konstanten B und C gibt mit der Eigenschaft, daß für alle $n > B$ gilt $f(n) \leq Cn^d$. Man kann z.B. $B = 1$ und $C = \sum_{i=0}^{d} |a_i|$ wählen.

Übung 2.12.13 1. Jeder Teiler von $a_1, \ldots, a_k$ ist auch ein Teiler von a_1 und $\gcd(a_2, \ldots, a_k)$ und umgekehrt. Daraus folgt die Behauptung.

2. Die Behauptung wird durch Induktion über k bewiesen. Für $k = 1$ ist sie offensichtlich korrekt. Sei also $k > 1$ und gelte die Behauptung für alle $k' < k$. Dann gilt $\gcd(a_1, \ldots, a_k)\mathbb{Z} = a_1\mathbb{Z} + \gcd(a_2, \ldots, a_k)\mathbb{Z} = a_1\mathbb{Z} + a_2\mathbb{Z} + \ldots + a_k\mathbb{Z}$ nach 1., Theorem 2.7.5 und der Induktionsannahme.

3. und 4. werden analog bewiesen.

5. Diese Behauptung wird mittels Korollar 2.7.8 durch Induktion bewiesen.

Übung 2.12.15 Wir wenden den erweiterten euklidischen Algorithmus an und erhalten folgende Tabelle

k	0	1	2	3	4	5	6
r_k	235	124	111	13	7	6	1
q_k		1	1	8	1	1	
x_k	1	0	1	1	9	10	19
y_k	0	1	1	2	17	19	36

Damit ist $\gcd(235, 124) = 1$ und $19 * 235 - 36 * 124 = 1$.

Übung 2.12.17 Wir verwenden die Notation aus dem erweitern euklidischen Algorithmus. Es gilt $S_0 = T_{n+1}$ und daher $x_{n+1} = u_1$ und $y_{n+1} = u_0$. Weiter ist S_n die Einheitsmatrix, also insbesondere $u_n = 1 = r_n / \gcd(a, b)$ und $u_{n+1} = 0 = r_{n+1} / \gcd(a, b)$. Schließlich haben wir in (2.8) gesehen, daß die Folge (u_k) derselben Rekursion genügt wie die Folge (r_k). Daraus folgt die Behauptung.

Übung 2.12.19 Die Bruchdarstellung einer rationalen Zahl $\neq 0$ ist eindeutig, wenn man verlangt, daß der Nenner positiv und Zähler und Nenner teilerfremd sind. Es genügt daher, zu zeigen, daß der euklidische Algorithmus angewandt auf a, b genauso viele Iterationen braucht wie der euklidische Algorithmus angewandt auf $a/\gcd(a, b), b/\gcd(a, b)$. Das folgt aber aus der Konstruktion.

Übung 2.12.21 Nach Korollar 2.7.7 gibt es x, y, u, v mit $xa + ym = 1$ und $ub + vm = 1$. Daraus folgt $1 = (xa + ym)(ub + vm) = (xu)ab + m(xav + yub + yvm)$. Dies impliziert die Behauptung.

Übung 2.12.23 Ist n zusammengesetzt, dann kann man $n = ab$ schreiben mit $a, b > 1$. Daraus folgt $\min\{a, b\} \leq \sqrt{n}$. Weil nach Theorem 2.11.2 dieses Minimum einen Primteiler hat, folgt die Behauptung.

Übung 3.23.1 Einfache Induktion.

Übung 3.23.3 Wenn e und e' neutrale Elemente sind, gilt $e = e'e = e'$.

Übung 3.23.5 Wenn e neutrales Element ist und $e = ba = ac$ ist, dann folgt $b = be = b(ac) = (ba)c = c$.

Übung 3.23.7 Es gilt $4 * 6 \equiv 0 \equiv 4 * 3 \mod 12$, aber $6 \not\equiv 3 \mod 12$.

Übung 3.23.9 Sei R ein kommutativer Ring mit Einselement e und bezeichne R^* die Menge aller invertierbaren Elemente in R. Dann ist $e \in R^*$. Seien a und b invertierbar in R mit Inversen a^{-1} und b^{-1}. Dann gilt $aba^{-1}b^{-1} = aa^{-1}bb^{-1} = e$. Also ist $ab \in R^*$. Außerdem hat jedes Element von R^* definitionsgemäß ein Inverses.

Übung 3.23.11 Sei $g = \gcd(a, m)$ ein Teiler von b. Setze $a' = a/g$, $b' = b/g$ und $m' = m/g$. Dann ist $\gcd(a', m') = 1$. Also hat nach Theorem 3.6.2 die Kongruenz $a'x' \equiv b' \bmod m'$ eine mod m' eindeutig bestimmte Lösung. Sei x' eine solche Lösung. Dann gilt $ax' \equiv b \bmod m$. Für alle $y \in \mathbb{Z}$ erhält man daraus $a(x' + ym') = b + a'ym \equiv b \bmod m$. Daher sind alle $x = x' + ym'$, $y \in \mathbb{Z}$ Lösungen der Kongruenz $ax \equiv b \bmod m$. Wir zeigen, daß alle Lösungen so aussehen. Sei x eine Lösung. Dann ist $a'x \equiv b' \bmod m'$. Also ist $x \equiv x' \bmod m'$ nach Theorem 3.6.2 und das beendet den Beweis.

Übung 3.23.13 Die invertierbaren Restklassen mod 25 sind $a + 25\mathbb{Z}$ mit $a \in \{1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 13, 14, 16, 17, 18, 19, 21, 22, 23, 24\}$.

Übung 3.23.14 Seien a und b ganze Zahlen ungleich Null. Ohne Beschränkung der Allgemeinheit nehmen wir an, daß beide positiv sind. Die Zahl ab ist Vielfaches von a und b. Also gibt es ein gemeinsames Vielfaches von a und von b. Da jedes solche gemeinsame Vielfache wenigstens so groß wie a ist, gibt es ein kleinstes gemeinsames Vielfaches. Das ist natürlich eindeutig bestimmt.

Übung 3.23.15 Induktion über die Anzahl der Elemente in X. Hat X ein Element, so hat Y auch ein Element, nämlich das Bild des Elementes aus X. Hat X n Elemente und ist die Behauptung für $n - 1$ gezeigt, so wählt man ein Element $x \in X$ und entfernt x aus X und $f(x)$ aus Y. Dann wendet man die Induktionsvoraussetzung an.

Übung 3.23.16 Die Untergruppe ist $\{a + 17\mathbb{Z} : a = 1, 2, 4, 8, 16, 15, 13, 9\}$.

Übung 3.23.18

a	2	4	7	8	11	13	14
$\operatorname{ord} a + 15\mathbb{Z}$	4	2	4	4	2	4	2

Übung 3.23.20 Wir zeigen zuerst, daß alle Untergruppen von G zyklisch sind. Sei H eine solche Untergruppe. Ist sie nicht zyklisch, so sind alle Elemente in H von kleinerer Ordnung als $|H|$. Nach Theorem 3.9.5 gibt es für jeden Teiler e von $|G|$ genau $\varphi(e)$ Elemente der Ordnung d in G, nämlich die Elemente g^{xd} mit $1 \leq x \leq |G|/d$ und $\gcd(x, |G|/d) = 1$. Dann folgt aber aus Theorem 3.8.4, daß H weniger als $|H|$ Elemente hat. Das kann nicht sein. Also ist H zyklisch.

Sei nun g ein Erzeuger von G und sei d ein Teiler von $|G|$. Dann hat das Element $h = g^{|G|/d}$ die Ordnung d; es erzeugt also eine Untergruppe H von G der Ordnung d. Wir zeigen, daß es keine andere gibt. Die Untergruppe H hat nach obigem Argument genau $\varphi(d)$ Erzeuger. Diese Erzeuger haben alle

die Ordnung d. Andererseits sind das auch alle Elemente der Ordnung d in G. Damit ist H die einzige zyklische Untergruppe von G der Ordnung d und da alle Untergruppen von G zyklisch sind, ist H auch die einzige Untergruppe der Ordnung d von G.

Übung 3.23.22 Nach Theorem 3.9.2 ist die Ordnung von g von der Form $\prod_{p||G|} p^{x(p)}$ mit $0 \leq x(p) \leq e(p) - f(p)$ für alle $p \mid |G|$. Nach Definition von $f(p)$ gilt aber sogar $x(p) = e(p) - f(p)$ für alle $p \mid |G|$.

Übung 3.23.24 Nach Korollar 3.9.3 ist die Abbildung wohldefiniert. Aus den Potenzgesetzen folgt, daß die Abbildung ein Homomorphismus ist. Weil g ein Erzeuger von G ist, folgt die Surjektivität. Aus Korollar 3.9.3 folgt schließlich die Injektivität.

Übung 3.23.27 2, 3, 5, 7, 11 sind Primitivwurzeln mod 3, 5, 7, 11, 13.

Übung 4.16.1 Der Schlüssel ist 8 und der Klartext ist BANKGEHEIMNIS.

Übung 4.16.3 Die Entschlüsselungsfunktion, eingeschränkt auf des Bild der Verschlüsselungsfunktion, ist deren Umkehrfunktion.

Übung 4.16.5 Die Konkatenation ist offensichtlich assoziativ. Das neutrale Element ist der leere String ε. Die Halbgruppe ist keine Gruppe, weil die Elemente im allgemeinen keine Inversen haben.

Übung 4.16.7 1. Kein Verschlüsselungssystem, weil die Abbildung nicht injektiv ist, also keine Entschlüsselungsfunktion definiert werden kann. Ein Beispiel: Sei $k = 2$. Der Buchstabe A entspricht der Zahl 0, die auf 0, also auf A, abgebildet wird. Der Buchstabe N entspricht der Zahl 13, die auf $2 * 13 \mod 26 = 0$, also auch auf A, abgebildet wird. Die Abbildung ist nicht injektiv und nach Übung 4.16.3 kann also kein Verschlüsselungsverfahren vorliegen.

2. Das ist ein Verschlüsselungssystem. Der Klartext- und Schlüsseltextraum ist Σ^*. Der Schlüsselraum ist $\{1, 2, \ldots, 26\}$. Ist k ein Schlüssel und $(\sigma_1, \sigma_2, \ldots, \sigma_n)$ ein Klartext, so ist $(k\sigma_1 \mod 26, \ldots, k\sigma_n \mod 26\}$ der Schlüsseltext. Das beschreibt die Verschlüsselungsfunktion zum Schlüssel k. Die Entschlüsselungsfunktion erhält man genauso. Man ersetzt nur k durch sein Inverses mod 26.

Übung 4.16.9 Die Anzahl der Bitpermutationen auf $\{0,1\}^n$ ist $n!$. Die Anzahl der zirkulären Links- oder Rechtsshifts auf dieser Menge ist n

Übung 4.16.11 Die Abbildung, die 0 auf 1 und umgekehrt abbildet, ist eine Permutation, aber keine Bitpermutation.

Übung 4.16.13 Gruppeneigenschaften sind leicht zu verifizieren. Wir zeigen, daß S_3 nicht kommutativ ist. Es gilt

$$\begin{pmatrix}1&2&3\\3&2&1\end{pmatrix} \circ \begin{pmatrix}1&2&3\\1&3&2\end{pmatrix} = \begin{pmatrix}1&2&3\\2&3&1\end{pmatrix}$$

aber

$$\begin{pmatrix}1&2&3\\1&3&2\end{pmatrix} \circ \begin{pmatrix}1&2&3\\3&2&1\end{pmatrix} = \begin{pmatrix}1&2&3\\3&1&2\end{pmatrix}.$$

Übung 4.16.15 ECB-Mode: 011100011100
CBC-Mode: 011001010000
CFB-Mode: 100010001000
OFB-Mode: 101010101010.

Übung 4.16.17 Definiere eine Blockchiffre mit Blocklänge n folgendermaßen: Der Schlüssel ist der Koeffizientenvektor $(c_1, \ldots, c_n)$. Ist $w_1 w_2 \ldots w_n$ ein Klartextwort, so ist das zugehörige Schlüsseltextwort $w_{n+1} w_{n+2} \ldots w_{2n}$ definiert durch

$$w_i = \sum_{j=1}^{n} c_j w_{i-j} \bmod 2, \quad n < i \leq 2n.$$

Dies ist tatsächlich eine Blockchiffre, weil die Entschlüsselung gemäß der Formel

$$w_i = w_{n+i} + \sum_{j=1}^{n-1} c_j w_{n+i-j} \bmod 2, \quad 1 \leq i \leq n$$

erfolgen kann. Wählt man als Initialisierungsvektor den Stromchiffreschlüssel $k_1 k_2 \ldots k_n$ und $r = n$, so erhält man die Stromchiffre.

Übung 4.16.19 Ist

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix},$$

so ist $\det A = a_{1,1}a_{2,2}a_{3,3} - a_{1,1}a_{2,3}a_{3,2} - a_{1,2}a_{2,1}a_{3,3} + a_{1,2}a_{2,3}a_{3,1} + a_{1,3}a_{2,1}a_{3,2} - a_{1,3}a_{2,2}a_{3,1}$.

Übung 4.16.21 Die Inverse ist

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix}.$$

Übung 4.16.22 Wir wählen als Schlüssel die Matrix

$$A = \begin{pmatrix} x & 0 & 0 \\ 0 & y & 0 \\ 0 & 0 & z \end{pmatrix}, \quad b = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}.$$

Dann müssen die Kongruenzen

$$\begin{aligned} 17x &\equiv 6 \bmod 26 \\ 14y &\equiv 20 \bmod 26 \\ 19z &\equiv 19 \bmod 26 \end{aligned}$$

gelten. Die sind alle drei lösbar, nämlich mit $x = 8$, $y = 20$, $z = 1$. Damit ist die affin lineare Chiffre bestimmt.

Übung 5.8.1 1. Die Ereignisse S und $\emptyset$ schließen sich gegenseitig aus. Daher gilt $1 = \Pr(S) = \Pr(S \cup \emptyset) = \Pr(S) + \Pr(\emptyset) = 1 + \Pr(\emptyset)$. Daraus folgt $\Pr(\emptyset) = 0$.

2. Setze $C = B \setminus A$. Dann schließen sich die Ereignisse A und C gegenseitig aus. Damit ist $\Pr(B) = \Pr(A \cup C) = \Pr(A) + \Pr(C)$. Da $\Pr(C) \geq 0$ ist, folgt $\Pr(B) \geq \Pr(A)$.

Übung 5.8.3 Mit K bezeichne Kopf und mit Z Zahl. Dann ist die Ergebnismenge {KK,ZZ,KZ,ZK}. Die Wahrscheinlichkeitsverteilung ordnet jedem Elementarereignis die Wahrscheinlichkeit 1/4 zu. Das Ereignis "wenigstens eine Münze zeigt Kopf" ist {KK,KZ,ZK}. Seine Wahrscheinlichkeit ist 3/4.

Übung 5.8.5 Das Ereignis "beide Würfel zeigen ein verschiedenes Ergebnis" ist $A = \{12,13,14,15,16,17,18,19,21,13,\ldots,65\}$. Seine Wahrscheinlichkeit ist 5/6. Das Ereignis "die Summe der Ergebnisse ist gerade" ist $\{11,13,15,22,24,26,\ldots,66\}$. Seine Wahrscheinlichkeit ist 1/2. Der Durchschnitt beider Ereignisse ist $\{13,15,24,26,\ldots,64\}$. Seine Wahrscheinlichkeit ist 1/3. Die Wahrscheinlichkeit von A unter der Bedingung B ist damit 2/3.

Übung 5.8.7 Sei n die Ordnung von G. Wir wenden die Ergebnisse von Abschnitt 5.3 an. Der "Geburtstag" eines Exponenten e_i ist das Gruppenelement g^{e_i}. Es gibt also n "Geburtstage". In Abschnitt 5.3 wurde gezeigt, daß für $k > (1 + \sqrt{1 + (8 \ln 2)n})/2$ die gesuchte Wahrscheinlichkeit größer als 1/2 ist.

Übung 5.8.9 Wir wenden das Geburtstagsparadox an. Es ist $n = 10^4$. Wir brauchen also $k \geq (1 + \sqrt{1 + 8 * 10^4 * \log 2})/2 \geq 118.2$ Leute.

Übung 5.8.10 Nach Definition der perfekten Sicherheit müssen wir prüfen, ob $\Pr(\mathbf{p}|\mathbf{c}) = \Pr(\mathbf{p})$ gilt für jeden Chiffretext $\mathbf{c}$ und jeden Klartext $\mathbf{p}$. Für $n \geq 2$ ist das falsch. Wir geben ein Gegenbeispiel. Sei $\mathbf{p} = (0,0)$ und $\mathbf{c} = (0,0)$. Dann ist $\Pr(\mathbf{p}) = 1/4$ und $\Pr(\mathbf{p}|\mathbf{c}) = 1$.

Übung 6.5.1 Der Schlüssel ist

$$K = 000100110011010001010111011110011001101110111100110111111111110001.$$

Der Klartext ist

$$P = 000000010010001101000101011001111000100110101011110011011110111 1.$$

Damit gilt für die Generierung der Rundenschlüssel

$$
\begin{aligned}
C_0 &= 11110000110011001010101111\\
D_0 &= 01010101011001100111100011111\\
v &= 1\\
C_1 &= 11100001100110010101011111\\
D_1 &= 10101010110011001111000111110\\
v &= 1\\
C_2 &= 11000011001100101010111111\\
D_2 &= 01010101100110011110001111101.
\end{aligned}
$$

In der ersten Runde der Feistelchiffre ist

$$
\begin{aligned}
L_0 &= 11001100000000000110011001111111\\
R_0 &= 11110000101010101111000010101010\\
k_1 &= 000110110000000101110111111111100011100000011100 10\\
E(R_0) &= 01111010000101010101010101111010000101010101010 1\\
B &= 01100001000101111011101010000110011001010010011 1.
\end{aligned}
$$

S	1	2	3	4	5	6	7	8
$Wert$	5	12	8	2	11	5	9	7
C	0101	1100	1000	0010	1011	0101	1001	0111

$$
\begin{aligned}
f_{k_1}(R_0) &= 00000011010010111010100110111011\\
L_1 &= 11110000101010101111000010101010\\
R_1 &= 11001111010010110110010101000100.
\end{aligned}
$$

In der zweiten Runde der Feistelchiffre ist

$$
\begin{aligned}
L_1 &= 11110000101010101111000010101010\\
R_1 &= 11001111010010110110010101000100\\
k_2 &= 011110011010111011011001110110111100100111100101\\
E(R_1) &= 01100101111010100101011010110000101010100000 1001\\
B &= 00011100010001001000111101101011011000111110 1100.
\end{aligned}
$$

S	1	2	3	4	5	6	7	8
$Wert$	4	8	13	3	0	10	10	14
C	0100	1000	1101	0011	0000	1010	1010	1110

$$
\begin{aligned}
f_{k_2}(R_1) &= 10111100011010101000010100100001\\
L_2 &= 11001111010010110110010101000100\\
R_2 &= 01001100110000000111010110001011.
\end{aligned}
$$

Übung 6.5.3 Wir beweisen die Behauptung zuerst für jede Runde. Man verifiziert leicht, daß $E(\overline{R}) = \overline{E(R)}$ gilt, wobei E die Expansionsfunktion des DES und $R \in \{0,1\}^{32}$ ist. Ist $i \in \{1,2,\ldots,16\}$ und $K_i(k)$ der i-te DES-Rundenschlüssel für den DES-Schlüssel k, dann gilt ebenso $K_i(\overline{k}) = \overline{K_i(k)}$. Wird also k durch $\overline{k}$ ersetzt, so werden alle Rundenschlüssel K durch $\overline{K}$ ersetzt. Wird in einer Runde R durch $\overline{R}$ und K durch $\overline{K}$ ersetzt, so ist gemäß (6.3) die Eingabe für die S-Boxen $\overline{E(R)} \oplus \overline{K}$. Nun gilt $a \oplus b = \overline{a} \oplus \overline{b}$ für alle $a, b \in \{0,1\}$. Daher ist die Eingabe für die S-Boxen $E(R) \oplus K$. Da die initiale Permutation mit der Komplementbildung vertauschbar ist, gilt die Behauptung.

Übung 6.5.5 1.) Dies ergibt sich unmittelbar aus der Konstruktion.

2.) Sei $K_i = (K_{i,0},\ldots,K_{i,47})$ der i-te Rundenschlüssel und sei $C_i = (C_{i,0},\ldots,C_{i,27})$ und $D_i = (D_{i,0},\ldots,D_{i,27})$, $1 \le i \le 16$.

Es gilt $K_i = \mathrm{PC2}(C_i, D_i)$. Die Funktion PC2 wählt Einträge ihrer Argumente gemäß Tabelle 6.5 aus. Die zugehörige Auswahlfunktion für die Indizes sei g. Es ist also $g(1) = 14$, $g(2) = 17$ etc. Die Funktion g ist injektiv, aber nicht surjektiv, weil $9, 18, 22, 25$ keine Funktionswerte sind. Die inverse Funktion auf der Bildmenge sei g^{-1}. Sei $i \in \{0,\ldots,26\}$. Wir unterscheiden zwei Fälle. Im ersten ist $i+1 \notin \{9,18,22,25\}$; $i+1$ ist also ein Bild unter g. Aus der ersten Behauptung der Übung und wegen $K_1 = K_{16}$ folgt $C_{1,i} = C_{16,i+1} = K_{16,g^{-1}(i+1)} = K_{1,g^{-1}(i+1)} = C_{1,i+1}$. Im zweiten Fall ist $i+1 \in \{9,18,22,25\}$. Dann ist i ein Bild unter g und es folgt wie oben $C_{16,i} = C_{16,i+1} = K_{16,g^{-1}(i+1)} = K_{1,g^{-1}(i+1)} = C_{1,i+1}$. Damit ist gezeigt, daß $C_{1,0} = C_{1,1} = \ldots = C_{1,8}$, $C_{1,9} = \ldots = C_{1,17}$, $C_{1,18} = \ldots = C_{1,21}$, $C_{1,22} = \ldots = C_{1,24}$ und $C_{1,25} = \ldots = C_{1,27}$. Man zeigt $C_{1,8} = C_{1,9}$, $C_{1,17} = C_{1,18}$, $C_{1,21} = C_{1,22}$ und $C_{1,24} = C_{1,25}$ analog, aber unter Verwendung von $K_1 = K_2$. Entsprechend beweist man die Behauptung für D_i.

3.) Man kann entweder alle Bits von C_1 auf 1 oder 0 setzen und für D_1 genauso. Das gibt vier Möglichkeiten.

Übung 6.5.6 Alle sind linear bis auf die S-Boxen. Wir geben ein Gegenbeispiel für die erste S-Box. Es ist $S_1(000000) = 1110$, $S_1(111111) = 1101$, aber $S_1(000000) \oplus S_1(111111) = 1110 \oplus 1101 = 0011 \neq 1101 = S_1(111111) = S_1(000000 \oplus 111111)$.

Übung 7.6.2 `InvShiftRows`: Zyklischer Rechtsshift um c_i Positionen mit den Werten c_i aus Tabelle 7.1.

`InvSubBytes`: $b \mapsto (A_{-1}(b \oplus c))^{-1}$. Diese Funktion ist in Tabelle 17.1 dargestellt. Diese Tabelle ist folgendermaßen zu lesen. Um den Funktionswert von $\{uv\}$ zu finden, sucht man das Byte in Zeile u und Spalte v, So ist zum Beispiel `InvSubBytes({a5})` $= \{46\}$.

`InvMixColumns`: Das ist die lineare Transformation

$$s_j \leftarrow \begin{pmatrix} \{0e\} & \{0b\} & \{0d\} & \{09\} \\ \{09\} & \{0e\} & \{0b\} & \{0d\} \\ \{0d\} & \{09\} & \{0e\} & \{0b\} \\ \{0b\} & \{0d\} & \{09\} & \{0e\} \end{pmatrix} s_j \quad , 0 \le j < \mathrm{Nb}.$$

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Tabelle 17.1. `InvSubBytes`

Übung 8.6.1 Es ist $2^{1110} \equiv 1024 \bmod 1111$.

Übung 8.6.3 Die kleinste Pseudoprimzahl zur Basis 2 ist 341. Es ist $341 = 11 * 31$ und $2^{340} \equiv 1 \bmod 341 = 1$.

Übung 8.6.5 Sei n eine Charmichael-Zahl. Nach Definition ist sie keine Primzahl und nach Theorem 8.3.1 ist sie quadratfrei, also keine Primzahlpotenz. Also hat n wenigstens zwei Primteiler. Sei $n = pq$ mit Primfaktoren p, q, $p > q$. Nach Theorem 8.6.5 ist $p - 1$ ein Teiler von $n - 1 = pq - 1 = (p-1)q + q - 1$. Daraus folgt, daß $p - 1$ ein Teiler von $q - 1$ ist. Aber das ist unmöglich, weil $0 < q - 1 < p - 1$ ist. Damit ist die Behauptung bewiesen.

Übung 8.6.7 Wir beweisen, daß 341 zusammengesetzt ist. Dazu schreiben wir $340 = 4 * 85$. Es ist $2^{85} \equiv 32 \bmod 341$ und $2^{170} \equiv 1 \bmod 341$. Also ist n zusammengesetzt.

Übung 8.6.9 Die kleinste 512-Bit-Primzahl ist $2^{512} + 3$.

Übung 9.7.1 Ist $de - 1$ ein Vielfaches von $p - 1$ und von $q - 1$, so zeigt man wie im Beweis von Theorem 9.3.4, daß $m^{ed} \equiv m \bmod p$ und $m^{ed} \equiv m \bmod q$ für jedes $m \in \{0, 1, \ldots, n-1\}$ gilt, woraus nach dem chinesischen Restsatz $m^{ed} \equiv m \bmod n$ folgt.

Übung 9.7.3 Setze $p = 223$, $q = 233$, $n = 51959$, $e = 5$. Dann ist $d = 10301$, $m = 27063$, $c = 50042$.

Übung 9.7.5 Wir skizzieren einen einfachen Intervallschachtelungsalgorithmus. Setze $m_0 = 1$, $m_1 = c$. Dann wiederhole folgende Berechnungen, bis $m_1^e = c$ oder $m_0 = m_1$ ist: Setze $x = \lfloor (m_1 - m_0)/2 \rfloor$. Wenn $x^e \geq c$ ist, dann setze $m_1 = x$. Sonst setze $m_0 = x$. Ist nach der letzten Iteration $m_1^e = c$, so ist die e-te Wurzel von c gefunden. Andernfalls existiert sie nicht.

Übung 9.7.7 Es werden 16 Quadrierungen und eine Multiplikation benötigt.

Übung 9.7.9 Man berechnet die Darstellung $1 = xe + yf$ und dann $c_e^x c_f^y = m^{xe+yf} = m$.

Übung 9.7.11 Es ist $p = 37$, $q = 43$, $e = 5$, $d = 605$, $y_p = 7$, $y_q = -6$, $m_p = 9$, $m_q = 8$, $m = 1341$.

Übung 9.7.13 Da e teilerfremd zu $(p-1)(q-1)$ ist, gilt für die Ordnung k der primen Restklasse $e + \mathbb{Z}(p-1)(q-1)$: $e^k \equiv 1 \bmod (p-1)(q-1)$. Daraus folgt $c^{e^{k-1}} \equiv m^{e^k} \equiv m \bmod n$. Solange k groß ist, stellt dies keine Bedrohung dar.

Übung 9.7.15 Ja, denn die Zahlen $(x_5 2^5 + x_4 2^4 + x_3 2^3 + x_2 2^2) \bmod 253$, $x_i \in \{0, 1\}$, $2 \leq i \leq 5$, sind paarweise verschieden.

Übung 9.7.17 Low-Exponent-Attack: Wenn eine Nachricht $m \in \{0, 1, \ldots, n-1\}$ mit dem Rabin-Verfahren unter Verwendung der teilerfremden Moduln n_1 und n_2 verschlüsselt wird, entstehen die Schlüsseltexte $c_i = m^2 \mod n_i$, $i = 1, 2$. Der Angreifer bestimmt eine Zahl $c \in \{0, \ldots, n_1 n_2 - 1\}$ mit $c \equiv c_i \bmod n_i$, $i = 1, 2$. Dann ist $c = m^2$, und m kann bestimmt werden, indem aus c die Quadratwurzel gezogen wird. Gegenmaßnahme: Randomisierung einiger Nachrichtenbits.

Multiplikativität: Wenn Bob die Schlüsseltexte $c_i = m_i^2 \bmod n$, $i = 1, 2$, kennt, dann kann er daraus den Schlüsseltext $c_1 c_2 \bmod n = (m_1 m_2)^2 \bmod n$ berechnen. Gegenmaßnahme: spezielle Struktur der Klartexte.

Übung 9.7.19 Wenn $(B_1 = g^{b_1}, C_1 = A^{b_1} m_1)$, $(B_2 = g^{b_2}, C_2 = A^{b_2} m_2)$ die Schlüsseltexte sind, dann ist auch $(B_1 B_2, C_1 C_2 = A^{b_1 + b_2} m_1 m_2)$ ein gültiger Schlüsseltext. Er verschlüsselt $m_1 m_2$. Man kann diese Attacke verhindern, wenn man nur Klartexte von spezieller Gestalt erlaubt.

Übung 9.7.21 Der Klartext ist $m = 37$.

Übung 10.6.1 Da $x^2 \geq n$ ist, ist $\lceil \sqrt{n} \rceil = 115$ der kleinstmögliche Wert für x. Für dieses x müssen wir prüfen, ob $z = n - x^2$ ein Quadrat ist. Wenn nicht, untersuchen wir $x+1$. Es ist $(x+1)^2 = x^2 + 2x + 1$. Daher können wir $(x+1)^2$ berechnen, indem wir zu x^2 den Wert $2x+1$ addieren. Wir finden schließlich, daß $13199 = 132^2 - 65^2 = (132 - 65)(132 + 65) = 67 * 197$ ist. Nicht jede zusammengesetzte natürliche Zahl ist Differenz von zwei Quadraten. Daher funktioniert das Verfahren nicht immer. Wenn es funktioniert, braucht es $O(\sqrt{n})$ Operationen in $\mathbb{Z}$.

Übung 10.6.3 Die Faktorisierung $n = 11617 * 11903$ findet man, weil $p-1 = 2^5 * 3 * 11^2$ und $q = 2 * 11 * 541$ ist. Man kann also $B = 121$ setzen.

Übung 10.6.5 Die Anzahl der Primzahlen $\leq B$ ist $O(B/\log B)$ nach Theorem 8.1.6. Jede der Primzahlpotenzen, deren Produkt k bildet, ist $\leq B$. Damit ist $k = O(B^{B/\log B}) = O(2^B)$. Die Exponentiation von a mit k mod n erfordert nach Theorem 3.12.2 $O(B)$ Multiplikationen mod n.

Übung 10.6.7 Es ist $m = 105$. Man erhält mit dem Siebintervall $-10, \ldots, 10$ und der Faktorbasis $\{-1, 2, 3, 5, 7, 11, 13\}$ die zerlegbaren Funktionswerte $f(-4) = -2 * 5 * 7 * 13$, $f(1) = 5^3$, $f(2) = 2 * 13^2$, $f(4) = 2 * 5 * 7 * 11$, $f(6) = 2 * 5 * 11^2$. Man erhält daraus die Kongruenz $(106 * 107 * 111)^2 \equiv (2 * 5^2 * 11 * 13)^2 \bmod n$. Also ist $x = 106 * 107 * 111$, $y = 2 * 5^2 * 11 * 13$ und damit $\gcd(x - y, n) = 41$.

Übung 11.9.1 Der DL ist $x = 323$.

Übung 11.9.3 Die kleinste Primitivwurzel mod 1117 ist 2. Der DL ist $x = 1212$.

Übung 11.9.5 Die kleinste Primitivwurzel mod 3167 ist 5 und es gilt $5^{1937} \equiv 15 \bmod 3167$.

Übung 11.9.7 Die kleinste Primitivwurzel mod $p = 2039$ ist $g = 7$. Es gilt $7^{1344} \equiv 2 \bmod p$, $7^{1278} \equiv 3 \bmod p$, $7^{664} \equiv 5 \bmod p$, $7^{861} \equiv 11 \bmod p$, $7^{995} \equiv 13 \bmod p$.

Übung 12.9.1 Sei n ein 1024-Bit Rabin-Modul (siehe Abschnitt 9.4). Die Funktion $\mathbb{Z}_n \to \mathbb{Z}_n$, $x \mapsto x^2 \bmod n$ ist eine Einwegfunktion, falls n nicht faktorisiert werden kann. Das folgt aus den Überlegungen in Abschnitt 9.4.5.

Übung 12.9.3 Der maximale Wert von $h(k)$ ist 9999. Daraus ergibt sich die maximale Länge der Bilder zu 14. Eine Kollision ist $h(1) = h(10947)$.

Übung 13.10.1 Es ist $n = 127 * 227$, $e = 5$, $d = 22781$, $s = 7003$.

Übung 13.10.3 Die Signatur ist eine Quadratwurzel mod n aus dem Hashwert des Dokuments. Die Hashfunktion muß aber so ausgelegt werden, daß ihre Werte nur Quadrate mod n sind. Die Sicherheits- und Effizienzüberlegungen entsprechen denen in Abschnitt 9.4.

Übung 13.10.5 Es ist $A^r r^s = A^q (q^{(p-3)/2})^{h(m)-qz}$. Weil $gq \equiv -1 \bmod p$ ist, gilt $q \equiv -g^{-1} \bmod p$. Außerdem ist $g^{(p-1)/2} \equiv -1 \bmod p$, weil g eine Primitivwurzel mod p ist. Daher ist $q^{(p-3)/2} \equiv (-g)^{(p-1)/2} g \equiv g \bmod p$. Insgesamt hat man also $A^r r^s \equiv A^q g^{h(m)} g^{-qz} \equiv A^q g^{h(m)} A^{-q} \equiv g^{h(m)} \bmod p$. Die Attacke funktioniert, weil g ein Teiler von $p-1$ ist und der DL z von A^q zur Basis g^q berechnet werden konnte. Man muß das also verhindern.

Übung 13.10.7 Es ist $r = 799$, $k^{-1} = 1979$, $s = 1339$.

Übung 13.10.9 Es ist $q = 43$. Der Erzeuger der Untergruppe der Ordnung q ist $g = 1984$. Weiter ist $A = 834$, $r = 4$, $k^{-1} = 31$ und $s = 23$.

Übung 13.10.11 Sie lautet $g^s = A^r r^{h(x)}$.

Übung 14.4.1 Wir müssen dazu ein irreduzibles Polynom vom Grad 3 über GF(3) konstruieren. Das Polynom $x^2 + 1$ ist irreduzibel über GF(3), weil es keine Nullstelle hat. Der Restklassenring mod $f(X) = X^2 + 1$ ist also GF(9). Bezeichne mit α die Restklasse von X mod $f(X)$. Dann gilt also $\alpha^2 + 1 = 0$. Die Elemente von GF(9) sind 0, 1, 2, α, 2α, $1+\alpha$, $1+2\alpha$, $2+\alpha$, $2+2\alpha$. Die Additionstabelle ergibt sich unter Verwendung der Rechenregeln in $\mathbb{Z}/3\mathbb{Z}$. Für die Multiplikationstabelle braucht man zusätzlich die Regel $\alpha^2 = -1$.

Übung 14.4.3 Die Punkte sind $\mathcal{O}$, $(0,1)$, $(0,6)$, $(2,2)$, $(2,5)$. Die Gruppe hat also die Ordnung 5 und ist damit zyklisch. Jeder Punkt $\neq \mathcal{O}$ ist ein Erzeuger.

Übung 15.5.1 Alice wählt zufällig und gleichverteilt einen Exponenten $b \in \{0, 1, \ldots, p-2\}$ und berechnet $B = g^b \bmod p$. Sie schickt B an Bob. Bob wählt $e \in \{0, 1\}$ zufällig und gleichverteilt und schickt e an Alice. Alice schickt $y = (b + ea) \bmod (p-1)$ an Bob. Bob verifiziert $g^y \equiv A^e B \bmod p$. Das Protokoll ist vollständig, weil jeder, der den geheimen Schlüssel von Alice kennt, sich erfolgreich identifizieren kann. Wenn Alice das richtige y für $e = 0$ und für $e = 1$ zurückgeben kann, kennt sie den DL a. Daher kann sie nur mit Wahrscheinlichkeit $1/2$ betrügen. Das Protokoll ist also korrekt. Das Protokoll kann von Bob simuliert werden. Er wählt gleichverteilt zufällig $y \in \{0, 1, \ldots, p-2\}$, $e \in \{0, 1\}$ und setzt $B = g^y A^{-e} \bmod p$. Damit funktioniert das Protokoll und die Wahrscheinlichkeitsverteilungen sind dieselben wie im Originalprotokoll.

Übung 15.5.3 Ein Betrüger muß Zahlen x und y liefern, die das Protokoll erfüllen. Wenn er x mitteilt, kennt er das zufällige $e = (e_1, \ldots, e_k)$ nicht. Wäre er in der Lage, nach Kenntnis von e noch ein korrektes y zu produzieren, könnte er Quadratwurzeln mod n berechnen. Das kann er aber nicht. Also kann er x nur so so wählen, daß er die richtige Antwort y für genau einen Vektor $e \in \{0, 1\}^k$ geben kann. Er kann sich also nur mit Wahrscheinlichkeit 2^{-k} richtig identifizieren.

Übung 15.5.5 Der Signierer wählt r zufällig, berechnet $x = r^2 \bmod n$, $(e_1, \ldots, e_k) = h(x \circ m)$ und $y = r \prod_{i=1}^{k} s_i^{e_i}$. Die Signatur ist (x, y).

Übung 16.3.2 Es gilt $a(X) = sX + 3$, $y_1 = 7$, $y_2 = 9$, $y_3 = 0$, $y_4 = 2$.

Literaturverzeichnis

1. Advanced Encryption Standard. `http://csrc.nist.gov/encryption/aes/`.
2. M. Agraval, N. Kayal, and N. Saxena. Primes is in P. `http://www.cse.iitk.ac.in/news/primality.html`.
3. A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachusetts, 1974.
4. E. Bach and J. Shallit. *Algorithmic Number Theory*. MIT Press, Cambridge, Massachusetts and London, England, 1996.
5. F. Bauer. *Entzifferte Geheimnisse*. Springer-Verlag, Berlin, 1995.
6. F. Bauer. *Decrypted Secrets*. Springer-Verlag, Berlin, 2000.
7. M. Bellare and S. Goldwasser. Lecture notes on cryptography. `www.cse.ucsd.edu/users/mihir`.
8. M. Bellare and P. Rogaway. The exact security of digital signatures: How to sign with RSA and Rabin. In *Advances in Cryptology - EUROCRYPT '96*, pages 399–416. Springer-Verlag, 1996.
9. M. Bellare and P. Rogaway. Optimal asymmetric encryption - how to encrypt with RSA. In *Advances in Cryptology - EUROCRYPT '94*, pages 92–111. Springer-Verlag, 1996.
10. A. Beutelspacher, J. Schwenk, and K.-D. Wolfenstetter. *Moderne Verfahren der Kryptographie*. Vieweg, 1998.
11. E. Biham and A. Shamir. *Differential Cryptanalysis of the Data Encryption Standard*. Springer-Verlag, New York, 1993.
12. I.F. Blake, G. Seroussi, and N.P. Smart. *Elliptic Curves in Cryptography*. Cambridge University Press, Cambridge, England, 1999.
13. D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS#1. In *Advances in Cryptology - CRYPTO '98*, pages 1–12, 1998.
14. D. Boneh. The decision diffie-hellman problem. In *ANTS III*, volume 1423 of *Lecture Notes in Computer Science*, pages 48–63, 1998.
15. D. Boneh and G. Durfee. Cryptanalysis of RSA with private keys d less than $N^{0.292}$. *IEE Transactions on Information Theory*, 46(4):1339–1349, 2000.
16. J. Buchmann. Faktorisierung großer Zahlen. *Spektrum der Wissenschaften*, 9:80–88, 1996.
17. J. Buchmann and S. Paulus. A one way function based on ideal arithmetic in number fields. In B. Kaliski, editor, *Advances in Cryptology - CRYPTO '97*, volume 1294 of *Lecture Notes in Computer Science*, pages 385–394, Berlin, 1997. Springer-Verlag.
18. J. Buchmann and H. C. Williams. Quadratic fields and cryptography. In J.H. Loxton, editor, *Number Theory and Cryptography*, volume 154 of *London Mathematical Society Lecture Note Series*, pages 9–25. Cambridge University Press, Cambridge, England, 1990.

19. Johannes Buchmann, Carlos Coronado, Erik Dahmen, Martin Döring, and Elena Klintsevich. CMSS — an improved Merkle signature scheme. In *Progress in Cryptology - INDOCRYPT 2006*, volume 4329 of *Lecture Notes in Computer Science*, pages 349–363. Springer-Verlag, 2006.
20. Johannes Buchmann, Erik Dahmen, Elena Klintsevich, Katsuyuki Okeya, and Camille Vuillaume. Merkle signatures with virtually unlimited signature capacity. In *Applied Cryptography and Network Security - ACNS 2007*, volume 4521 of *Lecture Notes in Computer Science*, pages 31–45. Springer-Verlag, 2007.
21. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Massachudetts, 1990.
22. R. Cramer and V. Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attacks. In *Advances in Cryptology - CRYPTO 99*, pages 13–25, 1998.
23. N.G. de Bruijn. On the number of integers $\leq x$ and free of prime factors $> y$. *Indag. Math.*, 38:239–247, 1966.
24. Discrete Logarithm Records. `http://www.medicis.polytechnique.fr/~lercier/english/dlog.html`.
25. H. Dobbertin. The status of MD5 after a recent attack. *CryptoBytes*, 2(2):1–6, 1996.
26. H. Dobbertin. Cryptanalysis of MD4. *Journal of Cryptology*, 11(4):253–271, 1998.
27. Chris Dods, Nigel Smart, and Martijn Stam. Hash based digital signature schemes. In *Cryptography and Coding*, volume 3796 of *Lecture Notes in Computer Science*, pages 96–115. Springer-Verlag, 2005.
28. D. Dolev, C. Dwork, and M. Naor. Non-malleable cryptography. In *23rd Annual ACM Symposium on Theory of Computing (STOC)*, pages 542–552, 1991.
29. Factoring records. `www.crypto-world.com`.
30. A. Fiat and M. Naor. Rigorous time/space trade offs for inverting functions. In *23rd ACM Symp. on Theory of Computing (STOC)*, pages 534–541. ACM Press, 1991.
31. FIPS 186-2, Digital Signature Standard (DSS). Federal Information Processing Standards Publication 186-2, U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, Virginia, 2000.
32. FlexiPKI. `http://www.informatik.tu-darmstadt.de/TI/Forschung/FlexiPKI`.
33. O. Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudorandomness*. Springer-Verlag, New York, 1999.
34. S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28:270 – 299, 1984.
35. D. M. Gordon. A survey of fast exponentiation methods. *Journal of Algorithms*, 27:129–146, 1998.
36. M. Hellman. A cryptanalytic time-memory trade-off. *IEEE Trans. on Information Theory*, 26(4):401–406, 1980.
37. P. Horster. *Kryptologie*. Bibliographisches Institut, 1987.
38. Internet users. `http://www.nua.ie/surveys/how_many_online/`.
39. ISO/IEC 9796. Information technology - Security techniques - Digital signature scheme giving message recovery. International Organization for Standardization, Geneva, Switzerland, 1991.
40. D. Kahn. *The codebreakers*. Macmillan Publishing Company, 1967.
41. Lars R. Knudsen. Contemporary block ciphers. In Ivan Damgard, editor, *Lectures on Data Security*, volume 1561 of *LNCS*, pages 105–126. Springer-Verlag, New York, 1999.

42. D.E. Knuth. *The art of computer programming. Volume 2: Seminumerical algorithms*. Addison-Wesley, Reading, Massachusetts, 1981.
43. N. Koblitz. *A Course in Number Theory and Cryptography*. Springer-Verlag, 1994.
44. Leslie Lamport. Constructing digital signatures from a one way function. Technical Report SRI-CSL-98, SRI International Computer Science Laboratory, 1979.
45. A.K. Lenstra and E.R.Verheul. Selecting cryptographic key sizes, October 1999.
46. A.K. Lenstra and H.W. Lenstra, Jr. Algorithms in Number Theory. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A, Algorithms and Complexity*, chapter 12. Elsevier, Amsterdam, 1990.
47. A.K. Lenstra and H.W. Lenstra Jr. Algorithms in number theory. In J. van Leeuwen, editor, *Handbook of theoretical computer science. Volume A. Algorithms and Complexity*, chapter 12, pages 673–715. Elsevier, 1990.
48. A.K. Lenstra and H.W. Lenstra Jr., editors. *The Development of the Number Field Sieve*. Lecture Notes in Math. Springer-Verlag, Berlin, 1993.
49. H.W. Lenstra, Jr. and C. Pomerance. A rigorous time bound for factoring integers. *Journal of the AMS*, 5:483–516, 1992.
50. LiDIA. `www.informatik.tu-darmstadt.de/TI/Welcome-Software.html`.
51. A. Menezes. *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, Dordrecht, 1993.
52. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, Florida, 1997.
53. Ralph C. Merkle. A certified digital signature. In *CRYPTO '89: Proceedings on Advances in cryptology*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer-Verlag, 1989.
54. Mersenn1039. `http://www.loria.fr/~zimmerma/records/21039-`.
55. K. Meyberg. *Algebra Teil 1*. Carl Hanser Verlag, 1980.
56. K. Meyberg. *Algebra Teil 2*. Carl Hanser Verlag, 1980.
57. B. Möller. Improved techniques for fast exponentiation. In *Proceedings of ICISC 2002*. Springer-Verlag, 2003.
58. M.Naor and M. Yung. Public-key cryptosystems provably secure against chosen ciphertext attacks. In *22nd Annual ACM Symposium on Theory of Computing (STOC)*, 1990.
59. E. Oeljeklaus and R. Remmert. *Lineare Algebra I*. Springer-Verlag, Berlin, 1974.
60. PKCS#1. `www.rsasecurity.com/rsalabs/pkcs/pkcs-1/index.html`.
61. D. Pointcheval and J. Stern. Security arguments for digital signatures and blind signatures. *Journal of Cryptology*, 13:361–396, 2000.
62. C. Rackoff and D.R. Simon. Non-interactive zero knowledge proof of knowledge and chosen ciphertext attack. In *Advances in Cryptology - CRYPTO '91*, volume 576 of *Lecture Notes in Computer Science*, pages 433–444. Springer-Verlag, 1991.
63. RFC 1750. Randomness requirements security. Internet Request for Comments 1750.
64. H. Riesel. *Prime Numbers and Computer Methods for Factorization*. Birkhäuser, Boston, 1994.
65. M. Rosing. *Implementing elliptic curve cryptography*. Manning, 1999.
66. J. Rosser and L. Schoenfeld. Approximate formulas for some functions of prime numbers. *Illinois Journal of Mathematics*, 6:64–94, 1962.
67. Rsa200. `http://www.loria.fr/~zimmerma/records/rsa200`.
68. R.A. Rueppel. *Analysis and Design of Stream Ciphers*. Springer-Verlag, Berlin, 1986.

69. O. Schirokauer, D. Weber, and T. Denny. Discrete logarithms: the effectiveness of the index calculus method. In H. Cohen, editor, *ANTS II*, volume 1122 of *Lecture Notes in Computer Science*, Berlin, 1996. Springer-Verlag.
70. B. Schneier. *Applied cryptography.* Wiley, New York, second edition, 1996.
71. Secure Hash Standard. `http://csrc.nist.gov/`.
72. A. Shamir. How to share a secret. *Communications of the ACM*, 22:612–613, 1979.
73. C.E. Shannon. Communication theory of secrecy systems. *Bell Sys. Tech. Jour.*, 28:656–715, 1949.
74. P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26:1484–1509, 1997.
75. Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *IEEE Symposium on Foundations of Computer Science*, pages 124–134, 1994.
76. V. Shoup. Why chosen ciphertext security matters. IBM Research Report RZ 3076, IBM Research Division, 1998.
77. V. Shoup. OAEP reconsidered. In *Advances in Cryptology - CRYPTO 2001*, pages 239–259. Springer-Verlag, 2001.
78. D. Stinson. *Cryptography.* CRC Press, Boca Raton, Florida, 1995.
79. D. Stinson. *Cryptography, Theory and Practice.* CRC Press, Boca Raton, Florida, second edition edition, 2002.
80. Michael Szydlo. Merkle tree traversal in log space and time. In *Advances in Cryptology - EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 541–554, 2004.
81. Lieven M. K. Vandersypen, Matthias Steffen, Gregory Breyta, Costantino S. Yannoni, Mark H. Sherwood, and Isaac L. Chuang. Experimental realization of shor's quantum factoring algorithm using nuclear magnetic resonance. *Nature*, 414:883, 2001.
82. M.J. Wiener. Efficient DES key search - an update. *CryptoBytes*, 3(2):6–8, 1998.

Sachverzeichnis