Kurz-Tips: Die RSA-Chiffre

Autor: mm_freak ‹never@drwxr-xr-x.org›

Datum: 2004-08-10 (letzte Änderung: 2004-12-20)

Kurz-Tips: Die RSA-Chiffre

Öffentliche Vorführung ist gestattet.

Inhalt

Allgemein

Zahlentheorie I

Modulare Arithmetik

Zahlentheorie II

Das RSA-Verfahren

Zahlentheorie III

Unkryptologische Attacken auf asymmetrische Verfahren

Programmierung

Ketchup und Senf für die Welt

Wer sich bereits weitgehend mit Mengenlehre und Primzahl- und Teilbarkeitstheorie auskennt, kann diese beiden Kapitel überspringen. Ich empfehle dennoch, sie zu lesen. Dieses Dokument ist linear organisiert. Es ist also darauf ausgelegt, Kapitel für Kapitel und nicht kreuz und quer gelesen zu werden. Daher finden sich in diesem Text auch keine Vorwärtsreferenzen (Links, die im Dokument weiter nach unten führen).

Vorwort

In diesem Text wird die Funktionsweise eines der abstraktesten mathematischen Verfahren dargestellt. Es wird erklärt, wie und, vor allem, *warum* es funktioniert. In gewisser Hinsicht dient dieses Dokument auch als Gesamtratgeber, da viele andere Dokumentationen, die RSA unter die Lupe nehmen, einfach nicht alle wichtigen Details ansprechen. In diesem Text wird alles von den Theorien über die *modulare Arithmetik* bis hin zum RSA-Verfahren selbst behandelt. Auch die Gefahren und Tücken, die sich hinter RSA verstecken, werden aufgezeigt und erklärt.

Hier geht es vor allem um die Verständlichkeit. Auch Nichtmathematiker dürften mit diesem Text etwas anfangen können. Andere Texte über RSA sind oft allerhöchstens eine Seite lang. Das ist durchaus kein Fehler, denn RSA lässt sich innerhalb von wenigen Sätzen *definieren*. Dies setzt vom Leser aber voraus, dass er das mathematische Vorwissen hat und die ganzen Ideen und Theorien, die hinter RSA stecken, bereits kennt. Mit dieser Voraussetzung ist der Leser in diesem Text nicht konfrontiert. Außerdem wurde diese Dokumentation nicht wie ein Fachbuch geschrieben, das nur aus kryptischen mathematischen Aussagen besteht, die man erst entschlüsseln und danach auch noch verstehen muss. Der Leser wird in alle Bestandteile von RSA ausführlich eingeführt. Vom Leser wird lediglich die Kenntnis der elementaren Bestandteile der Mathematik vorausgesetzt. Genau genommen braucht der Leser folgendes mathematisches Vorwissen: simple Arithmetik (Addition und Multiplikation und die damit verbundenen Regeln), Potenzrechnung, Funktionen, Gleichungen und Kongruenzen (wobei dennoch eine kurze Einführung in Kongruenzen geboten wird). Auch werden Themen behandelt, die leider sehr oft unterschätzt werden, z.B. die Sicherheit bei der Generierung von Zufallszahlen.

Dieser Text füllt eine Lücke, die die meisten Dokumentationen zu RSA offen lassen. Es wird sehr viel mit Beispielen gearbeitet. Ich habe versucht, jeden Satz zu beweisen. Dabei muss angemerkt werden, dass ich die meisten Beweise selbst erarbeitet habe und dabei könnten mir Fehler unterlaufen sein. Falls ein solcher gefunden wird, bin ich für eine Rückmeldung sehr dankbar.

Da nicht jeder an den Beweisen interessiert ist, sind diese speziell hervorgehoben. Sie befinden sich in Kästen mit gepunkteter gelber Umrahmung. Wer die Beweise nicht unbedingt benötigt, kann diese Kästen ohne die Befürchtung, Informationen zu verlieren, überspingen. Beispiel:

TODO

Mathematische Formeln und Terme werden hier in der bc-Schreibweise notiert, damit sie mit jedem Browser einfach lesbar sind. Funktionsnamen werden benutzerfreundlicher gestaltet. So heißt es z.B. sin() statt s(). Ganzzahlanteil und Nachkommaanteil werden nicht durch Komma, sondern durch einen Punkt getrennt. So heißt es 13.4 statt 13,4. Multiplikationen werden nicht wie gewöhnlich abgekürzt. Es heißt also a * b statt ab. Dennoch arbeiten wir meist nur mit

einfachen Variablennamen, die aus einem einzigen Buchstaben bestehen. Die Exponentiation wird durch das Zeichen ^ beschrieben. Der modulo-Operator (Divisionsrest) lautet mod. Da in diesem Text fast nur mit Ganzzahlen gearbeitet wird, wird nur an den Stellen, an denen Fließkommazahlen zum Einsatz kommen, speziell darauf hingewiesen (z.B. wenn Zahlen Elemente von der Menge R, also reelle Zahlen sind). Das heißt auch, dass der Operator / eine Ganzzahldivision darstellt (sofern nicht anders angegeben), das Ergebnis ist also der Quotient ohne den Nachkommaanteil. Generell wird eine Kongruenz durch ein Gleichheitszeichen mit drei Strichen dargestellt. Da dieses nicht zur Verfügung steht, wird sie mit einem normalen Gleichheitszeichen ausgedrückt. Kommentare innerhalb von mathematischen Ausdrücken werden durch doppelte Schrägstriche eingeleitet, ähnlich wie bei *C*++. Mengen werden als normale Buchstaben angegeben. Subskripte werden durch eckige Klammern ausgedrückt. Beispiele:

```
f(x) = p + a * sin(f * x)  // Normale Sinuswellenfunktion
x^y                        // Exponentiation ("x hoch y")
e = 2.7182818284590452354  // Gleichung
f(x) = e^x                 // Natürliche Exponentialfunktion
51 * x = 0 (mod 51)        // Modulare Kongruenz
17 mod 5                   // Divisionsrest von 17 / 5
13/4 + 1                   // Ergebnis: 4! Nicht 4.25
Z[17]                      // Die Menge Z mit dem Subskript 17
```

In diesem Text finden sich anwendbare Informationen für die Internetgemeinde. Sie wurden vom Autor mit größter Sorgfalt erarbeitet und sehr intensiv auf Mängel überprüft. Jedoch lassen sich Fehler nicht ausschließen. **Aus diesem Grund erhebt der Autor keine Gewähr für Richtigkeit oder Vollständigkeit. Er haftet nicht für Schäden und Folgeschäden, die durch diesen Text, oder die Verwendung dessen, entstehen. Kurz: Die Verwendung dieses Textes geschieht auf eigene Gefahr!**

Was ist RSA?

RSA ist ein Verschlüsselungsverfahren, das nach den Nachnamen der Mathematiker *Ronald Rivest*, *Adi Shamir* und *Leonard Adleman* benannt ist. Obwohl die Sicherheit dieses Verfahrens nie bestätigt wurde, ist sich jeder sicher, dass es nicht in nächster Zeit geknackt werden kann. Die Sicherheit dieses Verfahrens basiert auf der Schwierigkeit, große Zahlen mit wenig Primfaktoren zu faktorisieren. Gelingt dies für einen RSA-Schlüssel, so

ist dieser *geknackt* und damit wertlos. Doch es verbergen sich noch weitere Tücken, die später behandelt werden.

RSA ist das erste praktisch angewandte Beispiel einer sogenannten *Falltürfunktion*. Jeder kann durch eine Falltür fallen, doch nur diejenigen, die den geheimen Ausgang kennen, kommen auch wieder heraus. Die erste Idee für solch eine Falltürfunktion kam vom Studenten *Ralph Merkle*, der mit seiner Idee *Merkles Rätsel* ein Chiffreverfahren analog zur Falltürfunktion publizierte und damit die Existenz *asymmetrischer* Chiffren ein Stück glaubhafter machte. Sein Verfahren wurde nie praktisch angewandt, lieferte aber eine Basis für eine völlig neuartige Chiffre, die das *Schlüsseltauschproblem* lösen sollte. Kurz darauf wurde von *Whitfield Diffie* und *Martin Hellman* das *Diffie-Hellman*-Protokoll bekannt gegeben, das es ermöglicht, einen geheimen Schlüssel über eine unsichere Leitung zu vereinbaren. Dieses Protokoll lieferte die Grundidee für das RSA-Verfahren. Da das Protokoll sehr simpel ist, wird es später kurz vorgestellt. Der aufmerksame Leser dürfte eine gewisse Ähnlichkeit mit dem RSA-Verfahren feststellen.

Hier ist anzumerken, dass das RSA-Verfahren oft fälschlicherweise als *RSA-Algorithmus* bezeichnet wird. Diese Bezeichnung trifft nicht zu und ist damit nicht korrekt. Die darunterliegenden einzelnen Rechenschritte lassen sich als Algorithmen implementieren, doch RSA selbst ist nur ein mathematischer Satz; daher sollten die Bezeichnungen *RSA-Verfahren* oder *RSA-Chiffre* vorgezogen werden. Der Satz wird am Ende des Kapitels *Eulersche phi-Funktion - Eulers Theorie* aufgezeigt und im Kapitel *Das RSA-Verfahren* dann angewandt.

Symmetrisch? Asymmetrisch?

Vor 1976 gab es nur die sogenannten *symmetrischen* Chiffren. Diese arbeiten mit einem einzigen Schlüssel für Ver- und Entschlüsselung. Lautet die Chiffrierfunktion E(m, k), wobei m die Nachricht und k der Schlüssel ist, so lautet die Dechiffrierfunktion D(c, k) mit c als Chiffretext (Rückgabewert von E). Die Funktion D wirkt nur dann invers zu

E, wenn für beide Schritte der gleiche Schlüssel k oder ein dazu kongruenter Schlüssel angegeben wird. In dem Fall kann man sagen: m = D(E(m, k), k). Diese Form der Verschlüsselung wird auch heute sehr exzessiv genutzt, da sie, verglichen mit den asymmetrischen Chiffren, durch einen Computer sehr schnell und effizient durchfürbar ist.

Die asymmetrische Chiffre teilt den Schlüssel k in zwei gegenseitig inverse Schlüssel e und d auf. Wird mit dem Schlüssel e chiffriert, kann nur mit dem Schlüssel d wieder dechiffriert werden. Wird hingegen mit dem Schlüssel d chiffriert, so kann nur mit e dechiffriert werden. Einer dieser Schlüssel wird als *öffentlicher* und der andere als *geheimer* Schlüssel festgelegt. Jeder kann mit dem öffentlichen Schlüssel verschlüsseln, jedoch kann nur der Eigentümer des dazugehörigen geheimen Schlüssels die Verschlüsselung umkehren. Andersrum kann eine Nachricht mit dem geheimen Schlüssel verschlüsselt werden. Jeder kann sie mit dem öffentlichen Schlüssel dechiffrieren und lesen, aber nur der Eigentümer des geheimen Schlüssels kann solche Nachrichten erstellen (Signaturprinzip). Lautet die Chiffrierfunktion E(m, e) mit m als Nachricht und e als öffentlichen Schlüssel, so lautet die Dechiffrierfunktion D(c, d), mit c als Chiffretext und d als geheimen Schlüssel. Die Funktion D wirkt nur dann invers zu E, wenn beim Dechiffrieren der zum öffentlichen Schlüssel e gehörende geheime Schlüssel d oder ein Schlüssel kongruent zu d angegeben wird, oder andersrum. Man kann also sagen: m = D(E(m, e), d) und m = D(E(m, d), e). Diese Form der Verschlüsselung wird meist nur zur Schlüsselvereinbarung und zur Signatur genutzt, da sie wesentlich langsamer als die symmetrische Chiffre ist. Sie wird mit den symmetrischen Chiffren kombiniert.

Das Schlüsseltauschproblem

Die klassischen *symmetrischen* Chiffren hatten alle ein Problem gemeinsam: das *Schlüsseltauschproblem*. Die Sicherheit eines Verfahrens ist wertlos, solange man keine Möglichkeit hat, mit dem anderen Endpunkt der Kommunikation einen geheimen Schlüssel zu vereinbaren. Man stelle sich folgendes Szenario vor: Alice möchte eine geheime Nachricht an Bob senden. Die einzige Kommunikationsmöglichkeit zwischen Alice und Bob ist

jedoch eine lange Kupferleitung, die von jedem angezapft werden könnte. Alice und Bob kennen sich nicht und haben noch nie zuvor einen geheimen Schlüssel vereinbart. Genau hier wird das Schlüsseltauschproblem deutlich. Wie kann Alice auf sicherem Wege mit Bob einen Schlüssel vereinbaren?

1976 wurde die erste elegante Lösung dieses Problems vorgestellt: das *Diffie-Hellman*-Protokoll. Eine wichtige Einschränkung dieses Protokolls ist, dass es eben nur ein Protokoll zur Schlüsselvereinbarung ist und interaktiver Informationsaustausch notwendig ist. Mit diesem Protokoll sind weder Signaturen, noch Authentifizierung möglich. 1977 wurde dann die erste echte Chiffre bekannt gegeben, die das Schlüsseltauschproblem *fast* (siehe unten das Man in the Middle - Problem) gegenstandlos machte: RSA. Mit dieser kann man Nachrichten nun auch signieren und Benutzer authentifizieren.

Mengenlehre

Nun möchten wir langsam zum mathematischen Teil übergehen. Hier folgt nun eine kleine Einführung in die Mengenlehre. Wer sich mit dieser bereits auskennt, kann dieses Kapitel überspringen. Ich empfehle dennoch, es zu lesen. Man lernt nie aus.

In der Mathematik arbeiten wir mit einer Einheit, die wir schlichtweg *Zahl* nennen. Es gibt nun verschiedene Zahlen mit verschiedenen Eigenschaften. Es gibt *ganze Zahlen*, also Zahlen ohne Nachkommaanteil, es gibt *rationale Zahlen*, also Zahlen, die sich als *Bruch* (engl. ratio) mit ganzen Zahlen als Zähler und Nenner darstellen lassen und *irrationale Zahlen*, also Zahlen, die sich nicht als Bruch darstellen lassen. Es gibt noch weitere Arten von Zahlen wie die *komplexen Zahlen*. Die Zahl 25 ist beispielsweise eine ganze Zahl. Wir sagen, sie hat keinen Nachkommaanteil, was allerdings nicht ganz der Wahrheit entspricht. Nach dem Komma folgen unendlich viele Nullen. Diese Zahl könnte man also auch als 25.00000 notieren. Die 25 ist gleichzeitig auch eine rationale Zahl, denn sie lässt sich als Bruch 25/1 darstellen. Noch ein Beispiel für eine rationale Zahl wäre 1.25. Diese Zahl ist keine ganze

Zahl, lässt sich jedoch als Bruch darstellen: 5/4. Ein weiteres Beispiel wäre die Zahl 1/3. Sie lässt sich nur mit einer Periode eindeutig als Dezimalzahl darstellen. Beispiele für irrationale Zahlen sind die Quadratwurzel aus zwei, die Eulersche Zahl e und die Kreiszahl pi (dargestellt als griechischer Buchstabe *PI*). Diese Zahlen haben unendlich viele Nachkommastellen ohne eine Periode und lassen sich auch nicht als Bruch darstellen.

Diese Zahlen werden benannten Mengen untergeordnet. Beispielsweise gibt es die Menge der natürlichen Zahlen. Diese Menge enthält alle Ganzzahlen von 1 aufwärts. Die Zahlen 1, 2, 3, 4 usw. gehören zu dieser Menge. Man sagt, sie sind *Elemente* der Menge der natürlichen Zahlen. Auch werden die Mengen selbst abgekürzt. Die Menge der natürlichen Zahlen heißt N. Man sieht sofort, dass diese Menge unendlich viele Elemente besitzt. Die untere Grenze ist 1, es gibt also keine Zahl kleiner als 1. Eine obere Grenze ist jedoch nicht gesetzt, also gibt es von 1 aufwärts unendlich viele Elemente. Die Menge N mit dem Subskript 0, also die Menge N[0] enthält zusätzlich noch die 0. N ist eine Untergruppe von N[0], denn alle Elemente von N sind gleichzeitig Elemente von N[0]. Eine weitere Menge mit ganzen Zahlen ist die Menge Z. In dieser Menge hat jedes Element ein eindeutiges additives inverses Element. In diesem Fall sind das die negativen Zahlen. Das additive Inverse zu 3 ist -3. In jeder Menge ist das neutrale Element der Addition invers zu sich selbst. In N[0] und Z ist 0 das neutrale Element der Addition (Addition mit 0 hat keine Wirkung). N hat kein solches Element. Z ist die erste Menge, die beidseitig unendlich ist. Sie geht unendlich in die positive und in die negative Richtung. N und N[0] sind Untergruppen von Z.

Die Menge der rationalen Zahlen wird als Q bezeichnet. Alle Zahlen, die sich als Bruch darstellen lassen, sind Elemente von Q. Es wird ersichtlich, dass Z eine Untergruppe von Q ist, denn alle Elemente von Z sind gleichzeitig auch Elemente von Q. Andersherum ist dies nicht der Fall. Auch zeigt sich, dass N und N[0] auch Untergruppen von Q sind, da sie Untergruppen von Z sind und Z eine Untergruppe von Q ist. Auch diese Menge ist unendlich. Interessant ist, dass zwischen jeden zwei Elementen unendlich viele weitere Elemente liegen. Zwischen 3.5261 und 3.5262 liegen unendlich viele Elemente wie z.B. 3.52617.

Wenn wir uns nun den Graphen für die Funktion f(x) = x, wobei x eine rationale Zahl sein muss, ansehen, sehen wir eine diagonale Gerade. Überraschenderweise ist diese Gerade alles andere als dicht. Zwischen zwei Punkten existieren unendlich viele weitere Punkte. Aber auch existieren unendlich viele Löcher, denn zwischen jeden zwei Elementen von Q gibt es unendlich viele Zahlen, die sich nicht als Bruch darstellen lassen. Diese Funktion ist beispielsweise für die Quadratwurzel aus zwei nicht definiert. Also ist an dieser Stelle ein Loch in der Geraden. Aus diesem Grund führen wir eine weitere Menge ein: die Menge der *reellen Zahlen*. Nicht, dass alle anderen Zahlen unreell sind. Dies ist schlicht der Name für eine Menge, in der die rationalen Zahlen und die irrationalen Zahlen zusammengefasst werden. Die Menge heißt R. Jetzt erst ist der Werte- und Definitionsbereich von f(x) = x dicht.

Es gibt noch weitere Mengen wie die Menge der komplexen Zahlen C, doch dies übersteigt den Rahmen dieser Dokumentation. Die bisher vorgestellten Mengen waren unendlich. Entweder ließen sie sich von einem Punkt unendlich auf eine oder beide Seiten fortführen, oder es existieren allein zwischen zwei noch so nah beieinander liegenden Punkten unendlich viele weitere Punkte. In diesem Text allerdings arbeiten wir mit einer speziellen Menge, der *endlichen Menge*. Unsere Menge besteht nur aus Ganzzahlen, daher machen wir uns Z zum Vorbild und nennen unsere Menge Z[n], wobei n das *Modul* ist. Das Modul ist die Anzahl der Zahlen in dieser Form der endlichen Menge. Wir bezeichnen sie als Modul, weil wir die *modulare Arithmetik* in dieser Menge anwenden können. Es gibt zwar keine negativen Zahlen, wir haben uns aber dennoch für den Namen Z[n] statt N[n] entschieden, denn in der Menge Z geht es nicht um die Einführung der negativen Zahlen, sondern um die Einführung eines additiven Inversen für jedes Element. In N[0] ist 0 das einzige Element mit einem additiven Inversen. Das Element ist invers zu sich selbst. In N gibt es überhaupt keine additiven inversen Elemente. Additionen sind nicht per Addition umkehrbar. In Z[n] gibt es wie in Z auch zu jedem Element ein eindeutiges additives inverses Element, das allerdings nicht negativ ist. Auch in Z[n] ist die 0 invers zu sich selbst. Die endlichen Mengen werden im Kapitel über die *modulare Arithmetik* eingeführt.

Mengen haben bestimmte Eigenschaften bezüglich den erlaubten Operationen. Beispielsweise hat in N kein einziges Element ein additives inverses Element. In Z lässt sich die Multiplikation nur umkehren, wenn wir mit 1 (dem neutralen Element der Multiplikation) multiplizieren. Eine Menge wird als *additive Gruppe* bezeichnet, sofern sie folgende Voraussetzungen erfüllt: es existiert für jedes Element ein eindeutiges

additives inverses Element und es existiert genau ein neutrales Element der Addition, das invers zu sich selbst ist. Für die Addition gelten das Kommutativgesetz und das Assoziativgesetz. Eine Menge wird als *multiplikative Gruppe* bezeichnet, sofern sie folgende Voraussetzungen erfüllt: es existiert für jedes Element, außer dem neutralen Element der Addition, ein eindeutiges multiplikatives inverses Element und es existiert genau ein neutrales Element der Multiplikation. Das neutrale Element der Multiplikation ist invers zu sich selbst und eine Multiplikation mit dem neutralen Element der Addition ist nicht umkehrbar. Für die Multiplikation gelten das Kommutativgesetz, das Assoziativgesetz und das Distributivgesetz. Hier wird sichtbar, dass N und N[0] weder eine additive, noch eine multiplikative Gruppe sind. Z ist eine additive Gruppe, denn jedes Element hat ein eindeutiges inverses Element (z.B. ist -5 invers zu 5 und andersherum), es existiert ein neutrales Element der Addition: 0. Z ist jedoch keine multiplikative Gruppe. Q und R sind additive und multiplikative Gruppen. Z[n] ist eine additive Gruppe und unter bestimmten Voraussetzungen auch eine multiplikative Gruppe. Endliche Mengen, die additive *und* multiplikative Gruppen sind, werden als *endliche Körper* bezeichnet. Mehr dazu später.

Eine *Kongruenz* drückt aus, dass zwei Zahlen in einer Menge die gleichen Zahlen sind. Man könnte sie also problemlos untereinander austauschen, ohne Einfluss. Die Kongruenz wird normalerweise mit einem Gleichheitszeichen mit drei Strichen ausgedrückt, aber dieses steht nicht zur Verfügung. Somit arbeiten wir mit den normalen Gleichheitszeichen, was nicht unbedingt korrekt ist. Die Aussage 3 = 7 ist falsch. Soll sie jedoch eine Kongruenz darstellen, hängt es davon ab, in welcher Menge wir arbeiten. In Z[4] wäre diese Aussage nämlich richtig. Wir dürfen sie dennoch nicht als Gleichung darstellen, denn 3 und 7 sind nicht die gleichen Zahlen. Man sagt, 3 ist kongruent zu 7 in der Menge Z[4]. Kongruenzen in der endlichen Menge Z[n] werden folgendermaßen ausgedrückt:

```
a = b (mod n)
```

Diese Kongruenz sagt aus, dass a kongruent zu b in der Menge Z[n] ist. Diese Form der Kongruenz lässt sich auch in eine Gleichung umschreiben. Die Kongruenz entspricht der folgenden Gleichung:

```
a mod n = b mod n
```

Primzahlen und Teilbarkeit

Wir werfen eine Geburtstagsfeier. So wie es sich gehört haben wir auch eine Torte gemacht. Natürlich möchten wir fair sein und teilen diese Torte in genau so viele Teile auf wie die Anzahl der Teilnehmer an der Feier. Sind also 8 Teilnehmer anwesend, teilen wir diese Torte in genau 8 Teile. Was ist aber, wenn wir eine vorgefertigte und bereits in 20 Teile geschnittene Torte kaufen? Jedes Teil hat eine einheitliche Form. Wir möchten die einzelnen Teile nicht zerschneiden, sonst würden wir diese Form zerstören. Hier stehen wir vor einem Problem. 20 lässt sich nicht restlos durch 8 teilen. Wir müssten jedem Teilnehmer zwei Teile übergeben, denn das wäre fair. Was machen wir aber mit den vier verbleibenden Teilen?

Wir sehen, dass 20 nicht restlos durch 8 teilbar ist. Hätten wir eine Torte mit 16 Teilen gekauft, wäre das kein Problem gewesen, denn 16 ist restlos durch 8 teilbar. Die Teilbarkeit von ganzen Zahlen spielt in sehr vielen Bereichen der Mathematik eine Rolle, z.B. beim Kürzen von Brüchen. Zunächst führen wir einige mathematische Operationen ein. Die bekannteste und wichtigste Operation ist die Division. Da wir jedoch nur mit Ganzzahlen arbeiten, arbeiten wir auch nur mit *Ganzzahldivisionen*. Eine Ganzzahldivision ist nicht anders als die normale Division, mit dem Unterschied, dass vom Ergebnis der Nachkommaanteil abgeschnitten wird. So ist das Ergebnis von 20 / 8 die Zahl 2. Das tatsächliche Ergebnis wäre 2.5, doch wir haben den Nachkommaanteil abgeschnitten. Das ist gleichbedeutend mit *Abrunden*. Es ist sehr wichtig, anzumerken, dass wir niemals aufrunden. Selbst 2.9999 runden wir zu 2 ab. Als nächste Operation ist die *modulo*-Operation einzuführen. Die Operation a mod b (sprich: a *modulo* b) ist nichts anderes als der Rest der Division a / b. Somit ist 17 mod 3 der Rest der Division 17 / 3. Das Ergebnis ist also 2.

Diese beiden Operationen haben einige Eigenschaften, die ebenfalls besprochen werden müssen. 20 / 8 liefert als Ergebnis 2.5. Das heißt, 2.5 * 8 ergibt wieder 20. Wir subtrahieren jedoch den Nachkommaanteil 0.5 vom Ergebnis, somit verbleibt ein Rest von 0.5 * 8, also 4. Wir können den Rest jedoch auch ohne Kommazahlen berechnen. 20 / 8 ergibt 2, aber 2 * 8 ergibt nicht 20, sondern 16. Somit ist ein Wert von 4, oder 20 - 16 verloren

gegangen. a mod b können wir also dadurch berechnen, dass wir zuerst a durch b teilen und gleich darauf wieder mit b multiplizieren. Das Ergebnis subtrahieren wir von a.
Somit lässt sich a mod b anhand folgender Formel berechnen, sofern kein Rechner mit modulo-Operation zur Verfügung steht:

```
// Achtung: Ganzzahldivision!
a mod b = a - a / b * b

// Beispiel:
  20 mod 8
= 20 - 20 / 8 * 8
= 20 - 2 * 8
= 20 - 16
= 4
```

Hier sollen noch einige Eigenschaften der modulo-Operation zusammengefasst werden, welche für die modulare Arithmetik von sehr großer Bedeutung sind:

```
1. (a + b) mod n = ((a mod n) + (b mod n)) mod n
2. (a * b) mod n = ((a mod n) * (b mod n)) mod n
3. a^b mod n = (a mod n)^(b mod phi(n)) mod n
```

Der Beweis des ersten Satzes ist relativ simpel. Wir teilen a in eine Summe aus ar = a mod n und aq = a - ar auf.
b teilen wir in eine Summe aus br = b mod n und bq = b - br auf. Damit lässt sich die Addition umschreiben:

```
(aq + bq + ar + br) mod n
```

aq ist also a, abzüglich des Restes a mod n und ist damit auf jeden Fall durch n teilbar. Das gleiche gilt für bq. Diese beiden Werte können also aus der Addition gestrichen werden und es verbleiben nur noch die Summanden ar = a mod n und br = b mod n:

```
  (aq + bq + ar + br) mod n
= (ar + br) mod n
= ((a mod n) + (b mod n)) mod n
```

Auch der zweite Satz ist relativ einfach zu beweisen. Wir teilen a und b wie eben beschrieben auf:

```
  (a * b) mod n
= ((aq + ar) * (bq + br)) mod n
```

Danach wenden wir das Distributivgesetz der Multiplikation an und erhalten:

```
  ((aq + ar) * (bq + br)) mod n
= ((aq * bq) + (aq * br) + (ar * bq) + (ar * br)) mod n
```

Da aq und bq restlos durch n teilbar sind, sind auch Vielfache von aq und bq restlos durch n teilbar. Wir können also alle Vielfachen von aq und bq aus der Addition streichen und erhalten:

```
  ((aq * bq) + (aq * br) + (ar * bq) + (ar * br)) mod n
= (ar * br) mod n
= ((a mod n) * (b mod n)) mod n
```

Der dritte Satz ist etwas kniffeliger zu beweisen. Hier wurde die Eulersche Funktion *phi* angewandt. Diese wird später vorgestellt. Der Satz ist besonders interessant. Ist b = 0 (mod phi(n)), sollte das Ergebnis eigentlich 1 sein. Das trifft jedoch nur dann zu, wenn a *relativ prim* (siehe unten) zu n ist. Der Grund wird im Kapitel über die Eulersche phi-Funktion aufgezeigt. Da die phi-Funktion erst später besprochen wird, lässt sich der Satz an dieser Stelle noch nicht beweisen.

Wir gehen weiterhin davon aus, dass wir nur mit Ganzzahlen arbeiten, uns also keine Möglichkeit für Kommazahlen zur Verfügung steht. Wir können also testen, ob a durch b teilbar ist, indem wir a durch b teilen und anschließend sofort wieder mit b multiplizieren. Ist das Ergebnis a, so ist a durch b teilbar. Liefert a / b * b als Ergebnis a, dann ist auch der Rest 0, denn in dem Fall gilt a - a / b * b = 0 bzw. a - a = 0. Wir können also sagen, a ist durch b teilbar, wenn a mod b = 0 gilt; oder als Kongruenz ausgedrückt: a = 0 (mod b). Weiters gilt:

(k * n) mod n = 0 für jede beliebige Ganzzahl k, denn
eine Zahl teilt immer restlos Vielfache von sich selbst.

Beweis:

```
(k * n) mod n = 0          // Nach der oben genannten Formel:
k * n - k * n / n * n = 0
k * n - k * n = 0
```

Als Kongruenz ausgedrückt: k * n = 0 (mod n). Wir können also auch sagen: k * n + x = x (mod n). Dies ist ein sehr wichtiger Satz für die modulare Arithmetik. Mehr dazu später.

Es gibt bestimmte Zahlen, die genau zwei Teiler haben. Sie lassen sich nur durch 1 und durch sich selbst teilen. Ein Beispiel ist die 7. Die 7 lässt sich nur durch 1 und durch 7 teilen. Ein anderes Beispiel ist die 47, welche nur durch 1 und 47 teilbar ist. Um dies zu verdeutlichen, greifen wir das obige Beispiel mit der Torte noch einmal auf. Wäre die Anzahl der Tortenstücke eine solche Zahl, so hätten wir auf jeden Fall einen Rest, wenn die Anzahl der Teilnehmer weder 1, noch die Anzahl der Tortenstücke ist. Hätten wir also eine Torte mit 17 Teilen, so müssten entweder einer oder 17 Teilnehmer anwesend sein. Bei jeder beliebigen Zahl zwischen 1 und 17 würde ein Rest verbleiben. Zahlen mit dieser Teilbarkeitseigenschaft nennen wir *Primzahlen*. Es ist bereits erwiesen, dass es unendlich viele Primzahlen gibt. Die kleinste Primzahl ist 2. Die Zahl 1 ist keine Primzahl, denn sie hat nur einen Teiler: 1. Auch die Zahl 0 ist keine Primzahl, denn sie hat unendlich viele Teiler. Jede Zahl (außer 0 natürlich) teilt die 0 restlos.

Beweis:

```
0 mod x = 0        // Nach der obigen Formel:
0 - 0 / x * x = 0
0 - 0 = 0
```

Eine interessante Eigenschaft der Primzahlen ist, dass sich jede beliebige

Ganzzahl größer als 1 als Produkt von Primzahlen darstellen lässt. Beispielsweise lässt sich 42 als Produkt der Primzahlen 2, 3 und 7 darstellen: 2 * 3 * 7 = 42. Die einzelnen Faktoren werden dabei *Primfaktoren* genannt. Noch viel interessanter ist, dass man jeder Zahl größer als 1 einen eindeutigen Satz Primfaktoren zuordnen kann. 2 * 3 * 7 ergibt auf jeden fall 42 und es gibt garantiert keine andere Zahl, die sich als Produkt von 2, 3 und 7 darstellen lässt. Es gibt auch garantiert keinen anderen Weg, die 42 in Primfaktoren zu zerlegen. Man kann also sagen, jede Zahl lässt sich auch eindeutig als Produkt von Primzahlen darstellen.

Oftmals kommen das *kleinste gemeinsame Vielfache* (kgV) und (vor allem bei RSA) der *größte gemeinsame Teiler* (ggT) zum Einsatz. Das kgV ist die kleinste Zahl, die sich gleichzeitig als Vielfaches zweier Zahlen a und b darstellen lässt. Das kgV von 15 und 24 ist also 120, denn es gibt keine kleinere Zahl, die gleichzeitig Vielfaches von 15 *und* 24 ist. Der ggT ist die größte Zahl, die zwei Zahlen a und b restlos teilt. Der ggT von 12 und 15 ist also 3, denn es gibt keine größere Zahl, die 12 *und* 15 restlos teilt.

Das kgV lässt sich berechnen, indem man die Primfaktoren von a und b kombiniert, von den gemeinsamen Faktoren aber nur die höchste Potenz verwendet. Beispiel:

```
// kgV von 56 und 60:
56 = 2 * 2 * 2 * 7 = 2^3 * 7
60 = 2 * 2 * 3 * 5 = 2^2 * 3 * 5

kgv(56, 60) = 2^3 * 3 * 5 * 7 = 840
```

Der ggT lässt sich dadurch berechnen, dass man die *gemeinsamen* Primfaktoren von a und b kombiniert:

```
// ggT von 350 und 616
350 = 2 * 5^2 * 7  = 2 * 5 * 5 * 7
616 = 2^3 * 7 * 11 = 2 * 2 * 2 * 7 * 11

ggT(350, 616) = 2 * 7 = 14
```

Sind keine gemeinsamen Primfaktoren aufzufinden, so ist 1 der größte

gemeinsame Teiler, denn 1 teilt jede Zahl restlos. Wenn der größte gemeinsame Teiler zweier Zahlen 1 ist, so sagt man, die beiden Zahlen sind *teilerfremd* oder *relativ prim*, sie haben also keine Primfaktoren gemeinsam. Zum Thema ggT wird später der *Euklidsche Algorithmus* vorgestellt, mit dem sich der ggT auf systematischem Wege berechnen lässt, ohne a und b erst faktorisieren zu müssen. Wir verwenden ab jetzt die englische Bezeichnung *gcd* (engl. greatest common divisor) anstatt ggT für den größten gemeinsamen Teiler.

Modulare Arithmetik I - Einführung

Um RSA überhaupt verstehen zu können, müssen wir uns erst einmal einer ganz neuen Zahlenmenge zuwenden, der *endlichen Menge*. Diese Menge ist ungewöhnlich. Interessant wird sie dadurch, dass fast alle normalen Rechenregeln auch in dieser Menge Geltung finden. Wie der Name schon sagt, handelt es sich bei der endlichen Menge um eine Menge, in der es nicht unendlich viele Elemente gibt. Wir arbeiten mit einer Form der endlichen Menge, in der es, von 0 beginnend, nur Ganzzahlen bis zu einem bestimmten Maximalwert gibt. In einer solchen Menge findet die modulare Arithmetik Anwendung. Wir bezeichnen diese Menge als Z[n], wobei n die Anzahl der Zahlen in der Menge (das *Modul*) darstellt. Ist n = 14, so heißt die Menge Z[14] und gibt es 14 Zahlen in der Menge: 0 bis 13. Modulare Arithmetik ist nicht anders als die normale Arithmetik, mit dem Unterschied, dass wir nur auf Ganzzahlen operieren und alle Rechenoperationen *modulo n* durchführen.

In diesem Text werden oft modulare Kongruenzen der Form $x = y \pmod n$ dargestellt. Diese Kongruenz drückt aus, dass x und y in der Menge Z[n] gegenseitig ersetzbar sind. Das heißt, dass x und y in Z[n] die gleichen Zahlen sind. Beispiel: $17 = 6 \pmod{11}$. Erinnerung: die Kongruenz lässt sich auch zu einer Gleichung umschreiben: $x = y \pmod n$ entspricht der Gleichung $x \bmod n = y \bmod n$.

Am besten lässt sich die endliche Menge am Beispiel einer Uhr verdeutlichen. Der Stundenzeiger einer Uhr arbeitet in der Menge Z[12]. Es gibt also die Stunden 0 bis 11. Wenn wir gerade 9 Uhr haben, dann ist es nach 5 Stunden 2 Uhr. Das ist das Prinzip des *Umklappens*. Nach der Stunde 11 klappt der Zeiger wieder um und landet auf der Stunde 0 (meist als 12 gekennzeichnet). Diese Rechnung können wir allerdings auch mathematisch durchführen:

```
9 + 5 = 14     // Normale Addition,
14 mod 12 = 2  // jedoch modulo 12
```

Die modulo-Operation a mod b ist nichts anderes als der Rest der Division a / b (siehe Primzahlen und Teilbarkeit). Was haben wir also hier getan? Wir haben normal addiert und das Ergebnis dann durch die modulo-Operation wieder in die Menge Z[12] gebracht. Das garantiert, dass weder ein Operand, noch ein Ergebnis jemals größer oder gleich 12 sind. Wir können dieses Spiel fortsetzen. Wenn es jetzt 7 Uhr ist, wieviel Uhr ist es dann nach 16 * 17 Stunden? Stellen wir also unsere Aufgabe auf:

```
x = 7 + 16 * 17 (mod 12)
```

Der eine oder andere Leser dürfte bereits angefangen haben, 16 * 17 auszurechnen, was jedoch völlig unnötig ist. Hier zeigt sich, dass die modulare Arithmetik viele Rechenaufgaben erheblich erleichtert. Wir greifen die oben genannten Regeln der modulo-Operation auf:

```
(a + b) mod n = ((a mod n) + (b mod n)) mod n
(a * b) mod n = ((a mod n) * (b mod n)) mod n
```

Das heißt, wir können unseren Term (7 + 16 * 17) mod 12 etwas abändern:

```
  (    7      +      16 * 17      ) mod 12
= ((7 mod 12) + ((16 * 17) mod 12)) mod 12
```

Nun haben wir einen Nebenterm (16 * 17) mod 12 geschaffen, den wir

ebenfalls abändern können:

```
  (16 * 17) mod 12
= ((16 mod 12) * (17 mod 12)) mod 12
= (4 * 5) mod 12
```

Wir können also bei Additionen und Multiplikationen jede beliebige Zahl erst mal durch eine modulo-Operation in die Menge Z[12] bringen. Weder 16, noch 17 sind in der Menge Z[12], also bringen wir sie erst in die Menge:

```
16 * 17 = (16 mod 12) * (17 mod 12) (mod 12)  // Wir "reduzieren" 16 und 17
16 * 17 = 4 * 5 (mod 12)                      // Normale Multiplikation
16 * 17 = 20 (mod 12)                         // Auch 20 ist nicht in Z[12]
16 * 17 = 8 (mod 12)                          // 20 = 8 (mod 12)
```

Hier haben wir nun aus allen zu großen Zahlen den Divisionsrest durch 12 gebildet. 16 mod 12 ist 4, 17 mod 12 ist 5, also lautet unsere neue Gleichung:

```
x = 7 + 4 * 5 (mod 12)
```

Jetzt berechnen wir noch 4 * 5 und kommen auf 20. Auch die 20 ist außerhalb von Z[12], also wird sie wieder reduziert.
20 mod 12 ist 8. Nun unsere neue, stark simplifizierte Gleichung, die wir jetzt problemlos auflösen können:

```
x = 7 + 8 (mod 12)  // Normale Addition
x = 15 (mod 12)     // 15 ist außerhalb Z[12], also...
x = 3 (mod 12)      // ... reduzieren wir wieder: 15 mod 12 = 3
```

Und unsere Antwort: wenn es jetzt 7 Uhr ist, ist es nach 16 * 17 Stunden 3 Uhr.

So einfach? Unmöglich! Also überprüfen wir unser Ergebnis und rechnen das ganze noch einmal auf die harte Tour:

```
x = 7 + 16 * 17 (mod 12)
x = 7 + 272 (mod 12)
x = 279 (mod 12)
x = 3 (mod 12)            // 279 = 3 (mod 12)
```

Tatsächlich! Ach, wenn wir doch alles auf der Welt *modulo 12* rechnen könnten! Die modulo-Operation lässt sich übrigens auch so beschreiben: wir stellen den Zeiger auf 0 Uhr und drehen ihn einmal rund herum. Damit sind 12 Stunden vergangen und wir sind wieder auf 0 Uhr (das ist analog zu der weiter oben aufgeführten Kongruenz k * n = 0 (mod n)). Also ziehen wir 12 von 279 ab. Das machen wir so oft, bis unsere Zahl unter 12 ist. Gleiches Prinzip: wir stellen ihn auf 7 Uhr und fahren 272 Stunden (16 * 17) weiter. So kann der Skeptiker das an einer echten Uhr überprüfen.

Im Kapitel über Primzahlen und Teilbarkeit wurde erklärt, dass sich a mod b durch die Formel a mod b = a - a / b * b (Ganzzahldivision!) berechnen lässt. Um die modulare Arithmetik besser verstehen zu können, soll hier allerdings noch ein weiterer Weg vorgestellt werden: a mod b lässt sich auch berechnen, indem man wiederholt b von a subtrahiert, bis a kleiner als b ist. Dies kann man optimieren, indem man erst Vielfache von b subtrahiert. Damit lässt sich 632457 mod 412 dadurch berechnen, dass man wiederholt 412000 von 632457 abzieht, bis das Ergebnis kleiner als 412000 ist. Von diesem zieht man dann 41200 ab, bis das Ergebnis kleiner als 41200 ist. Von diesem zieht man dann wiederholt 4120 ab und schließlich 412. So lässt sich die modulo-Operation sehr einfach von Hand durchführen.

Modulare Arithmetik II - Definition

Nun zu den trockenen Fakten. Für jedes Element x der Menge Z[n] gilt die Kongruenz x = x + k * n (mod n) für jede beliebige Zahl k (merke: wir arbeiten nur mit Ganzzahlen, somit muss auch k eine solche sein). Verallgemeinert lautet die Kongruenz: k * n = 0 (mod n).

Beweis: da k eine Ganzzahl
ist, ist k * n garantiert ein Vielfaches von n. Da eine
Zahl immer restlos Vielfache von sich selbst teilt, ist der Divisionsrest
immer 0. Siehe Primzahlen und Teilbarkeit für eine
detailiertere Erklärung.

Die Subtraktion a - b ist definiert, solange b kleiner
als a ist, da sonst negative Zahlen resultieren, welche weder
Elemente der Menge Z[n] sind, noch sich durch die modulo-Operation
korrekt in diese überführen lassen. Die Division ist gar nicht definiert. Es
gibt jedoch multiplikative Inverse. Mehr dazu weiter unten.

In der Menge Z[n] gelten alle normalen Gesetze für die Addition. Zu
jedem Element x ungleich 0 der Menge Z[n] existiert ein
additives inverses Element -x und es existiert genau ein neutrales
Element der Addition: 0. Hier muss erwähnt werden, dass es sich bei der
Darstellung -x **nicht** um eine negative Zahl
handelt. Die Darstellung mit dem negativen Vorzeichen ist reine Formsache.
Addition des additiven Inversen eines Elements macht eine Addition mit dem
Element selbst rückgängig. Addition des inversen Elements -x zu
x resultiert im neutralen Element. Addition des neutralen Elements
der Addition hat keine Wirkung. Das neutrale Element ist invers zu sich
selbst:

```
a + x = b (mod n)   // Addition des Elements x
b + -x = a (mod n)  // Addition des additiven Inversen zu x ("Subtraktion")

a + 0 = a (mod n)   // Addition des neutralen Elements der Menge Z[n]
x + -x = 0 (mod n)  // Addition von -x zu x ergibt neutrales Element

// Zusammenfassung (addition des neutralen Elements):
a + (x + -x) = a (mod n)
```

Das additive Inverse eines Elements x in der Menge Z[n]
ist definiert als n - x.

Beweis:

```
x + -x = 0 (mod n)
-x = 0 - x (mod n)
// bzw.:
-x = n - x (mod n)

// denn:
x + (n - x) = n (mod n)
// und:
n = 0 (mod n)
```

Man erinnere sich an die obige Kongruenz k * n = 0 (mod n). Wir wählten 1 für k, denn 1 * n - x ist immer in der Menge Z[n], solange x selbst in dieser ist, denn dann ist x garantiert kleiner als n. Dadurch ist die Subtraktion gültig und das Ergebnis muss nicht erst reduziert werden.

Somit ist das additive Inverse zu 9 in der Menge Z[12] errechenbar: 12 - 9 = 3. Überprüfung:

```
7 + 9 = 4 (mod 12)
4 + 3 = 7 (mod 12)

  7 + (9 + 3) (mod 12)
= 7 + 12 (mod 12)
= 7 + 0 (mod 12)
= 7 (mod 12)
```

Für die Addition gelten das Kommutativgesetz (a + b = b + a (mod n)) und das Assoziativgesetz ((a + b) + c = a + (b + c) (mod n)). Somit ist Z[n] eine *additive Gruppe*.

In der Menge Z[n] gelten, sofern n prim ist, alle normalen Gesetze der Multiplikation. Zu jedem Element x › 0 in der Menge Z[n] (n prim) gibt es genau ein multiplikatives

Inverses 1/x und es existiert genau ein neutrales Element der Multiplikation: 1. Hier ist zu erwähnen, dass die Bruchschreibweise reine Formsache ist. Hierbei handelt es sich **nicht** um einen echten Bruch, da dies zu Kommazahlen oder falschen Ergebnissen führen würde (die Division ist in Z[n] ohnehin nicht definiert). Multiplikation mit dem multiplikativen Inversen eines Elements macht eine Multiplikation mit dem Element selbst rückgängig. Multiplikation eines Elements x mit seinem multiplikativen Inversen 1/x resultiert im neutralen Element. Multiplikation mit dem neutralen Element hat keinen Einfluss. Das neutrale Element ist invers zu sich selbst. Multiplikation mit 0 resultiert in 0 und ist nicht umkehrbar, daher hat 0 kein multiplikatives Inverses:

```
a * x = b (mod n)    // Multiplikation mit x
b * 1/x = a (mod n)  // Multiplikation mit dem multiplikativen Inversen von x

a * 0 = 0 (mod n)    // Multiplikation mit 0 ist nicht umkehrbar
a * 1 = a (mod n)    // Multiplikation mit dem neutralen Element
x * 1/x = 1 (mod n)  // Multiplikation von x und 1/x ergibt neutrales Element

// Zusammenfassung (Multiplikation mit neutralem Element):
a * (x * 1/x) = a (mod n)
```

Das multiplikative Inverse eines Elements x in der Menge Z[n] (n prim!) ist definiert als x^(n - 2) (mod n) (der Beweis folgt im Abschnitt über die Eulersche phi-Funktion). Somit ist das multiplikative Inverse zu 4 in der Menge Z[13] errechenbar: 4^(13 - 2) = 4^11 = 10 (mod 13).
Überprüfung:

```
8 * 4 = 6 (mod 13)
6 * 10 = 8 (mod 13)

  8 * (4 * 10) (mod 13)
= 8 * 40 (mod 13)
= 8 * 1 (mod 13)
= 8 (mod 13)
```

Für die Multiplikation gelten, auch, wenn n nicht prim ist, das Kommutativgesetz (a * b = b * a (mod n)), das Assoziativgesetz ((a * b) * c = a * (b * c) (mod n)) und das Distributivgesetz (a * (b + c) = a * b + a * c (mod n)). Z[n] ist jedoch nur dann eine *multiplikative Gruppe*, wenn n prim ist, da sonst nicht jedes Element x › 0 ein eindeutiges multiplikatives Inverses besitzt. Dies kann man sich durch die folgenden

Multiplikationsmatrizzen verdeutlichen:

```
Menge: Z[7] (multiplikative Gruppe, da n prim)

    0  1  2  3  4  5  6
  *---------------------
0 | 0  0  0  0  0  0  0
1 | 0  1  2  3  4  5  6
2 | 0  2  4  6  1  3  5
3 | 0  3  6  2  5  1  4
4 | 0  4  1  5  2  6  3
5 | 0  5  3  1  6  4  2
6 | 0  6  5  4  3  2  1

Menge: Z[9] (keine multiplikative Gruppe, da n nicht prim)

    0  1  2  3  4  5  6  7  8
  *---------------------------
0 | 0  0  0  0  0  0  0  0  0
1 | 0  1  2  3  4  5  6  7  8
2 | 0  2  4  6  8  1  3  5  7
3 | 0  3  6 (0) 3  6 (0) 3  6
4 | 0  4  8  3  7  2  6  1  5
5 | 0  5  1  6  2  7  3  8  4
6 | 0  6  3 (0) 6  3 (0) 6  3
7 | 0  7  5  3  1  8  6  4  2
8 | 0  8  7  6  5  4  3  2  1
```

Hier zeigt sich deutlich, dass in Z[9] nur Multiplikationen mit Elementen, die *relativ prim* (teilerfremd) zum Modul (9) sind, auch eindeutig umkehrbar sind. Wer genau hinsieht, dem dürfte auch auffallen, dass die restlichen Elemente auch keine multiplikativen Inversen besitzen (man erinnere sich: x * 1/x = 1). Es gibt in Z[9] keine Zahl, die, mit 3 oder 6 multipliziert, das neutrale Element 1 ergibt. Damit ist Z[9] keine multiplikative Gruppe.

Hier soll jedoch erwähnt werden, dass jedes Element x in der Menge Z[n] ein eindeutiges multiplikatives Inverses besitzt, wenn x und n relativ prim sind. Auch dann, wenn n nicht prim ist. In diesem Fall kann dieses inverse Element jedoch nicht durch die obige Formel x^(n - 2) (mod n) berechnet werden. Wie man dieses Element ermittelt, wird im Abschnitt über den *erweiterten Euklidschen*

Algorithmus erklärt.

Da in Z[n] die normalen Gesetze der Multiplikation gültig sind, sind auch die normalen Gesetze der Exponentiation gültig, auch, wenn n nicht prim ist. Diese lauten:

```
a^0 = 1 (mod n)  // Null-Exponent: 0
a^1 = a (mod n)  // Neutraler Exponent: 1

(a^b)^c = (a^c)^b (mod n)      // Kommutativgesetz
(a^b)^c = a^(b * c) (mod n)    // Assoziativgesetz
a^b * a^c = a^(b + c) (mod n)  // Distributivgesetz
```

Die Bezeichnungen *Kommutativgesetz*, *Assoziativgesetz* und *Distributivgesetz* sind in diesem Fall nicht ganz richtig, passen jedoch am ehesten zu den gegebenen Rechengesetzen.

Modulare Arithmetik III - Modulare Exponentiation

Exponentiation ist ein sehr rechenintensiver Prozess. Allein der sehr simpel aussehende Term 4132^4152 liefert simplifiziert eine Zahl mit 15015 Dezimalstellen. Die Dinge ändern sich jedoch dramatisch, wenn wir in der endlichen Menge rechnen. Dazu sehen wir uns den Term 3^4 (mod 11) genauer an:

```
3^4 = 3 * 3 * 3 * 3 (mod 11)
```

Nun greifen wir eine Regel der modulo-Operation (siehe Primzahlen und Teilbarkeit) auf, die wir auf diesen Term anwenden können:

```
(a * b) mod n = ((a mod n) * (b mod n)) mod n
```

Das heißt, wir können nach jeder Multiplikation das Ergebnis wieder reduzieren und erhalten damit einen Term, der sehr einfach zu lösen ist:

```
  3^4 (mod 11)
= 3 * 3 * 3 * 3 (mod 11)
= (((3 mod 11) * 3 mod 11) * 3 mod 11) * 3 (mod 11)
= ((3 * 3 mod 11) * 3 mod 11) * 3 (mod 11)
= (9 * 3 mod 11) * 3 (mod 11)
= 5 * 3 (mod 11)
= 4 (mod 11)
```

Wir probieren dies anhand des Beispiels 63^165 (mod 285) aus. Die Vereinfachung dieses Prozesses basiert darauf, dass wir ihn in mehrere Teilprozesse unterteilen und an jeder Zwischenstation (dort, wo multipliziert wird) das Ergebnis wieder in die Menge Z[285] bringen. Zunächst teilen wir den Exponenten in eine Summe von Zweierpotenzen (2^x) auf und zerlegen die Exponentiation entsprechend den im letzten Abschnitt genannten Gesetzen:

```
  63^165 (mod 285)
= 63^(128 + 32 + 4 + 1) (mod 285)
= 63^128 * 63^32 * 63^4 * 63 (mod 285)
= 63^(2^7) * 63^(2^5) * 63^(2^2) * 63 (mod 285)
```

Auf den ersten Blick sieht unser neuer Term noch schlimmer aus als der alte, doch wir sehen, dass es sich nur noch um wiederholte Quadrierung handelt. Also ziehen wir Schritt für Schritt Quadrate. Zuerst nehmen wir das Quadrat von 63 und erhalten 3969. Dieses Quadrat ist nicht mehr in Z[285], also kürzen wir es:

```
3969 = 264 (mod 285)
```

Nun ersetzen wir jedes Vorkommen von 63^2 durch 264:

```
  63^(2^7) * 63^(2^5) * 63^(2^2) * 63 (mod 285)
= 264^(2^6) * 264^(2^4) * 264^2 * 63 (mod 285)
```

Dieses Spiel setzen wir fort, bis wir nur noch eine Reihe Multiplikationen

haben:

```
  264^(2^6) * 264^(2^4) * 264^2 * 63 (mod 285)  // 264^2 = 156 (mod 285)
= 156^(2^5) * 156^(2^3) * 156 * 63 (mod 285)    // 156^2 = 111 (mod 285)
= 111^(2^4) * 111^(2^2) * 156 * 63 (mod 285)    // 111^2 = 66 (mod 285)
= 66^(2^3) * 66^2 * 156 * 63 (mod 285)          // 66^2 = 81 (mod 285)
= 81^(2^2) * 81 * 156 * 63 (mod 285)            // 81^2 = 6 (mod 285)
= 6^2 * 81 * 156 * 63 (mod 285)                 // 6^2 = 36 (mod 285)
= 36 * 81 * 156 * 63 (mod 285)
```

Was haben wir nun getan? Aus unserer gigantischen Exponentation sind nur noch drei simple Multiplikationen übrig geblieben, die wir noch durchführen müssen. Aber auch hier können wir weiter von der Eigenschaft der endlichen Menge Gebrauch machen, indem wir die Ergebnisse der Multiplikation auch reduzieren:

```
  36 * 81 * 156 * 63 (mod 285)
= 2916    * 9828     (mod 285)
= 66      * 138      (mod 285)
= 9108 (mod 285)
= 273 (mod 285)
```

Diese Exponentiation hätte man sogar von Hand durchführen können, da man immer allerhöchstens dreistellige Zahlen multipliziert. Auch das scheint zu schön, um wahr zu sein, also probieren wir es auch hier auf die harte Tour und berechnen 63^165:

```
778378192247619740380721780831222290114610481624105151532375518829875807596
116289582565299183256830709798510100387818773018002979076581023295840489802
219795340758238588944653298749004558556165447877535493042131134313364098586
596859205285801099216441875976059243003292302074746769640034486477031743

// Diese Zahl modulo 285 ergibt:
273
```

Natürlich kann das nur der nachprüfen, der einen geeigneten Rechner hat. Die Kommandozeilenprogramme bc und dc (enthalten in allen Unix- und in den meisten Linux-Distributionen) eignen sich hervorragend für Berechnungen mit unbegrenzt großen Zahlen. Als kleiner Hinweis: das Programm dc hat sogar einen Operator für die modulare Exponentiation, doch dies übersteigt den Rahmen dieser Dokumentation. Für Windows gibt es diverse Algebraprogramme wie Mathematica und Derive. Diese sind jedoch kostenpflichtig. Für rein numerische Berechnungen oder simplere

Algebraprogramme führt eine kurze Suche bei SourceForge vielleicht zum Erfolg. Dort finden sich kostenfreie Programme für diese Zwecke.

Diese Art der Exponentiation lässt sich sehr einfach von einem Computer durchführen. Dazu betrachten wir das, was wir oben getan haben, von einer anderen Perspektive. Bleiben wir bei der Aufgabe 63^165 (mod 285). Wir können nun entweder 165 mal die 63 mit sich selbst multiplizieren, oder wir gehen etwas systematischer vor. Ausgänglich haben wir den Wert m = 63 und e = 165. Wir führen zwei zusätzliche Variablen r und s. In der Variablen r halten wir immer das aktuelle Ergebnis und in der Variablen s führen wir immer eine Exponentiation von 63 mit Zweierpotenzen, so wie wir das oben von Hand gemacht haben. s wird mit m initialisiert und r wird mit 1 initialisiert. Damit lässt sich 63^165 (mod 285) systematisch ausrechnen. Wir sehen, ob e ungerade ist. Wenn ja, multiplizieren wir r mit dem Wert von s und dekrementieren e um 1. Wenn nicht, dann multiplizieren wir s mit sich selbst und halbieren e. Sobald e = 0 ist, erhalten wir in r das gewünschte Ergebnis. Alle Multiplikationen werden modulo n durchgeführt. Das garantiert, dass keines der Multiplikationsergebnisse größer als (n - 1)^2 sein kann.

```
m = 63
n = 285

  s    r    e
---  ---  ---
 63    1  165  e ungerade, also: r = r * s; e = e - 1
 63   63  164  e gerade, also: s = s * s; e = e / 2
264   63   82  e gerade
156   63   41  e ungerade
156  138   40  e gerade
111  138   20  e gerade
 66  138   10  e gerade
 81  138    5  e ungerade
 81   63    4  e gerade
  6   63    2  e gerade
 36   63    1  e ungerade
 36  273    0  Schleife abgeschlossen, r enthält das Ergebnis
```

Mehr auf mathematischer Basis erklärt funktioniert der Algorithmus folgendermaßen. Wenn e ungerade ist, wird m^e (mod n) zurückgeführt auf m^(e - 1) * m (mod n). Ist e gerade, so wird m^e (mod n) zurückgeführt auf (m * m)^(e / 2) (mod n). Dieser Algorithmus beweist sich selbst, denn es gilt: m^e = m^(e - 1) * m und, wenn e gerade ist: m^e = (m * m)^(e / 2).

Die folgenden Pseudocodes implementieren die modulare Exponentiation einmal wie eben beschrieben und einmal rekursiv. Meist wird in Dokumentationen die rekursive Form aufgezeigt. Diese ist jedoch langsamer, unsicherer und schwerer zu verstehen. Je mehr Stellen der Exponent hat, desto öfter muss die Funktion sich selbst aufrufen. Genau genommen muss sie sich für jede binäre Stelle im Exponent genau einmal selbst aufrufen. Für jede binäre 1, die dabei angetroffen wird, ruft sie sich zusätzlich noch einmal selbst auf, außer die letzte (höchstwertige) 1. Dies kann bei besonders großen Zahlen oder besonders kleinem Stapelspeicher zu einem Stapelüberlauf und in den meisten Fällen zum Programmabsturz führen. Wer diesen Text liest, dürfte am Thema *Sicherheit* interessiert sein. Ein Programm, dessen Stapel überläuft, ist **nicht** sicher! Der Vollständigkeit halber wird die rekursive Version hier dennoch vorgeführt. Die beiden Funktionen rechnen exakt nach dem selben Algorithmus. Die eine arbeitet, wie oben beschrieben, iterativ, und die andere arbeitet rekursiv:

```
// Iterative Version:
MODEXP(m, e, n) {
  LOCAL s, r

  s = m
  r = 1
  WHILE(e != 0) {
    IF (e MOD 2 == 1) THEN
      r = r * s MOD n
      e = e - 1
    ELSE
      s = s * s MOD n
      e = e / 2
    ENDIF
  }
  RETURN r
}

// Rekursive Version:
MODEXP_RC(m, e, n) {
```

```
  IF (e == 1) THEN RETURN m

  IF (e MOD 2 == 1) THEN
    RETURN (m * MODEXP_RC(m, e - 1, n)) MOD n
  ELSE
    RETURN MODEXP_RC((m * m) MOD n, e / 2, n)
  ENDIF
}
```

Die iterative Version lässt sich übrigens noch weiter optimieren. Wenn e gerade ist, wird es halbiert. Das so oft, bis e ungerade wird. Das können wir in eine eigene innere Schleife einschließen. Solange e gerade ist, führen wir s = s * s (mod n) und e = e / 2 (mod n) durch. Nachdem wir die Schleife verlassen, ist garantiert, dass e ungerade ist. Hier also die optimierte Version:

```
// Optimierte iterative Version:
MODEXP(m, e, n) {
  LOCAL s, r

  s = m
  r = 1
  WHILE(e != 0) {
    WHILE(e MOD 2 == 0) {
      s = s * s MOD n
      e = e / 2
    }
    r = r * s MOD n
    e = e - 1
  }
  RETURN r
}
```

In den meisten Programmiersprachen wie C lässt sich die Funktion noch einmal zusätzlich optimieren. Wir können statt s auch einfach m weiterverwenden, da die Variable in den meisten Programmiersprachen ohnehin lokal zur Funktion ist. Das spart eine Zuweisung und je nach Größe der Zahlen eine immense Menge Stapelspeicher. Ob e gerade ist, können wir testen, indem wir das niederwertigste Bit abfragen. Ist dieses 0, so ist e gerade. Die Division durch zwei können wir durch eine Rechtsschiebeoperation optimieren und die Subtraktion können wir, da die Zahl auf jeden Fall ungerade ist, durch eine binäre XOR-Operation durchführen. Also hier noch einmal die stark optimierte Version in einem C-ähnlichen Pseudocode:

```
// Weiter optimierte iterative Version:
huge_t modexp(huge_t m, huge_t e, huge_t n) {
  huge_t res = 1;

  while(e) {
    while(!(e & 1)) {
      m = m * m % n;
      e ››= 1;
    }
    res = res * m % n;
    e ^= 1;
  }
  return res;
}
```

ggT I - Der Euklidsche Algorithmus

Dieser Algorithmus berechnet den größten gemeinsamen Teiler zweier nicht negativer Ganzzahlen. Die Funktion gcd(a, b) (engl. greatest common divisor) hat folgende Eigenschaften:

```
// a und g sind Elemente von N
// b ist Element von N[0]
gcd(a, b) = g

// u und v sind Elemente von Z. Es gibt unendlich viele Paare (u, v), die die
// folgende Gleichung Erfüllen:
g = u * a + v * b
```

Der Algorithmus führt gcd(a, b) wiederholt auf gcd(b, a mod b) zurück. Diese Schleife endet, sobald a mod b = 0 ist. Dann ist b der größte gemeinsame Teiler von den ursprünglichen a und b. Ist dieser 1, so sind a und b *relativ prim* (teilerfremd), haben also keine Primfaktoren gemeinsam.

Somit ist der Euklidsche Algorithmus definiert als:

```
// Wenn b ungleich 0
gcd(a, b) = gcd(b, a mod b)

// Wenn b gleich 0
gcd(a, b) = a
```

Als kleines Beispiel ermitteln wir den größten gemeinsamen Teiler von 11 * 13 * 17 * 23 = 55913 und 11^2 * 13^2 = 20449. Der größte gemeinsame Teiler ist übrigens die Kombination der gemeinsamen Primfaktoren. In dem Fall müsste das Ergebnis 11 * 13 = 143 lauten:

```
    a      b  a mod b
-----  -----  -------
55913  20449    15015  Hier berechnen wir a mob b und...
20449  15015     5434  ... setzen das Ergebnis nach b; das alte b wird zu a.
15015   5434     4147  Das wiederholen wir...
 5434   4147     1287  ... nochmal,
 4147   1287      286  ... nochmal,
 1287    286      143  ... nochmal,
  286    143        0  ... nochmal,
  143      0        -  bis b = 0 ist. Dann ist a der ggT.
```

Der folgende Pseudo-Code sollte dies veranschaulichen. Hier werden wieder eine iterative und eine rekursive Version aufgezeigt. Die rekursive Funktion wird wieder nur der Vollständigkeit halber aufgeführt. Sie sollte nicht verwendet werden, da sie langsamer und unsicherer ist. Anmerkung: wie oft sich die rekursive Funktion selbst aufruft, hängt allein von den beiden Funktionsparametern ab. Es kann vorkommen, dass sich die Funktion sehr oft selbst aufruft. Im obigen Falle würde sie sich genau 7 mal selbst aufrufen. Das ist purer Zufall. Es kann passieren, dass sich die Funktion bei größeren Zahlen mehrere Millionen mal selbst aufrufen muss. Das passiert nur bei wenigen Zahlenpaaren, kommt aber vor.

```
// Iterative Version
GCD(a, b) {
  LOCAL x, y, temp

  x = a
```

```
  y = b
  WHILE(y != 0) {
    temp = x
    x = y
    y = temp MOD y
  }
  RETURN x
}

// Rekursive Version
GCD_RC(a, b) {
  IF (b == 0) THEN
    RETURN a;
  ELSE
    RETURN GCD_RC(b, a MOD b)
  ENDIF
}
```

In den meisten Programmiersprachen wie C/C++ sind die beiden lokalen Variablen x und y überflüssig. Man kann auch direkt auf a und b operieren. Dies ist jedoch nicht in jeder Programmiersprache der Fall und deswegen legen wir zwei lokale Variable an.

Wie viele Verfahren, lässt sich auch dieses geometrisch beweisen. Euklid suchte die größte gemeinsame Einheit zweier Längen. Durch gegenseitige Subtraktion (hier optimiert durch die modulo-Operation) bleibt zum Schluss die größte gemeinsame Einheit übrig. Auf rein algebraischer Basis lässt sich der Algorithmus folgendermaßen beweisen. Wir wissen bereits, dass die gemeinsamen Teiler zweier Summanden auch Teiler der Summe sind. Beweis:

```
x = a * b
y = a * c
x + y = (a * b) + (a * c) = a * (b + c)

// Beispiel:
18 = 2 * 3^2
60 = 2^2 * 3 * 5
78 = 2 * 3 * 13
```

Wir wissen, dass x * k durch x teilbar ist. Kommt ein Primfaktor mehrmals vor, so ist er auch ein Faktor für jedes Produkt, das dann

als Summand für die endgültige Summe eingesetzt wird. Wenn wir nun den größten gemeinsamen Teiler von a und b suchen, können wir der
Reihe nach alle nicht gemeinsamen Faktoren fortsubtrahieren, ohne diese zu kennen. Wenn wir annehmen, a ist größer als b, können
wir uns a als Summe von b und einer Unbekannten
x vorstellen:

```
a = x + b
```

a enthält also alle gemeinsamen Primfaktoren von x und
b. Da alle drei Variablen die gleichen gemeinsamen Teiler haben,
können wir a verwerfen und mit x weiterrechnen. Wir
ersetzen a durch x = a - b. Diesen Schritt führen wir
fort, bis a kleiner als b ist. Danach tauschen wir die
beiden Variablen aus. Irgendwann gilt a = 2 * b und
x = b. Dann ist x unmittelbar der größte gemeinsame
Teiler der beiden ursprünglichen Werte, da alle restlichen Teiler durch die Subtraktionen ausgeschlossen wurden.

Wie oben schon erwähnt, lässt sich anhand des Euklidschen Algorithmus sehr einfach feststellen, ob zwei Zahlen *relativ prim*
(teilerfremd) sind. Ist dies der Fall, so ist der größte gemeinsame
Teiler immer 1. Die beiden Eingabewerte haben somit keine Primfaktoren gemeinsam. Dies kann zum Faktorisieren größerer Zahlen eingesetzt werden, indem man einfach wiederholt den größten gemeinsamen Teiler der zu faktorisierenden Zahl und einer Zufallszahl sucht. Das ist jedoch sehr ineffizient.

Die Menge Z[n] hat übrigens eine sehr interessante Eigenschaft.
Diese soll wieder am Beispiel einer Uhr verdeutlicht werden. Wir setzen den Stundenzeiger auf 0 Uhr. Jetzt fahren wir wiederholt 3 Stunden fort. Wir kommen also auf 3 Uhr, dann auf 6 Uhr, dann 9 Uhr und schließlich wieder 0 Uhr. Damit beginnt die Sequenz von vorne. Probieren wir das gleiche mit der Zahl 7. Wir setzen den Zeiger auf 0 Uhr und fahren 7 Stunden fort. Dann haben wir erst 7 Uhr, dann 2 Uhr, dann 9 Uhr, dann 4 Uhr usw. Sobald wir wieder bei 0 Uhr angekommen sind, stellen wir fest, dass wir jede Stunde genau einmal angefahren haben, bevor sich die Sequenz wiederholt. Warum ist das mit 7 passiert, mit 3 aber nicht? Wir merken, dass 7 *relativ prim*
(teilerfremd) zu 12 ist, dass 7 also keine Primfaktoren mit 12
gemeinsam hat. Der größte gemeinsame Teiler von 7 und 12 ist also 1. Da 3

nicht relativ prim zu 12 ist, teilt 3 die 12 in genau 4 Teile auf: 0, 3, 6 und 9. Da 12 schon wieder 0 ist, treffen Vielfache von 3 wieder die 0, nachdem sie alle Teile durchlaufen sind. Damit gilt folgender Satz: die Periode von f(x) = x * f (mod n) ist genau n / gcd(n, f). Für relativ prime Zahlen ist sie also n. Damit kommen wir mit f(x) = x * 3 (mod 12) nur auf die Ergebnisse 0, 3, 6 und 9. Die anderen Elemente von Z[12] treffen wir mit dieser Funktion nicht. Mit f(x) = x * 7 (mod 12) treffen wir jedoch jedes Element und können jede beliebige Zahl in Z[12] als Ergebnis erhalten. Das beweist auch, dass x kein multiplikatives inverses Element in Z[n] besitzt, wenn es nicht relativ prim zu n ist. Es gibt dann nämlich kein Element, das mit x multipliziert 1 ergibt. Mit Vielfachen von x können wir die 1 nicht treffen.

Mit Hilfe des Euklidschen Algorithmus lässt sich auch das kleinste gemeinsame Vielfache zweier Zahlen a und b ermitteln. Im Kapitel über Primzahlen und Teilbarkeit haben wir gelernt, dass das kgV die Kombination aller Primfaktoren beider Zahlen ist, wobei von den gemeinsamen Faktoren nur die höchste Potenz verwendet wird. Der größte gemeinsame Teiler ist die Kombination aller *gemeinsamen* Primfaktoren, wobei immer nur die kleinste Potenz verwendet wird. Somit ist das kgV die Kombination *aller* Primfaktoren, abzüglich der gemeinsamen Primfaktoren. Wir können das kgV (im Folgenden *lcm* (least common multiplicate) genannt) dadurch berechnen, dass wir alle Primfaktoren kombinieren, also a mit b multiplizieren und anschließend durch den größten gemeinsamen Teiler teilen:

```
lcm(a, b) = (a * b) / gcd(a, b)
```

Hier wird eine weitere interessante Eigenschaft ersichtlich. Sind die beiden Eingabewerte relativ prim, der größte gemeinsame Teiler also 1, so ist das kleinste gemeinsame Vielfache das Produkt beider Zahlen.

Die Eulersche phi-Funktion I - Defintion

Die Eulersche phi-Funktion, die schlichtweg als griechischer Buchstabe *phi* geschrieben wird, liefert für jede Zahl x in der Menge N die Anzahl der Elemente in Z[x], die relativ prim zu x sind. Für Primzahlen ist sie definiert als phi(x) = x - 1. Da jede Zahl kleiner als x, außer 0, relativ prim zu x ist, wenn x selbst eine Primzahl ist, ist diese Funktion sehr einfach zu überprüfen: phi(7) = 7 - 1 = 6. Die Zahlen 1 bis 6 sind alle relativ prim zu 7. Die 0 hingegen ist nicht relativ prim zu 7, denn 0 lässt sich durch jede Zahl, also auch 7, restlos teilen. Dies kann man ebenfalls leicht überprüfen: gcd(7, 0) = 7. Nur Zahlen, die 7 als Primfaktor haben, sind nicht relativ prim zu 7, also alle Vielfache von 7. Es kann kein Vielfaches von 7 kleiner als 7 geben. Somit sind alle Zahlen kleiner als 7 relativ prim zu 7, außer 0.

Auch für Produkte zweier **unterschiedlicher** Primzahlen p und q ist diese Funktion simpel definiert: phi(x) = (p - 1) * (q - 1). Um dies zu überprüfen, wählen wir p = 5 und q = 7. Das Produkt ist 35. Wir rechnen:

```
35 = 5 * 7

  phi(35)
= (5 - 1) * (7 - 1)
= 4 * 6
= 24
```

24 Zahlen aus Z[35] sind relativ prim zu 35. Das überprüfen wir, indem wir eine Liste aufstellen, die relativ primen Zahlen hervorgehoben:

```
Z[35] = {
   0,  1,  2,  3,  4,  // 4 Elemente in dieser Zeile
   5,  6,  7,  8,  9,  // 3 Elemente in dieser Zeile
  10, 11, 12, 13, 14,  // 3 Elemente in dieser Zeile
  15, 16, 17, 18, 19,  // 4 Elemente in dieser Zeile
  20, 21, 22, 23, 24,  // 3 Elemente in dieser Zeile
  25, 26, 27, 28, 29,  // 3 Elemente in dieser Zeile
  30, 31, 32, 33, 34   // 4 Elemente in dieser Zeile
}                      // = insgesamt 24 Elemente
```

Damit ist diese Funktion das Produkt aller Primfaktoren des Eingabewertes

minus 1. Dies funktioniert jedoch nur dann, wenn sich *alle* Primfaktoren voneinander unterscheiden. Für den Eingabewert 5 * 11 * 17 = 935 gilt daher:

```
  phi(935)
= (5 - 1) * (11 - 1) * (17 - 1)
= 4 * 10 * 16
= 640
```

Die allgemeine Defintion der Funktion für eine Zahl x mit den Potenzen ihrer Primfaktoren p[1]^e[1] bis p[n]^e[n] lautet:

```
phi(x) = (p[1] - 1) * p[1]^(e[1] - 1) * ... * (p[n] - 1) * p[n]^(e[n] - 1)
```

Beispiel:

```
x = 2^5 * 3 * 7^3 = 32928

  phi(x)
= ((2 - 1) * 2^(5 - 1)) * (3 - 1) * ((7 - 1) * 7^(3 - 1))
= (1 * 2^4) * (2) * (6 * 7^2)
= 16 * 2 * 294
= 9408
```

Die Anzahl der Zahlen, die *nicht* relativ prim zu x in Z[x] sind, ist x - phi(x). Damit möchten wir die Funktion beweisen. Als Beispiel wählen wir eine Zahl n mit zwei Primfaktoren p und q und berechnen:

```
  n - phi(n)
= n - (p - 1) * (q - 1)           // Ausmultiplizieren
= n - (p * (q - 1) - 1 * (q - 1)) // Klammern auflösen
= n - (p * q - p - q + 1)         // Klammern auflösen
= n - p * q + p + q - 1           // Weil n = p * q:
= p + q - 1
```

Damit befinden sich p + q - 1 Zahlen in Z[n], die nicht relativ prim zu n sind. Es gibt p - 1 Vielfache von q in Z[n], denn p * q ist schon wieder

n. Andersrum gibt es q - 1 Vielfache von p in Z[n], da q * p schon wieder n ist. Es gibt also schon mal p + q - 2 Zahlen in Z[n], die nicht relativ prim zu n sind. Da die 0 auch nicht relativ prim zu n ist, kommt noch diese Zahl hinzu und der Satz ist bewiesen: p + q - 2 + 1 = p + q - 1.

Es zeigt sich deutlich, dass wir die Primfaktoren von x brauchen, um phi(x) zu berechnen. Genau diese Tatsache macht sich RSA zunutze. Mehr dazu später.

Anhand dieser Funktion lässt sich der folgende Satz, der im Kapitel über Primzahlen und Teilbarkeit vorgestellt wurde, beweisen:

```
x^e mod n = (x mod n)^(e mod phi(n)) mod n
```

Dass wir x auf x mod n reduzieren können, lässt sich ähnlich wie die anderen beiden modulo-Sätze beweisen. Wir teilen x in eine Summe von xr = x mod n und xq = x - xr auf. Wir können also sagen:

```
xr = x mod n
xq = x - xr

  x^e mod n
= (xq + xr)^e mod n
```

Wir multiplizieren wiederholt xq + xr mit sich selbst. Hier können wir das Distributivgesetz der Multiplikation anwenden, um xq zu eliminieren:

```
  (xq + xr) * (xq + xr)
= (xq * xq) + (xq * xr) + (xr * xq) + (xr * xr)
```

Auch hier können wir, wie bereits beschrieben, alle Vielfachen von xq aus der Addition streichen und es verbleibt:

```
  (xq * xq) + (xq * xr) + (xr * xq) + (xr * xr)
= xr * xr
```

Das lässt sich auf alle weiteren Multiplikationen fortführen. Der Beweis dafür, dass wir e auf e mod phi(n) zurückführen können, ist jedoch etwas kniffeliger.

TO DO: Beweis des Exponentiationsgesetzes fertigstellen.

Die Falltürfunktion

Mathematische Funktionen haben genau eine Aufgabe. Sie geben einen Wert zurück. Für jede umkehrbare Funktion lässt sich eine Umkehrfunktion schreiben. Folgendes Beispiel soll dies verdeutlichen:

```
// x ist Element von R
f(x) = 13 * x + 7
```

Um diese Funktion umzukehren, müssen wir nur in der richtigen Reihenfolge alle Operationen rückgängig machen. Also subtrahieren wir erst 7 und teilen dann durch 13. Die resultierende *Umkehrfunktion* sieht also so aus:

```
// x ist Element von R
g(x) = (x - 7) / 13
```

Für jedes x (aus R) gilt dann x = g(f(x)).
Die beiden aufgeführten Funktionen sind gegenseitig invers. Somit gilt auch x = f(g(x)). Dies ist jedoch keine Voraussetzung.

Man stelle sich nun vor, man hat eine Funktion vor sich, die auf jeden Fall eindeutig umkehrbar ist. Es ist auch bestätigt, dass es die Umkehrfunktion dazu schon gibt. Aus diesem Wissen soll man nun die Umkehrfunktion schreiben. Das klingt alles problemlos. Aber sobald auch noch die kleine Einschränkung hinzukommt, dass man die Umkehrung dieser Funktion in weniger als 15 Milliarden Jahren fertiggestellt haben muss, ändert sich die Lage dramatisch.

Hierbei handelt es sich um das Falltürprinzip. Es gibt bestimmte Funktionen, die schnell definiert werden können, deren Umkehrung aber so aufwändig ist, dass sich diese nicht lohnt durchzuführen. Hier wird klar, dass solche Funktionen vor allem im kryptographischen Bereich ihre Anwendung finden. Dabei hängt die Umkehrung von einem *Geheimnis* ab, welches nur der Autor der Funktion kennt. Ohne dieses Geheimnis kann die Funktion nicht umgekehrt werden.

Die Eulersche phi-Funktion II - Eulers Theorie

Euler ist damals auf eine sehr interessante Eigenschaft der endlichen Menge gestoßen. Solange x und n teilerfremd sind
(gcd(x, n) = 1), gilt:

```
x^phi(n) = 1 (mod n)
```

Dies beweist auch, dass x^(n - 2) (mod n) das multiplikative Inverse zu x in der Menge Z[n] ist, sofern n
prim ist, denn:

```
// Wenn n prim ist, gilt:
phi(n) = n - 1

// damit gilt:
x^(n - 1) = 1 (mod n)
```

```
// Wir legen fest:
1/x = x^(n - 2) (mod n)

// und wissen:
x * 1/x = 1 (mod n)

// Das kombinieren wir:
x * x^(n - 2) = 1 (mod n)

// denn:
x * x^(n - 2) = x^(n - 1) (mod n)

// und:
x^(n - 1) = 1 (mod n)

// weil:
x^phi(n) = 1 (mod n)
```

Hier handelt es sich bei 1/x wieder nicht um einen echten Bruch, sondern um die Darstellung des multiplikativen Inversen zu x.

Um diesen Satz zu erklären, entnehmen wir einerseits eine wichtige Regel der Exponentiation: x^0 = 1. Danach sehen wir uns folgende Regel der modulo-Operation an:

```
a^b mod n = (a mod n)^(b mod phi(n)) mod n
```

Ist b also kongruent zu 0 in Z[phi(n)], ist das Ergebnis 1. Aber nur, wenn a und n teilerfremd sind. Um verständlich machen zu können, was genau daran so interessant sein soll, müssen wir diesen Satz ein wenig ausbauen:

```
x^phi(n) * x^phi(n) = 1 * 1 (mod n)
```

Es stellt sich heraus, dass wir dies beliebig oft wiederholen können, das Ergebnis jedoch immer 1 bleibt. Also verallgemeinern wir diesen Satz:

```
x^(t * phi(n)) = 1 (mod n)
```

Dieser Satz kommt der obigen modulo-Regel etwas näher. Natürlich verlassen wir uns nicht blind darauf und überprüfen das:

```
n = 41 * 67 = 2747
phi(n) = 40 * 66 = 2640

20^2640 = 1 (mod 2747)
21^2640 = 1 (mod 2747)
22^2640 = 1 (mod 2747)
23^2640 = 1 (mod 2747)

// 574 ist nicht relativ prim zu 2747:
574^2640 = 738 (mod 2747)
```

Tatsächlich. Aber dieser Satz hat in der Form noch überhaupt keinen Nutzen für uns. Also sehen wir doch mal, was wir aus dem Satz machen können:

```
// Hier wieder unsere Ausgangskongruenz
x^(t * phi(n)) = 1 (mod n)

// Multiplizieren wir diese Kongruenz doch mal mit x:
x^(t * phi(n)) * x = 1 * x (mod n)
x^(t * phi(n) + 1) = x (mod n)
```

Was haben wir nun? Wir nehmen irgendeinen Wert x, führen eine modulare Exponentiation mit einer bestimmten Zahl durch und als Ergebnis erhalten wir x. Überraschenderweise entfällt plötzlich die Voraussetzung, dass gcd(x, n) = 1 gelten muss, denn der Exponent ist nicht mehr Vielfaches von phi(n). Er ist also nicht mehr kongruent zu 0 in Z[phi(n)]. Falls das noch nicht interessant klingt, sehen wir uns doch mal den Exponenten genauer an:

```
// Unser Exponent
k = t * phi(n) + 1

// Oder als Kongruenz ausgedrückt (Erinnerung: y * n = 0 (mod n), y beliebig):
k = 1 + t * phi(n) (mod phi(n))
k = 1 + 0 (mod phi(n))
k = 1 (mod phi(n))

// Damit der etwas simplifizierte Satz:
k = 1 (mod phi(n))
```

```
x^k = x (mod n)
```

Was genau tun wir hier? Die Lage hat sich nicht geändert. Wir haben eine Zahl x und führen eine modulare Exponentiation durch. Aber halt! Der Exponent kann jede beliebige Zahl kongruenz zu 1 in Z[phi(n)] sein. Unser Ergebnis bleibt weiterhin x. Das interessante hierbei ist, dass der Exponent kongruent zu 1 in der Menge Z[phi(n)] und nicht Z[n] sein muss. Sehen wir doch mal, ob wir daraus etwas machen können.

Wir erinnern uns, dass 1 das neutrale Element der Multiplikation in Z[n] (n beliebig) ist. Somit auch in Z[phi(n)]. Und weiter erinnern wir uns, dass es zu jeder Zahl e in Z[n], die relativ prim zu n ist, ein eindeutiges multiplikatives Inverses d gibt, für das gilt: e * d = 1 (mod n). Also nehmen wir ein beliebiges e, das relativ prim zu phi(n) ist und berechnen das multiplikative Inverse zu e in Z[phi(n)]. Damit können wir unseren Exponenten ein klein wenig abändern:

```
e * d = 1 (mod phi(n))
k = 1 (mod phi(n))
// Das heißt:
k = e * d (mod phi(n))

// Das wiederum heißt:
x^k = x (mod n)

// oder:
x^(e * d) = x (mod n)
```

In diesen Kongruenzen liegt der Schlüssel zum RSA-Verfahren, denn die Kongruenz x^(e * d) = x (mod n) können wir noch einmal zerlegen:

```
e * d = 1 (mod phi(n))
x^(e * d) = x (mod n)

// Nach den Regeln der Potenzrechnung:
(x^e)^d = x (mod n)

// oder:
```

```
x^e = y (mod n)
y^d = x (mod n)
```

Nach all den Umformungen und Einbeziehungen aller möglichen Sätze haben wir folgenden Satz:

```
e * d = 1 (mod phi(n))
x^e = y (mod n)
y^d = x (mod n)
```

Das RSA-Verfahren

Was wir im letzten Kapitel gelernt haben, können wir nun anwenden. Aber zunächst sehen wir uns das nochmal genauer an. Wir haben zwei Werte e und d, die in der Menge Z[phi(n)] multiplikativ invers zueinander sind. Führen wir in Z[n] (!) eine modulare Exponentiation mit Basis x und Exponent e durch, so erhalten wir eine andere Zahl y. Aus der Kenntnis von y lassen sich weder x, noch d, noch phi(n) rekonstruieren, denn y könnte jede beliebige Zahl in Z[n] sein. Nehmen wir nun diese Zahl y und führen in Z[n] eine modulare Exponentiation mit dem Exponenten d durch, so erhalten wir wieder x.

Wie funktioniert RSA nun? Wir wählen eine Zahl n, aus der wir sehr einfach phi(n) berechnen können. Optimal ist das Produkt zweier unterschiedlicher Primzahlen. Wir wählen also zwei unterschiedliche Primzahlen p und q und bilden das Produkt n = p * q.
Dann berechnen wir phi(n) = (p - 1) * (q - 1). Die Primzahlen brauchen wir ab jetzt nicht mehr und vergessen sie. Jetzt wählen wir eine Zahl e, die relativ prim zu phi(n) ist, für die also gilt gcd(e, phi(n)) = 1. Das garantiert, dass die Zahl ein eindeutiges multiplikatives Inverses in Z[phi(n)] besitzt. Als letzten Schritt suchen wir ein d, das die Kongruenz e * d = 1 (mod phi(n)) erfüllt. Wir suchen also genau das multiplikative Inverse zu e in der Menge Z[phi(n)]. Ab hier brauchen wir auch phi(n) nicht mehr. Wir legen fest, dass

e der öffentliche Exponent und d der private Exponent ist. Wenn m unser Klartext ist, so können wir m folgendermaßen verschlüsseln:

```
c = m^e (mod n)
```

c ist dann der Chiffretext. Nur mit der Kenntnis von d können wir den Chiffretext c folgendermaßen wieder in den Klartext überführen:

```
m = c^d (mod n)
```

Als kleines Beispiel erstellen wir erst einmal einen RSA-Schlüssel. Wir suchen uns zwei zufällige Primzahlen p und q aus und berechnen n sowie phi(n):

```
p = 11
q = 17
n = p * q = 187
phi(n) = (p - 1) * (q - 1) = 160
```

Nun suchen wir uns ein e, das relativ prim zu phi(n) = 160 ist. e = 7 erfüllt diesen Zweck. Zuletzt suchen wir ein d, das die Kongruenz e * d = 1 (mod phi(n)) erfüllt. Wir probieren d = 23 und testen:

```
e * d = 1 (mod phi(n))
7 * 23 = 161 = 1 (mod 160)
```

Damit ist unser Schlüssel fertiggestellt. Nun haben wir einen Klartext **52 73 121 173 65**. Diesen möchten wir verschlüsseln. Also verschlüsseln wir jede einzelne Komponente:

```
 52^7 =    1028071702528 =  35 (mod 187)
 73^7 =   11047398519097 =  61 (mod 187)
121^7 =  379749833583241 =  77 (mod 187)
173^7 = 4637914326451397 =  79 (mod 187)
 65^7 =    4902227890625 = 142 (mod 187)
```

Aus dem Klartext **52 73 121 173 65** haben wir nun den Chiffretext **35 61 77 79 142** gemacht. Diesen können wir nun mit dem Exponenten d wieder in die ursprüngliche Form bringen:

```
 35^23 =                 32626089263058801106214523315429687 5 =  52 (mod 187)
 61^23 =           1155011225795843885921385794579057586481 81 =  73 (mod 187)
 77^23 =         24506804088319785713436649354236658982956133 = 121 (mod 187)
 79^23 =         44200084506410989454600861862443675119534639 = 173 (mod 187)
142^23 = 3181499950464199729691617712190281936939724360908 8 =  65 (mod 187)
```

Es muss dringend erwähnt werden, dass die komponentenweise Verschlüsselung nur zur Demonstration so durchgeführt wurde. Man sollte unbedingt die Komponenten eines Klartextes kombinieren und daraus die größtmögliche Einheit bilden. Wie und warum wird weiter unten im Kapitel über die Tücken von RSA beschrieben.

Was macht RSA sicher?

Dazu schauen wir uns noch einmal den Schlüssel an. Nachdem wir zwei Primzahlen ausgewählt und n und phi(n) berechnet haben, vergessen wir die Primzahlen. Dann berechnen wir e und d und vergessen danach auch phi(n). Zum Schluss bleiben nur noch die Zahlen n als Modul, e als öffentlicher Exponent und d als privater Exponent übrig. Den öffentlichen Schlüssel bilden damit die Zahlen e und n. Jeder kann mit diesen beiden Werten eine Nachricht verschlüsseln. Ohne das entsprechende d kann sie jedoch nicht entschlüsselt werden.

Schlüpfen wir nun in die Rolle des Angreifers. Der Angreifer wird versuchen, d zu ermitteln. Er weiß:

```
e * d = 1 (mod phi(n))
```

Um d also berechnen zu können, muss er aus n einfach nur phi(n) berechnen. Er weiß, dass n aus genau zwei Primzahlen p und q besteht, mit denen er phi(n) berechnen kann:

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

Was muss er also tun? Er muss die Primzahlen p und q ermitteln. Das heißt, er muss n faktorisieren. Setzen wir das Beispiel im letzten Kapitel als Angreifer fort. Wir haben n = 187 und e = 7. 187 faktorisieren wir und erhalten damit die beiden Primfaktoren 11 und 17. Daraus können wir phi(n) berechnen. Sobald wir phi(n) haben, haben wir auch d, indem wir die obige Kongruenz nach d auflösen.

Was macht RSA also nun sicher? Es war sehr einfach, 187 zu faktorisieren. Was aber nun, wenn wir größere Primfaktoren wählen? Beispielsweise:

```
n = 710300483573436168165918436 27
e = 538475231
```

Also faktorisieren wir ganz einfach n. Ganz einfach? Hier stellt sich bald heraus, dass wir genau vor dem Problem stehen, das RSA sicher macht. Wir können n nur nach einem Rateverfahren faktorisieren. Es bleibt uns nichts anderes übrig, als n durch alle möglichen Primzahlen zu dividieren und zu sehen, ob eine der Probezahlen restlos in n geht. Das erklärt auch, warum wir nicht einfach eine große Primzahl als n genommen haben. Dann hätte der Angreifer phi(n) direkt, denn er müsste nur noch n - 1 berechnen. Durch ein kleines Computerprogramm kann n jedoch immer noch in relativ kurzer Zeit faktorisiert werden. Wir erhalten:

```
p = 267958487152523
q = 265078554190049
phi(n) = (p - 1) * (q - 1) = 710300483573430837795505010 56
```

Aus dieser Erkenntnis heraus können wir d berechnen. Mit dem *erweiterten Euklidschen Algorithmus* ermitteln wir:

```
e = 538475231
d = 61249433402649775486339595551
e * d = 32981302800110953867104851058771297281 = 1 (mod phi(n))
```

Was ist aber nun, wenn die Primfaktoren mehrere Hundert Dezimalstellen lang sind? Nach mehreren Tausend Jahren Forschung in der Zahlentheorie hat noch niemand ein Verfahren gefunden, mit dem sich jede beliebige Zahl auf direktem Wege faktorisieren lässt. Auch Primzahlen selbst scheinen relativ systemlos verteilt zu sein. Bisher konnte aber auch niemand beweisen, dass so ein Verfahren nicht existiert. Falls jemals jemand ein derartiges Verfahren finden sollte, ist RSA nicht mehr sicher und kann nicht mehr für kryptographische Zwecke eingesetzt werden. Aber eben angesichts der jahrtausendelangen Forschung im Bereich der Zahlentheorie, erscheint ein solches Verfahren sehr unwahrscheinlich.

Hier muss bedacht werden, dass es inzwischen schon sehr ausgeprägte Suchverfahren gibt. Im Moment (Oktober 2004) ist das bisher schnellste Verfahren zur Faktorisierung von großen Zahlen der *Zahlkörpersieb* (engl. number field sieve). Doch auch dieses Verfahren ist nur ein Annäherungsverfahren. Man kann sich vor einer erfolgreichen Modulfaktorisierung schützen, indem man die Primzahlen groß genug wählt. Dabei muss man mehrere Hundert Stellen in Betracht ziehen. Im Moment liegt die Empfehlung bei 1024 Bits (Zahlen von 0 bis 2^1024 - 1) für das Modul, also durchschnittlich 512 Bits für die beiden Primfaktoren. Man kann sich selbstverständlich größere Schlüssel generieren, doch irgendwann wird die Schlüsselgenerierung selbst unpraktisch, da die Suche nach den beiden Primzahlen mit steigender Anzahl an Bits zunehmend länger dauert. Das liegt daran, dass Primzahlen mit steigendem Wert immer seltener werden und die modulare Arithmetik immer langsamer. Außerdem muss darauf geachtet werden, dass viele *Keyserver* und *CAs* (certificate authorities) keine unbegrenzte Schlüssellänge zulassen. Schlüssel mit mehr als 4096 Bits werden nicht empfohlen, falls diese auf einem Keyserver oder ähnlichem abgelegt werden sollen, es sei denn, dem Anwender ist ein Keyserver oder CA bekannt, der die entsprechende Länge zulässt.

Es gibt noch einen zweiten Weg, die Verschlüsselung umzukehren. Wenn wir annehmen, die Chiffre würde nicht in der endlichen Menge durchgeführt werden, so hätten wir folgendes Schema:

```
c = m^e
```

Hier hätte der Angreifer ein leichtes Spiel. Er müsste nur die e'te Wurzel aus c ziehen. Dies ist durchführbar. Aber stattdessen haben wir folgendes Schema:

```
c = m^e (mod n)
```

Das ändert die Lage dramatisch, denn das Ergebnis könnte jede beliebige Zahl in Z[n] sein. Er kann also nicht einfach die Wurzel ziehen. Doch auch das Wurzelziehen ist in Z[n] möglich. Dieses ist aber mindestens eben so schwer wie Faktorisierung von n. Er kann systematisch m^e (mod n) mit allen möglichen Basen m ausprobieren und sehen, ob das Ergebnis c ist. Hierbei handelt es sich um das Problem der modularen Wurzel.

Es zeigt sich, dass RSA eine *Falltürfunktion* ist. Der Angreifer kann problemlos c = m^e (mod n) berechnen. Die Umkehrung dieses Terms ist jedoch eine immens schwierige Aufgabe. Weiter oben wurde erwähnt, dass die Umkehrung einer solchen Falltürfunktion von einem Geheimnis abhängt, das nur dem Autor der Funktion bekannt ist. Dieses Geheimnis ist hier der Wert von phi(n). Entweder, der Angreifer faktorisiert n, oder er sucht nach einem d, das c wieder in das ursprüngliche m überführt (dazu benötigt er phi(n)), oder er versucht, die e'te Wurzel aus c zu ziehen (hier kann er nur raten). Die beiden letzten Verfahren wären kein Problem, wenn wir nicht in der endlichen Menge rechnen würden. Die Faktorisierung ist ohnehin ein Problem.

Tücken des RSA-Verfahrens

RSA ist ein Verfahren, das widersprüchlicher nicht sein kann. Nicht nur, dass die Sicherheit von RSA unbestätigt ist und wahrscheinlich auch sehr lange so bleiben wird. RSA war lange Zeit ein Verfahren, dass sich auf

Schätzungen verlassen musste. Weiters verbergen sich eine Reihe Tücken in diesem Verfahren, die hier besprochen werden sollen. Diese Liste erhebt keinen Anspruch auf Vollständigkeit. Im Grunde ist sie mehr als unvollständig. Der Leser wird hier auf externe Literatur verwiesen, aber die wichtigsten Fehler, die von Programmierern gemacht werden, sollen hier besprochen werden.

1. Kleine Exponenten

Eine früher oft übersehene Sicherheitslücke war die Wahl kleiner öffentlicher Exponenten. Nehmen wir als Beispiel folgenden öffentlichen Schlüssel an:

```
n = 37762589
e = 7
```

Wir schlüpfen wieder in die Rolle des Angreifers und sehen, dass jemand folgende Nachricht an den Besitzer des Schlüssels schickt:

```
c = 35831808
```

Rein probehalber ziehen wir aus 35831808 die siebte Wurzel. Erstaunt und voller Freude stellen wir fest, dass das Ergebnis eine Ganzzahl ist: 12. Das heißt, 12^7 ergibt 35831808 und 35831808 ist ein gültiges Element von Z[37762589]. Die Originalnachricht muss also 12 sein. Das probieren wir aus:

```
12^7 = 35831808 (mod 37762589)
```

Was hat der Absender nun falsch gemacht? Genau genommen gar nichts. Dies ist eine kritische Tücke im RSA-Verfahren. Je kleiner der Exponent e ist, desto größer ist die Wahrscheinlichkeit, dass das Ergebnis von x^e innerhalb von Z[n] ist. Tritt der Fall ein, dass x^e in Z[n] ist, dass also x^e mod n = x^e ist, braucht man nur die e'te Wurzel aus x^e (mod n) zu ziehen und erhält x. Ob das funktioniert hat, erkennt man an der Wurzel. Ist sie eine Ganzzahl, so hält man die Nachricht im Klartext in den Händen.

Wir sollten also keinen zu kleinen öffentlichen Exponenten wählen. Auch sollten wir keinen zu großen wählen, denn sonst herrscht die Gefahr, dass der entsprechende private Schlüssel zu klein wird und dadurch per *Bruteforce* (ausprobieren) ermittelbar ist. Eine gute Regel ist es, einen Exponenten zu wählen, sodass 2^e außerhalb von Z[n] ist. So garantieren wir, dass jede Zahl mindestens einmal *umklappt* und dadurch kein Wurzelziehen möglich ist.

2. Trivialer Klartext

Auch wird oft übersehen, dass 0^x immer 0 und 1^x immer 1 sind. Lautet der Chiffretext also c = 0, so lautet garantiert auch der Klartext m = 0, denn m^e = 0^e = 0 (mod n) für jedes beliebige e. Das gleiche gilt für c = 1.
Problematisch ist auch, wenn wir zwei Chiffretexte c[1] und c[2] haben, die gleich sind (c[1] = c[2]). Wir als Angreifer wissen dann nämlich, dass die dazugehörigen Klartexte m[1] und m[2] auch gleich sind. Dies ist vor allem dann gefährlich, wenn wir Buchstabe für Buchstabe verschlüsseln, was ohnehin nicht gemacht werden sollte.

Um diese beiden Probleme zu lösen, sollten die untersten paar Bits des Klartextes immer irgendwelche Zufallszahlen über 1 sein. Es sollten also nur die höheren Bits für die eigentliche Nachricht genutzt werden. So verhindert man, dass der Klartext m jemals 0 oder 1 wird. Das erschwert dann auch die Kryptanalyse. Da die untersten paar Bits zufällig gewählt sind, liefert der gleiche Klartext immer einen unterschiedlichen Chiffretext. Je mehr Bits zufällig sind, desto mehr Chiffretextvarianten eines Klartextes gibt es. Sind 8 Bits zufällig, so hat jeder Klartext genau 2^8 = 256 verschiedene Chiffretextdarstellungen. Dies ist garantiert, da x^e (mod n) jedes x unterschiedlich abbildet, solange e relativ prim zu phi(n) ist und das ist Voraussetzung für die korrekte Funktionalität von RSA.

Es sollte auch immer die größtmögliche Einheit zum Verschlüsseln gewählt werden. Nehmen wir folgenden Schlüssel an:

```
n = 3382087550195091160127
e = 6341
d = 3244478562864920811989
```

Wir haben nun den Klartext **1 52 73 12 63 12 68**, den wir wie im letzten Kapitel beschrieben verschlüsseln:

```
1.  1^e = 1 (mod n)
2. 52^e = 2111984357415861106407 (mod n)
3. 73^e = 1274135684506913336235 (mod n)
4. 12^e = 2889964449295186937660 (mod n)
5. 63^e = 721727110139700243420 (mod n)
6. 12^e = 2889964449295186937660 (mod n)
7. 68^e = 2130683150637670776795 (mod n)
```

Hier werden die beiden zuletzt beschriebenen Probleme deutlich. Der erste Klartext 1 liefert auch den Chiffretext 1. Der Angreifer weiß also, dass der Klartext auf jeden Fall 1 sein muss. Das nächste Problem, das deutlich wird, ist, dass wir die 12 doppelt verschlüsseln. Der Angreifer weiß also, dass an 4. und 6. Stelle auf jeden Fall der gleiche Klartext steht.

Wie oben beschrieben setzen wir eine Zufallszahl an die ersten paar Stellen. Weiters bilden wir die größtmögliche Einheit. Wir kombinieren die Komponenten des Klartextes also so, dass wir eine Zahl kleiner als n erhalten.
Aus **1 52 73 12 63 12 68** machen wir also eine einzige Zahl: **0152731263126882**. Die untersten beiden Stellen der Zahl sind zufällig gewählt. Wir verschlüsseln diese Zahl:

```
152731263126882^e = 2371099362558603522389 (mod n)
```

Aus dem Ergebnis kann der Angreifer jetzt keine Informationen mehr über den ursprünglichen Wert ermitteln. Jetzt sieht er weder, dass eine Komponente auf jeden Fall 1 sein muss, noch, dass zwei Komponenten gleich sind. Der Empfänger der Nachricht entschlüsselt:

```
2371099362558603522389^d = 152731263126882 (mod n)
```

Diese Zahl **152731263126882** teilt er nun wieder in

zweistellige Komponenten auf: **1 52 73 12 63 12 68 82**. Die letzte Komponente verwirft er, da sie zufällig ist und nicht zur eigentlichen Nachricht gehört.

3. Schlechte Primfaktoren

Die beiden Primfaktoren, aus denen das Modul n gebildet wird, müssen zwar zufällig sein, aber nur begrenzt. Beispielsweise sollten die Primzahlen p und q nicht zu nah beieinander liegen, denn je kleiner der Betrag | p - q |, desto kleiner auch die Beträge | sqrt(n) - p | und | sqrt(n) - q | (sqrt() ist die Quadratwurzelfunktion). Ab einer gewissen Grenze kann man dann die Primfaktoren dadurch ermitteln, dass man von der Quardratwurzel von n beginnend auf- und/oder abwärts sucht. Die Faktoren sollten sich in mindestens 90% der niederwertigen Ziffern unterscheiden. Am besten ist es, wenn sich die höchstwertigen paar Stellen voneinander unterscheiden. Andererseits sollten die Primzahlen nicht zu weit auseinander liegen, da sonst die Gefahr herrscht, dass eine der Primzahlen zu klein wird. Eine gute Regel ist es, zwei Primzahlen zu finden, die die gleiche Anzahl an Hexadezimalstellen haben, deren höchste Stellen sich aber deutlich voneinander unterscheiden. Hier ein Beispiel schlechter Primzahlen:

```
// Zu nah beieinander:
p = 102430499
q = 102430519

// Zu weit auseinander:
p = 845613173
q = 521
```

Und als Kontrast, hier noch einmal ein Beispiel mit guten Primzahlen:

```
p = 635678137
q = 295298623
```

Natürlich sind diese Zahlen allgemein zu klein, aber sie zeigen deutlich, worauf geachtet werden muss.

4a. Falsche RSA-Schlüssel

So simpel RSA auch sein mag. All das funktioniert nicht, solange wir keine Möglichkeit haben, die beiden Primzahlen p und q zu finden. Bisher konnte noch niemand ein System in der Verteilung von Primzahlen entdecken. Wir wissen nur, dass jede Primzahl ungleich zwei ungerade ist, denn sonst wäre die Zahl durch 2 teilbar und damit keine Primzahl mehr. Lange Zeit war das ein großes Problem für die Implementierung des RSA-Verfahrens. Man war darauf angewiesen, die Primzahlen nach einem Rate- und Schätzverfahren zu ermitteln. Es gab nämlich nur Primzahltests, die feststellen konnten, ob eine Zahl *nicht* prim ist. Das Problem ist inzwischen gelöst, aber um zu verdeutlichen, welche Folgen ein falscher RSA-Schlüssel haben kann, müssen wir uns vorübergehend in die Vergangenheit begeben.

Wie schon gesagt, arbeiten wir nach einem Rate- und Schätzverfahren. Wir wählen irgendeine Zahl x und müssen nun überprüfen, ob diese prim ist. Hier sind wir mit einem Jahrtausende alten Problem konfrontiert. Wie können wir ermitteln, ob x prim ist? Wir haben die Möglichkeit einer Garantie, dass x *nicht* prim ist, aber wir können nicht garantieren, *dass* x prim ist. Was passiert also nun, wenn unsere Schätzung fehlschlägt? Hier ein Beispiel mit kleinen Zahlen:

```
// Unsere Schätzung hat ergeben, dass die folgenden Zahlen prim sind:
p = 637243  // Diese Zahl ist prim
q = 387103  // Diese jedoch nicht: 521 * 743

// Also berechnen wir Modul und Anzahl der teilerfremden Zahlen:
n = p * q = 246678677029
phi(n) = (p - 1) * (q - 1) = 246677652684  // Falsch!

// Jetzt wählen wir die Exponenten:
e = 851
d = 123193892351
e * d = 1 (mod phi(n))
```

Damit ist unser falscher Schlüssel fertiggestellt. Das Problem bei diesem Schlüssel ist, dass phi(n) einen falschen Wert hat und damit die folgende Kongruenz nicht mehr aufgeht:

```
x^phi(n) = 1 (mod n)
```

Das können wir testen:

```
10^246677652684 = 100373419417 (mod 246678677029)
11^246677652684 = 155962675279 (mod 246678677029)
12^246677652684 = 204098099770 (mod 246678677029)
13^246677652684 = 228224756993 (mod 246678677029)
14^246677652684 = 158516107980 (mod 246678677029)
// ...
```

An diesem Beispiel sehen wir deutlich, dass unsere gewählten Zahlen nicht korrekt waren. Wir können das auch mit Ver- und Entschlüsselung testen, denn sie wird nicht mehr funktionieren:

```
// Wir verschlüsseln 643:
643^851 = 202623744463 (mod 246678677029)

// Jetzt entschlüsseln wir:
202623744463^123193892351 = 49265256973 (mod 246678677029)
```

Die Verschlüsselung wurde nicht korrekt umgekehrt. Dieses Problem konnten wir nicht lösen, nur eingrenzen. Für jedes p und q (n = p * q) mussten wir einige Tests durchführen. Zuerst einmal testeten wir, ob die Kongruenz x^((p - 1) * (q - 1)) = 1 (mod n) aufgeht, wenn gcd(x, n) = 1 gilt. Wir führten diese Operation einfach mit verschiedenen Basen x, die relativ prim zu n sind, durch. Sobald das Ergebnis für eine Basis x nicht 1 war, wussten wir mit Sicherheit, dass n auf jeden Fall nicht das Produkt zweier Primzahlen ist. Somit war entweder p oder q faktorisierbar und damit unser Ergebnis falsch. In diesem Fall suchten wir ein neues Paar p und q und testeten noch einmal. Je öfter unser Ergebnis 1 blieb, desto unwahrscheinlicher wurde es, dass p und q nicht prim sind. Dies konnte jedoch nicht versichert werden und wir hatten zwei zusätzliche Probleme. Es gibt bestimmte Zahlen, die für jede Basis 1 als Ergebnis liefern. Diese Zahlen sind extrem selten und werden mit der Größe von n zunehmend seltener, jedoch existieren sie und es gibt unendlich viele davon. Diese Zahlen sind analog zu den Carmichael-Zahlen, welche später besprochen werden. Das zweite Problem ist, dass wir mit riesigen Zahlen arbeiten müssen, damit RSA sicher ist. Das heißt, wir konnten nicht alle möglichen Basen x ausprobieren, denn für jede Probe mussten wir einmal den größten gemeinsamen Teiler gcd(x, n) berechnen, um zu sehen, ob x und n relativ prim sind und einmal eine modulare Exponentiation x^((p - 1) * (q - 1)) (mod n) durchführen, um zu sehen, ob die obige Kongruenz aufgeht. Das erfordert immense Rechenzeit und konnte daher nur

auf eine winzige Untergruppe von x aus Z[n] durchgeführt werden. Zuletzt konnte man, um noch einmal sicher zu gehen, mit mehreren Probezahlen die Ver- und Entschlüsselung testen. Schlug dies fehl, so wussten wir auch mit Sicherheit, dass unser Ergebnis falsch war.

Wie bereits gesagt, ist dieses Problem inzwischen gelöst. Es gibt bereits Testverfahren wie das *Miller-Rabin-Verfahren*, die garantieren können, dass eine Zahl prim ist. Wir bleiben jedoch weiterhin in der Vergangenheit. Durch unvorsichtiges Programmieren können auch in der Gegenwart falsche Schlüssel entstehen. Im nächsten Abschnitt wird gezeigt, welche Gefahren in einem falschen Schlüssel stecken.

4b. Konsequenzen falscher Schlüssel

Damit wir Beispiele mit kleineren Zahlen liefern können, versetzen wir uns zurück in eine Zeit, in der es Jahrehunderte gedauert hat, eine Zahl mit 20 Stellen zu faktorisieren, aber nur wenige Minuten, um eine Zahl mit 10 Stellen zu faktorisieren. Falls dies unrealistisch klingt: die Zeit, die man zum Faktorisieren einer Zahl braucht, steigt **exponentiell** mit der Größe der Zahl, nicht etwa linear! Daher ist dieses Beispiel durchaus realistisch. Nehmen wir folgenden falschen Schlüssel:

```
p = 5928463679  // Ist prim.
q = 2168935883  // Ist nicht prim: 34171 * 63473
n = 12858457604445293557
phi(n) = 12858457596347893996  // Falsch!

e = 61
d = 8220981086189637145
e * d = 1 (mod phi(n))
```

Da n nun Produkt aus *drei* Primzahlen ist, wobei zwei davon nur fünfstellig sind, lässt sich diese Zahl wie ein zehnstelliges Produkt aus zwei Primzahlen innerhalb weniger Minuten faktorisieren. phi(n) ist für ein n mit drei Primfaktoren n = p * q * r definiert als phi(n) = (p - 1) * (q - 1) * (r - 1). Also faktorisieren wir n erst einmal und erhalten:

```
p = 5928463679
```

```
q = 34171
r = 63473
n = p * q * r = 12858457604445293557
```

Daraus können wir also nun den tatsächlichen Wert von phi(n) berechnen:

```
phi(n) = (p - 1) * (q - 1) * (r - 1) = 12857878729297446720
```

Wir wissen, dass e = 61 ist und können, da wir nun den richtigen Wert für phi(n) kennen, auch den richtigen Wert für d berechnen:

```
e = 61
d = 2950988560822364821
```

Aber wenn der Schlüssel so oder so falsch ist, wo ist dann das Problem? Dazu stellen wir uns folgendes Szenario vor. Bob generiert diesen falschen Schlüssel und legt ihn auf einem Keyserver ab. Da Bobs Software den Schlüssel nicht überprüft hat, merkt er nichts davon, dass der Schlüssel falsch ist. Nun holen sich Alice und Eve (die Angreiferin) diesen öffentlichen Schlüssel. Alice schickt nun eine verschlüsselte Nachricht an Bob und Eve hört diese Nachricht ab. Da Eve aus dem falschen Schlüssel problemlos einen richtigen Schlüssel machen konnte, wobei sich die Werte für n und e aber nicht ändern, kann sie die Nachricht problemlos entschlüsseln. Bob hingegen kann dies nicht, denn sein Schlüssel ist falsch. Alice berechnet aus m = 253761356 den Chiffretext:

```
c = 253761356^61 = 5441719397908522177 (mod n)
```

Diesen sendet Alice nun an Bob. Eve fängt diese Nachricht ab und entschlüsselt sie mit dem richtigen Wert für d:

```
m = 5441719397908522177^2950988560822364821 = 253761356 (mod n)
```

Bob hingegen, der eigentliche Empfänger der Nachricht, wird ein falsches Ergebnis bekommen, da sein d nicht stimmt:

```
// Falsch:
m = 544171939790852217 7^8220981086189637145 = 6461175203459092287 (mod n)
```

Bob ruft bei Alice an und beschwert sich über den Mist, den Alice ihm geschickt hat. In der Zwischenzeit versendet Eve bereits alle möglichen Nachrichten mit der Signatur von Bob.

Es zeigt sich deutlich, dass falsche Schlüssel zu einer ernsthaften Gefahr werden können. Wer den privaten Schlüssel eines anderen hat, kann problemlos alle verschlüsselten Nachrichten, die an den Besitzer des Schlüssels gerichtet sind, lesen. Noch schlimmer jedoch, kann er Signaturen im Namen des eigentlichen Schlüsselbesitzers erzeugen und damit sogar Verträge abschließen. Dieses Problem kann nur durch vorsichtiges Programmieren und durch korrekte Testverfahren gelöst werden. Wir verlassen an dieser Stelle die Vergangenheit und kehren zur Gegenwart zurück. Man kann den weiter oben erwähnten *Miller-Rabin-Primzahltest* verwenden, denn dieser kann garantieren, dass eine Zahl prim ist.

Anwendung von RSA

Nachdem wir alle wichtigen Bestandteile von RSA kennengelernt haben, möchten wir die Chiffre natürlich auch anwenden. Hier werden einige Anwendungsbeispiele vorgestellt und erklärt.

1. Verschlüsselung

Die Verschlüsselung ist wohl das trivialste Beispiel überhaupt. Wie in einem früheren Kapitel bereits erwähnt wurde, ist die RSA-Chiffre sehr langsam. Aus diesem Grund wird sie fast immer mit einer klassischen symmetrischen Chiffre kombiniert. Zunächst erstellen wir einen RSA-Schlüssel R und veröffentlichen den öffentlichen Teil Rp dieses Schlüssels. Der Absender, der nun einen verschlüsselten Text an uns senden möchte, wählt eine symmetrische Chiffre, z.B. *AES* oder *Blowfish* und erstellt

für diese Chiffre einen symmetrischen Schlüssel K. Er verschlüsselt seinen Text mit dem symmetrischen Verfahren unter Verwendung des Schlüssels K. Danach verschlüsselt er K mit RSA unter Verwendung des Schlüssel Rp. Die Kombination aus symmetrisch verschlüsseltem Text, asymmetrisch verschlüsseltem Schlüssel und eventuell einer Prüfsumme sendet er an uns.

Wir als Empfänger kennen den privaten Teil Rs von R und können somit den Schlüssel K, der per RSA verschlüsselt in der Nachricht enthalten ist, entschlüsseln. Diesen verwenden wir dann für die Entschlüsselung der eigentlichen Nachricht mit der gewählten symmetrischen Chiffre. Die Kombination aus symmetrischer und asymmetrischer Chiffre hat einen gewaltigen Geschwindigkeitsvorteil, vor allem, wenn wir große Nachrichten (z.B. multimediale Nachrichten) verschlüsseln. Wir müssen nur noch einen symmetrischen Schlüssel verschlüsseln und dafür reicht in den meisten Fällen eine einzige RSA-Operation.

2. Signaturen

Auch das Signieren von Nachrichten ist eine wichtige Anwendung von RSA. Diese erlaubt es uns, gleichzeitig die Integrität und die Vertrauenswürdigkeit einer Nachricht weitgehend zu garantieren. Zum Signieren bedienen wir uns einer speziellen Funktion, der *Hash-Funktion*. Eine Hash-Funktion ist eine Funktion, die folgende Eigenschaften hat:

- Der Rückgabewert der Funktion kann nicht auf den Eingabewert zurückgeführt werden.
- Es ist sehr schwierig, zwei Eingabewerte zu finden, die den gleichen Rückgabewert erzeugen.
- Es ist sehr schwierig, für einen bestimmten Rückgabewert einen passenden Eingabewert zu finden.

Eine Hash-Funktion ist also eine Einwegfunktion, deren Rückgabewert nur sehr schwer auf den Eingabewert zurückzuführen ist und deren Rückgabewert schwer zu reproduzieren ist. Der Rückgabewert einer Hash-Funktion wird *hash*, *message digest* oder *Prüfsumme* genannt. Bekannte Hash-Funktionen sind *MD5*, *SHA1* und *RIPE-MD*.
Empfohlen wird die Funktion SHA1, wobei MD5 bisher am weitesten verbreitet ist. Eine intensive Beschäftigung mit Hash-Funktionen sprengt den Rahmen dieser Dokumentation. Der Leser wird auf externe Literatur verwiesen.

Um eine Nachricht zu signieren, benötigen wir an erster Linie natürlich einen RSA-Schlüssel. Dieser kann der selbe sein, der auch zum Verschlüsseln verwendet wurde. Wir bezeichnen ihn als R. Der private Teil des Schlüssels wird als Rs und der öffentliche als Rp bezeichnet. Aus der Nachricht, die signiert werden soll, berechnen wir eine Prüfsumme mit einer bewährten Hash-Funktion wie SHA1. Diese verschlüsseln wir mit Rs und fügen sie zur Nachricht hinzu.

Der Empfänger der Nachricht kann diese nun verifizieren. Da Rp invers zu Rs ist, kann er die Signatur problemlos entschlüsseln. Er erstellt eine Prüfsumme der Nachricht und vergleicht diese mit der übergebenen Prüfsumme. Stimmen die Prüfsummen überein, ist garantiert, dass die Nachricht auf jeden Fall vom Besitzer des Schlüssels R versandt wurde und vollständig intakt ist. Niemand, außer dem Besitzer von R, kann solche Signaturen erstellen, denn nur er kennt Rs.

3. Authentifizierung

Ein weiterer interessanter Anwendungsbereich von RSA ist die Authentifizierung. Statt der normalen, passwortorientierten Authentifizierung verwenden wir den öffentlichen Schlüssel als Authentifizierungsgegenstand. Diese Form der Authentifizierung wird *public key authentication* genannt. Das Prinzip ist sehr simpel und erlaubt unter anderem beidseitige Identitätsverifikation. Der wichtigste Vorteil aber ist die Abwehr der *Man in the Middle - Attacke* (siehe unten). Dazu stellen wir uns eine Kommunikation zwischen Client und Server vor.

Nachdem die Verbindung hergestellt wurde, verlangt der Server den öffentlichen Schlüssel des Clients, den dieser bereitstellt. Gleichzeitig verlangt der Client den öffentlichen Schlüssel des Servers. Der Server überprüft, ob der übergebene öffentliche Schlüssel bekannt ist. Wenn ja, dann ist der Client identifiziert, aber noch nicht authentifiziert. Gleichzeitig prüft der Client, ob der öffentliche Schlüssel des Servers bekannt ist. Wenn ja, ist der Server identifiziert, aber noch nicht authentifiziert. Nach der Initialisierungsphase findet die eigentliche Authentifizierung statt, die allerdings relativ simpel ist und vollständig automatisch abläuft.

Der Client sendet eine zufällige, mit dem öffentlichen Schlüssel des Servers verschlüsselte Nachricht an den Server. Die Nachricht sollte nicht zu kurz sein. Der Server erstellt von dieser Nachricht eine Prüfsumme und sendet diese mit dem öffentlichen Schlüssel des Clients verschlüsselt an den Client zurück. Der Client überprüft diese Prüfsumme. Wenn sie korrekt ist, ist der Server authentifiziert. Gleichzeitig sendet der Server dem Client eine zufällige, mit dem öffentlichen Schlüssel des Clients verschlüsselte Nachricht. Dieser erstellt von der Nachricht eine Prüfsumme und sendet sie mit dem öffentlichen Schlüssel des Servers verschlüsselt an den Server zurück. Stimmt die Prüfsumme, so ist auch der Client authentifiziert. Eine passwortbasierte Authentifizierung ist hier nicht mehr notwendig, denn ein Angreifer braucht unbedingt den privaten Schlüssel des Clients, um dessen Identität vorzutäuschen, bzw. den privaten Schlüssel des Servers, um die Identität des Servers vorzutäuschen, denn er kann sonst die Prüfsummen nicht erstellen.

Die Eulersche phi-Funktion III - Primzahltest

Bereits im vorletzten Kapitel wurde auf ein Problem hingewiesen, das wir bei der Arbeit mit RSA haben. Wir benötigen riesige Primzahlen mit mehreren hundert Dezimalstellen. Hier soll nun ein etwas älterer Primzahltest vorgestellt werden. Dieser dient nur der Veranschaulichung der Eulerschen phi-Funktion. Er sollte nicht verwendet werden, da er nicht garantieren kann, dass eine Zahl prim ist. Weiter oben wurde bereits der *Miller-Rabin-Primzahltest* erwähnt, der hier momentan allerdings noch nicht dokumentiert ist. Das Verfahren, das hier vorgestellt wird, basiert auf Eulers Theorie, die oben genannt wurde:

```
// Wenn x in Z[n] und gcd(x, n) = 1, dann:
x^phi(n) = 1 (mod n)
```

Nehmen wir nun an, n ist prim, dann ist phi(n) definiert als phi(n) = n - 1 und damit gilt der folgende Satz:

```
// Wenn x ungleich 0 in Z[n] und n prim, dann:
x^(n - 1) = 1 (mod n)
```

Dieser Satz besagt, dass x^(n - 1) (mod n) für jedes x ungleich 0 in der Menge Z[n] immer 1 ergibt, sofern n prim ist. Beweis: jede Zahl in Z[n], außer 0, ist relativ prim zu n, sonst wäre n teilbar und damit keine Primzahl. Der Test besteht also darin, x^(n - 1) (mod n) mit verschiedenen Basen x zu berechnen und das Ergebnis mit 1 zu vergleichen. Erhalten wir für irgendeine Zahl ein Ergebnis ungleich 1, so ist n mit Sicherheit keine Primzahl.

Es zeigt sich, dass dieser Test nicht garantieren kann, dass n prim ist. Er kann nur garantieren, dass n *nicht* prim ist. Im Übrigen sagt dieser Test nur aus, *dass* eine Zahl nicht prim ist. Anhand dieses Tests erhalten wir keine Informationen über die Faktoren dieser Zahl.

Wir probieren diesen Test mit verschiedenen Werten für n aus:

```
// n = 3743; diese Zahl ist nicht prim:
590^3742 = 1 (mod 3743)
// aber:
591^3742 = 3349 (mod 3743)
592^3742 = 986 (mod 3743)
593^3742 = 3483 (mod 3743)
594^3742 = 92 (mod 3743)

// n = 5227; diese Zahl hingegen ist prim:
590^5226 = 1 (mod 5227)
```

```
591^5226 = 1 (mod 5227)
592^5226 = 1 (mod 5227)
593^5226 = 1 (mod 5227)
594^5226 = 1 (mod 5227)
```

Je mehr Einsen wir bekommen, desto unwahrscheinlicher wird es, dass n nicht prim ist. In Z[3743] gibt es beispielsweise nur vier Zahlen, die das Ergebnis 1 liefern: 1, 590, 3153 und 3742. Da die 1 relativ trivial ist, brauchen wir sie gar nicht zu testen, denn 1^x ergibt immer 1.

Unwahrscheinlicher, aber leider nicht unmöglich. Es gibt bestimmte Zahlen, die für jede Basis x als Ergebnis 1 liefern, aber nicht prim sind.
Diese Zahlen werden *Carmichael-Zahlen* genannt und sind extrem selten. Viel seltener als Primzahlen im höheren Bereich. Jedoch existieren sie und es gibt unendlich viele solcher Zahlen. Daher sollten wir uns nicht auf diesen Test verlassen. Es gibt noch haufenweise andere Testverfahren, auf die man am einfachsten im Internet stößt. Die Entwicklungen in den letzten Jahren umfassen Funktionen, die bereits garantieren können, dass eine Zahl prim ist.

ggT II - Der erweiterte Euklidsche Algorithmus

Wie im Kapitel über den *Euklidschen Algorithmus* bereits erwähnt wurde, hat der größte gemeinsame Teiler zweier Zahlen a (in der Menge N) und b (in der Menge N[0]) eine interessante Eigenschaft. Dieser lässt sich nämlich als Linearkombination von a und b mit ganzzahligen Koeffizienten darstellen. Somit gilt:

```
// u und v sind Elemente von Z
gcd(a, b) = g
g = u * a + v * b
```

Es gibt unendlich viele Paare u und v, die die genannte

Gleichung erfüllen. Mit dem *erweiterten Euklidschen Algorithmus* lassen sich diese beiden Koeffizienten berechnen.

Der normale Euklidsche Algorithmus führt gcd(a, b) auf gcd(b, a mod b) zurück. Der erweiterte Euklidsche Algorithmus führt zusätzlich u * a + v * b auf u' * b + v' * (a mod b) zurück, wobei u = v' und v = u' - a / b * v' gelten (Achtung: Ganzzahldivision!). Hier das Beispiel von oben:

```
       a      b  a mod b       u       v
   -----  -----  -------  ------  ------
1. 55913  20449    15015     -64     175  =  47 - 55913 / 20449 * -64
2. 20449  15015     5434      47     -64  = -17 - 20449 / 15015 *  47
3. 15015   5434     4147     -17      47  =  13 - 15015 /  5434 * -17
4.  5434   4147     1287      13     -17  =  -4 -  5434 /  4147 *  13
5.  4147   1287      286      -4      13  =   1 -  4147 /  1287 *  -4
6.  1287    286      143       1      -4  =   0 -  1287 /   286 *   1
7.   286    143        0       0       1  =   1 -   286 /   143 *   0
8.   143      0        -       1       0
```

Nachdem unser Algorithmus durchgelaufen ist, haben wir u = -64 und v = 175 und die Gleichung u * a + v * b = g. In unserem Beispiel ist g = 143 und wir testen:

```
a = 55913
b = 20449
g = gcd(a, b) = 143

u = -64
v = 175

  u *     a +   v *     b =   g
-64 * 55913 + 175 * 20449 = 143
```

Die Gleichung geht auf. Interessant ist, dass dies für jede Zeile zutrifft. So können wir Zeile 4 aus dem Beispiel nehmen und rechnen:

```
 u *    a +   v *    b =   g
13 * 5434 + -17 * 4147 = 143
```

Dieser Algorithmus funktioniert also nach dem Prinzip *rückwärts einsetzen*.

Auch dieser Algorithmus soll hier bewiesen werden. Wie oben erwähnt, wird die Darstellung von g = u * a + v * b auf g = u' * b + v' * (a mod b) zurückgeführt, wobei u = v' und v = u' - a / b * v' sind. Beweis:

```
  g
= (u' * b) + (v' * (a mod b))
= (u' * b) + (v' * (a - a / b * b))
= (u' * b) + (v' * a) - (a / b * v' * b)
= (v' * a) + (u' * b) - (a / b * v' * b)
= (v' * a) + (u' - a / b * v') * b
= u * a + v * b  // weil: u = v' und v = u' - a / b * v'
```

Dieser Algorithmus lässt sich nicht so einfach iterativ implementieren. Am einfachsten ist dieser rekursiv zu implementieren. Der hier vorgeführte Algorithmus dient nur zur Demonstration. Eigene Implementierungen sollten auf jeden Fall eine Rekursionsüberprüfung durchführen, damit sich die Funktion nicht zu oft selbst aufruft. Hier der Algorithmus:

```
// Rekursive Version des erweiterten Euklidschen Algorithmus
// Achtung: diese Version ist nicht thread-sicher!

VARIABLE u
VARIABLE v

GCD_EX(a, b) {
  LOCAL result, temp

  IF (b == 0) THEN
    u = 1
    v = 0
    RETURN a
  ELSE
    result = GCD_EX(b, a MOD b)
    temp = u
    u = v
    v = temp - a / b * v
```

```
    RETURN result
  ENDIF
}
```

```
// Hier ein C-ähnlicher Pseudo-Code einer thread-sicheren und
// rekursionsprüfenden Version. Ist der Rückgabewert 0, so wurde die Funktion
// öfter als maxcalls mal rekursiv aufgerufen (inklusive dem Hauptaufruf). In
// dem Fall sind u und v unbrauchbar.

huge_t gcd_ex(huge_t a, huge_t b, huge_t *u, huge_t *v, int maxcalls) {
  huge_t res, tmp;

  if (!maxcalls) return 0;
  if (!b) {
    *u = 1;
    *v = 0;
    return a;
  }
  res = gcd_ex(b, a % b, u, v, maxcalls - 1);
  tmp = *u;
  *u = *v;
  *v = tmp - a / b * *v;
  return res;
}
```

In diesem Text wurde öfter mit genau diesem Algorithmus das multiplikative Inverse zu e in der Menge Z[n] berechnet. Wie das? Wir betrachten noch einmal die Darstellung vom größten gemeinsamen Teiler als Linearkombination:

```
g = u * a + v * b
```

Wenn wir nun den größten gemeinsamen Teiler von e und n suchen, lautet die Gleichung:

```
gcd(n, e) = g

// Und:
g = u * n + v * e
```

Wenn nun e relativ prim zu n ist, dann ist der größte

gemeinsame Teiler 1. Also:

```
u * n + v * e = 1
```

Diese Gleichung überführen wir in eine Kongruenz in Z[n]:

```
u * n + v * e = 1 (mod n)
```

Und wir wissen, dass x * n = 0 (mod n) für jedes beliebige x gilt. Also:

```
u * n + v * e = 1 (mod n)

// Weil u * n = 0 (mod n):
0 + v * e = 1 (mod n)

// oder:
v * e = 1 (mod n)
```

Wir sehen deutlich, dass das Ergebnis der Multiplikation v * e kongruent zu 1 in Z[n] ist. Da eine Zahl mit seinem multiplikativen Inversen multipliziert immer das neutrale Element 1 der Multiplikation ergibt, wissen wir, dass v das multiplikative Inverse zu e sein muss. Hier ist anzumerken, dass v selbst nicht unbedingt in der Menge Z[n] ist, aber das ist kein Problem. Ist v negativ, bilden wir den Betrag und erhalten somit das additive Inverse von v. Wir invertieren v also. Danach reduzieren wir v, sodass der Wert innerhalb von Z[n] ist. Da wir v zuvor invertiert haben, müssen wir jetzt noch einmal das additive Inverse bilden. Damit gilt: ist v negativ, ist das multiplikative Inverse d zu e:

```
// Wenn v negativ
d = n - (|v| mod n)  // Ganzzahldivision!
```

Ist v hingegen positiv, so ist d wesentlich leichter zu berechnen:

```
// Wenn v positiv
d = v mod n
```

Das Diffie-Hellman-Protokoll

Hier wird ein Verfahren vorgestellt, das es ermöglicht, einen geheimen Schlüssel über eine unsichere Verbindung zu vereinbaren. Das Schlüsseltauschproblem wurde bereits weiter oben beschrieben. Bevor es RSA gab, war dieses Verfahren die einzige Lösung für das Problem. Wir gehen weit zurück bis in die Einführung in die modulare Arithmetik und greifen das Kommutativgesetz für die modulare Exponentiation auf:

```
(a^b)^c = (a^c)^b (mod n)
```

Die Kommunikationspartner Alice und Bob möchten nun einen geheimen Wert über eine Leitung vereinbaren, die durch jeden abgehört werden könnte. Dazu wählen sie sich eine große Primzahl n und eine Zahl b in Z[n]. Beide Zahlen dürfen öffentlich bekannt sein. Alice wählt nun eine geheime Zufallszahl x › 1 und berechnet:

```
p = b^x (mod n)
```

Sie schickt den Wert von p an Bob. Dieser wählt eine geheime Zufallszahl y › 1 und berechnet:

```
q = b^y (mod n)
```

Er schickt den Wert von q an Alice. Alice hat nun ihr x und Bobs q und berechnet:

```
k[0] = q^x (mod n)
```

Bob hat sein y und Alices p. Er rechnet:

```
k[1] = p^y (mod n)
```

Die Werte k[0] und k[1] sind gleich, denn es gilt:

```
p = b^x (mod n)
q = b^y (mod n)

k[0] = q^x = (b^y)^x (mod n)
k[1] = p^y = (b^x)^y (mod n)

// Und es gilt:
(b^x)^y = (b^y)^x (mod n)
```

Hier ein kleines Beispiel:

```
// Die öffentlichen Werte:
n = 82699
b = 13

// Alice wählt x und berechnet p:
x = 63141
p = b^x = 699 (mod n)

// Bob wählt y und berechnet q:
y = 13586
q = b^y = 15420 (mod n)

// Alice erhält q und berechnet:
k[0] = q^x = 15420^63141 = 3740 (mod n)

// Bob erhält p und berechnet:
k[1] = p^y = 699^13586 = 3740 (mod n)

// Der geheime Schlüssel ist also:
k = k[0] = k[1] = 3740
```

Nun wechseln wir in die Rolle des Angreifers. Nachdem wir die Kommunikation zwischen Alice und Bob abgehört haben, kennen wir die folgenden Werte:

```
n = 82699
b = 13
```

```
p = 699
q = 15420
```

Mehr wurde nicht übertragen. Es gibt nun zwei Möglichkeiten, den geheimen Schlüssel k zu berechnen. Entweder wir berechnen p^y (mod n) oder q^x (mod n). Hier stehen wir vor einem Problem. Wir kennen weder x, noch y, denn diese Werte wurden nicht übertragen. Man müsste also entweder b^t (mod n) für alle Werte von 2 bis n - 1 ausprobieren, bis wir entweder p oder q erhalten. Erhalten wir p, so ist t = x; erhalten wir q, so ist t = y. Oder man versucht, den Logarithmus zur Basis b von p oder q berechnen. Dies ist auch in Z[n] möglich, aber mindestens so schwer wie die Faktorisierung oder das Wurzelziehen und wird als das Problem des *diskreten Logarithmus* bezeichnet. Auch dies lässt sich nur nach einem Rateverfahren durchführen.

Das Protokoll wird also dadurch zunehmend sicherer, dass wir einen großen Wert für n wählen. Je größer dieser Wert ist, desto schwieriger ist der Logarithmus zu berechnen, da die Anzahl der Möglichkeiten steigt. Auch unser b sollte so gewählt werden, dass sich b^x (mod n) auf so viele Zahlen in Z[n] wie möglich abbilden lässt. Die Zufallszahlen sollten nicht zu klein sein, es sei denn, wir haben ein entsprechend großes b.

Problematisch ist bei diesem Protokoll die Wahl von n und b. Sichere Wertepaare müssen einige Voraussetzungen erfüllen. Zunächst sei gesagt, dass n nicht unbedingt prim sein muss, eine Primzahl die Sicherheit jedoch immens erhöht. In OpenSSL wird das folgendermaßen gehandhabt. Ein guter Wert für n ist eine Primzahl, für die (n - 1) / 2 ebenfalls eine Primzahl ist. Als b wird nach Möglichkeit 2, 3 oder 5 gewählt. Wenn b = 2 ist, dann sollte n = 11 (mod 24) sein; für b = 3 sollte n = 5 (mod 12) sein und für b = 5 sollte n = 3 (mod 10) oder n = 7 (mod 10) sein. Weiters sollte b relativ prim zu phi(n) = n - 1 sein. Ist diese Voraussetzung erfüllt, so wird b als *Generator* für n bezeichnet. Die Begründung dieser eher willkürlich erscheinenden Tests ist sehr kompliziert und aufwändig und sprengt den Rahmen dieser Dokumentation. Ich plane eine weitere Dokumentation speziell über Angriffe auf RSA und Diffie-Hellman. Dort wird dann darauf eingegangen.

Mathematiker bilden eine enge Verwandtschaft zwischen dem Faktorisierungsproblem und dem diskreten Logarithmus. Viele glauben, dass mit der Lösung eines dieser beiden Probleme auch gleichzeitig das andere Problem gelöst ist.

Das Nebeneffektproblem

Die Attacke über eine *Nebeneffektanalyse* (side channel attack) ist ein nennenswertes, jedoch oft unterschätztes Problem des RSA-Verfahrens und des Diffie-Hellman-Protokolls (und vieler anderer, auch symmetrischer Verfahren). Wird an einem Computer, zu dem mehrere Personen gleichzeitig Zugriff haben (beispielsweise über ein Netzwerk), ein RSA-Schlüssel erstellt, so kann ein Angreifer die dabei benötigte CPU-Kraft, den RAM-Verbrauch, den verwendeten Zufallszahlengenerator und die Dauer der Schlüsselgenerierung analysieren. Dazu sehen wir uns den *erweiterten Euklidschen Algorithmus* an.

Anhand der verwendeten CPU-Kraft sammelt ein Angreifer Informationen über den geheimen Exponenten d, der mit diesem Algorithmus ermittelt wird.
Der Algorithmus führt beispielsweise eine Division mit zunehmend größeren Zahlen durch. Der Angreifer kann am RAM-Verbrauch sehen, ob gerade dividiert wird. Die Division erfordert mehr RAM, als eine Addition oder Subtraktion, da diese nicht durch den Prozessor durchgeführt wird (wir arbeiten mit sehr großen Zahlen, müssen diese also emulieren). Steigt der RAM-Verbrauch, so weiß der Angreifer, dass gerade dividiert wird. Sobald der RAM-Verbrauch wieder sinkt, weiß er, dass nicht mehr dividiert wird. So kann er die Zeit messen, die zum Dividieren benötigt wurde. Je nach Divisionsalgorithmus sinkt die Dauer der Division mit der Größe des Ergebnisses und ist oftmals nur vom Ergebnis abhängig. So kann ein Angreifer den privaten Exponenten d des generierten Schlüssels schätzen und schließlich durch kurzes Ausprobieren ermitteln. Tatsächlich sind die ermittelten Werte so genau, dass der Angreifer selten lange raten muss.

Auf Linux-Maschinen wird oft ein systemweiter Zufallszahlengenerator verwendet. Dies stellt kein Problem dar, solange dieser nur zur

Initialisierung eines eigenen Generators genutzt wird. Der Generator verfolgt Systemereignisse und wertet sie aus. Die Ergebnisse werden zum Status des Zufallszahlengenerators (entropy pool) hinzugefügt. Verfolgt werden weitgehend zufällige Ereignisse wie das thermische Rauschen von Festplattenköpfen oder die Mausbewegungen und Tastatureingaben des Anwenders. Es zeigt sich, dass jeder diese Ereignisse generieren und somit den Zufallszahlengenerator beeinflussen kann. Der Angreifer kombiniert dies mit der CPU-Analyse, um die generierten Zufallszahlen in etwa *mitlesen* zu können. Er wird nicht die Zufallszahlen selbst kennen, aber er wird genug Informationen haben, um mit dem endgültigen Modul zusammen die CPU-Werte zu analysieren. Bei der Generierung des RSA-Schlüssels wird das genau bei der Wahl der beiden Primfaktoren gefährlich. Falls tatsächlich der systemweite Zufallszahlengenerator genutzt wird, kann der Angreifer mit viel Geschicklichkeit aus den ermittelten Daten bis zu einem gewissen Grad die beiden Primzahlen ablesen.

Diese Attacke lässt sich abwehren, indem man absichtliche Verzögerungen und unnütze RAM-Spielereien in die verwendeten Algorithmen einbaut. Dadurch erscheint dem Angreifer die Systemlast rein zufällig. Auch Zufallszahlen sollten durch einen ähnlich implementierten *eigenen* Algorithmus generiert werden. Falls ein solcher systemweiter Generator wie in Linux zur Verfügung steht, sollte dieser auf jeden Fall genutzt werden, um den eigenen Generator zu initialisieren. Er sollte **nicht** für die eigentliche Generierung der Zufallszahlen genutzt werden, denn das ist auch nicht der Zweck dieses Generators. Die einzige sichere Abwehr ist natürlich, seinen Schlüssel auf einem Rechner zu generieren, zu dem kein anderer Zugang hat.

Das "Man in the Middle"-Problem

Es existiert eine sehr effiziente Attacke auf das RSA-Verfahren und das Diffie-Hellman-Protokoll. Diese wird als *Man in the Middle - Attacke* (*MITM attack*) bezeichnet. Sie lässt sich durchführen, wenn man nicht nur die Möglichkeit des Abhörens, sondern auch die Möglichkeit der Manipulation des Kommunikationskanals hat. Dazu stellen wir uns eine Kommunikation zwischen Alice und Bob über den Boten *Mallory* vor. Mallory ist für die Übertragung der Nachrichten zwischen Alice und Bob zuständig. Er hat somit die Möglichkeit, die Nachrichten vor der Übermittlung abzuändern.

Alice und Bob haben noch nie ihre öffentlichen Schlüssel ausgetauscht und wollen dies jetzt tun. Alice schickt ihren Schlüssel A los. Mallory tauscht diesen Schlüssel gegen seinen eigenen Schlüssel M aus und schickt diesen an Bob, statt des eigentlichen Schlüssels A. Bob macht das gleiche. Er schickt seinen Schlüssel B an Alice. Auch diesen ersetzt Mallory während der Übertragung durch seinen Schlüssel M. Alice und Bob merken von der Manipulation nichts. Alice möchte nun eine Nachricht an Bob schicken und verschlüsselt diese mit M, statt mit B. Mallory empfängt die Nachricht und entschlüsselt sie. Da er B kennt, verschlüsselt er die Nachricht wieder mit B und schickt sie weiter an Bob. Bob verschlüsselt seine Nachricht mit M (statt A) und schickt sie los. Mallory entschlüsselt diese, verschlüsselt sie wieder mit A und schickt sie weiter. Dadurch kann Mallory den gesamten Datenverkehr zwischen Alice und Bob abhören, ohne, dass diese unmittelbar etwas davon merken.

Es stellt sich als besonders schwierig heraus, sich vor dieser Attacke zu schützen. Alice kann ihren Schlüssel signieren. Dies funktioniert aber nur dann, wenn Bob ihren Schlüssel schon vorher hatte, da sonst Mallory einfach seinen eigenen signierten Schlüssel an Bob schickt. Auch mit Prüfsummen funktioniert dies nicht, denn Mallory kennt A und B, kann diese Prüfsummen also problemlos selbst erzeugen.

Eine mögliche Abwehr lässt sich über eine Nebeneffektanalyse durchführen (siehe letztes Kapitel). Dazu benötigt man die durchschnittliche Zeit, die eine unverschlüsselte Nachricht von Alice zu Bob (über Mallory!) braucht. Mallory muss jedes einzelne Paket zwischen Alice und Bob entschlüsseln und erneut verschlüsseln, wenn er eine MITM-Attacke durchführt. Tut er dies nicht, so reicht er die Pakete einfach nur weiter. Er benötigt also mehr Zeit, wenn er eine Attacke durchführt. Da RSA und Diffie-Hellman sehr langsame Verfahren sind, ist der Zeitunterschied zwischen Attacke und Nichtattacke messbar. Braucht ein Paket von Alice zu Bob wesentlich länger als sonst, so besteht die Gefahr, dass man einer MITM-Attacke ausgesetzt ist.

Diese unkryptologische Abwehr lässt sich allerdings nur bei aktivem Informationsaustausch (z.B. Chats oder Dateiübertragungen) implementieren. Bei passivem Austausch (z.B. Emails oder FTP) ist der Zeitunterschied nicht mehr messbar, da er von zu vielen Umständen abhängt and zu stark variiert. Am

besten führt man diese Attacke ganz protokollunabhängig durch. Sobald der Dialog aufgebaut ist, sendet Alice ein willkürliches Paket an Bob. Nachdem Bob dieses Paket erhalten hat, sendet er ein willkürliches Paket zurück. Mallory müsste in diesem Fall insgesamt vier RSA-Operationen durchführen. Er braucht also länger als sonst. Fälschen kann er hier nicht, denn es gibt nichts, was zu fälschen wäre (die Nachricht ist willkürlich). Diese Abwehr ist jedoch zu natürlich, kann falschen Alarm auslösen (z.B. bei Routerproblemen) und kann umgangen werden (wenn Mallory einen schnellen Rechner hat und dadurch den Zeitunterschied minimiert).

Effektiv lässt sich diese Attacke nur abwehren, indem man die Schlüssel über eine sichere Leitung austauscht. Am besten übergibt man sie persönlich. Auch der Austausch über ein *Trustcenter* ist möglich, sofern beide Endpunkte der Kommunikation über eine sichere Leitung zu diesem verfügen (Firmennetzwerke und das Internet sind **keine** sicheren Leitungen!) und das Trustcenter selbst nicht korrupt ist.

Gute Zufälle, schlechte Zufälle - Der Zufallszahlengenerator

Ein altbekanntes Problem in der Computertechnologie ist es, Chaos ins System zu bringen. Wie kann ein System, das auf purer Logik und eindeutigen Regeln aufbaut, eine chaotische oder *zufällige* Aktion durchführen? Auch dieses Chaos muss systematisch ablaufen. Das System entscheidet sich für einen vieler Wege. Wie können wir dafür sorgen, dass diese Entscheidung auf purem Zufall basiert? Die Antwort lautet: gar nicht! Um puren Zufall in ein eintöniges, logisches, unzufälliges System zu bringen, brauchen wir eine Quelle für diesen Zufall, denn innerhalb dieses Systems ist kein Zufall möglich. Diese Quelle wird *Zufallszahlengenerator* (RNG - random number generator) genannt. Es handelt sich also um ein Modul, das Eingaben entgegen nimmt und Ausgaben produziert, die zufällig erscheinen. Die Eingabe kann dabei allerdings auf echtem Zufall basieren. Radioaktiver Zerfall ist beispielsweise im momentanen Zustand der Physikwissenschaft ein zufälliger Prozess. Also könnte so ein Modul den radioaktiven Zerfall analysieren und Zufallszahlen aus den Messergebnissen bilden. Was aber nun, wenn eine derartige "echte" Zufallszahlenquelle nicht zur Verfügung steht? In diesem Fall müssen wir eine solche Quelle auf mathematischer Basis *simulieren*. Die erzeugten Zufallszahlen werden dann *Pseudozufallszahlen* (pseudo-random

numbers) genannt und der dazugehörige Generator nennt sich *Pseudozufallszahlengenerator* (PRNG - pseudo-random number generator). Ein solcher greift nach allen Ereignissen, die halbwegs zufällig sind (beispielsweise Benutzeraktionen oder Festplattenzugriffe) und bildet daraus einen internen Zustand (*entropy pool*). Aus diesem internen Zustand berechnet er dann Pseudozufallszahlen, die keine Informationen über den internen Zustand geben. Genau diese Art von Zufallszahlengeneratoren sollen hier erklärt und implementiert werden. Daher nennen wir sie der Bequemlichkeit halber einfach *Zufallszahlengeneratoren* statt *Pseudozufallszahlengeneratoren*.

Oft wird übersehen, dass Zufallszahlen ein wichtiger Teil der Kryptographie sind. Sie werden für die Generierung von Schlüsseln, für die Signatur und sogar für die Verschlüsselung selbst genutzt. Somit kommt es darauf an, kryptographisch hochwertige Zufallszahlen zu generieren. Zufallszahlen also, die mit keinem Aufwand der Welt vorhersehbar sind. Sind Zufallszahlen vorhersehbar, sind in RSA z.B. die beiden Primzahlen ebenfalls vorhersehbar. Beispielsweise wird der Generator oft mit der aktuellen Uhrzeit initialisiert. Dies ist ein fataler Fehler, denn das heißt, um einen Schlüssel zu reproduzieren, braucht man einfach nur die Uhrzeit, zu der er erstellt worden ist. Viele Zufallszahlengeneratoren ändern ihren Status nämlich nicht und liefern mit dem gleichen Initialwert (random seed) auch immer die gleiche Zahlenreihe.

Interessanterweise ist es nicht schwierig, Zufallszahlen zu produzieren. Es gibt hunderte solcher Verfahren, eines sicherer als das andere. Doch bevor wir zu den sicheren Zufallszahlengeneratoren übergehen, sehen wir uns erst einmal einen unsicheren an:

```
r[0] = s (mod 13)
r[x] = r[x - 1] + 5 (mod 13)
```

s ist der Initialwert (random seed). Dieser Generator hat eine Periode von 13. Er liefert also alle Zahlen in Z[13] genau einmal (siehe Euklidscher Algorithmus). Diese Sequenz wird dann wiederholt. Sehen wir uns die Sequenz an:

```
r[0] = s (mod 13)
```

```
r[1] = s + 5 (mod 13)
r[2] = s + 10 (mod 13)
r[3] = s + 2 (mod 13)
r[4] = s + 7 (mod 13)
r[5] = s + 12 (mod 13)
r[6] = s + 4 (mod 13)
r[7] = s + 9 (mod 13)
// ...

// Mit s = 8
r[0] = 8
r[1] = 0
r[2] = 5
r[3] = 10
r[4] = 2
r[5] = 7
// ...
```

Diese Zahlen sehen doch äußerst zufällig aus, nicht wahr? Nein, nicht wahr! Das geschulte Auge sieht, dass wir wiederholt eine Konstante addieren und, dass das Ergebnis ab einer gewissen Grenze wieder umklappt. Der Angreifer schaut sich an, wo es umklappt und merkt, dass nach der 10 die 2 kommt. Er weiß also:

```
10 + x = 2 (mod n)
```

Also sieht er sich die vorhergehende Zahl an. Nach der 5 kam die 10, also könnte 5 zu 5 addiert worden sein. Er geht davon aus, dass jedes mal 5 addiert wird und schreibt dann seine Gleichung um:

```
10 + 5 = 2 (mod n)
```

Er glaubt also:

```
10 + 5 = k * n + 2
```

Er probiert erst: k = 1 und erhält damit eine Gleichung, die er lösen kann:

```
10 + 5 = 1 * n + 2
```

```
n + 2 = 15
n = 15 - 2
n = 13
```

Damit stellt er folgende Funktion auf, mit der er die nächste Zufallszahl voraussehen können will:

```
c(x) = x + 5 (mod 13)
```

Der Zufallszahlengenerator gibt den Wert 11 zurück. Er versucht nun, die nächste Zufallszahl vorauszusehen:

```
c(11) = 11 + 5 = 3 (mod 13)
```

Seine Schätzung besagt, dass nach seiner Schätzfunktion 3 die nächste Zufallszahl sein müsste. Also wartet er auf die nächste Zahl vom Generator und diese ist:

```
r[x - 1] = 11                 // Die vorige Zufallszahl
r[x] = r[x - 1] + 5 (mod 13)  // Die jetzige (nächste) Zufallszahl
r[x] = 11 + 5 (mod 13)
r[x] = 3 (mod 13)
```

Er probiert seine Schätzung noch mit weiteren Zahlen aus, die der Zufallszahlengenerator ausgibt und sie treffen zu. Damit kennt er die geheimen Werte der Funktion und kann jede beliebige Zufallszahl voraussehen. Eine weitere Schwäche dieses Generators ist die viel zu kleine Periode. Je größer die Periode eines Generators, desto sicherer ist er. Sobald der Generator die Sequenz wiederholt, kann ein Angreifer alle weiteren Zufallszahlen voraussehen, da sie sämtlich schon einmal generiert worden sind.

Es gibt nun eine sehr wichtige Regel in der Kryptographie. **Die Sicherheit eines Verfahrens darf nicht darauf beruhen, dass das Verfahren geheim bleibt!** Können wir einen Zufallszahlengenerator schreiben, der auch dann sicher ist, wenn das Verfahren öffentlich bekannt ist? Ja, das können wir. Wir machen uns bekannte Chiffre- und Prüfsummenverfahren zum Vorbild und kombinieren diese, um einen Zufallszahlengenerator zu erhalten. Damit stellt sich heraus, dass es sogar besonders einfach ist, zuverlässige

Zufallszahlen zu generieren. Doch auch in Zufallszahlengeneratoren stecken wichtige Tücken, die später behandelt werden. Hier werden nun einige Ideen für Zufallszahlengeneratoren aufgezeigt.

---

1. Rückgabewert von Hash-Funktionen

Die Spezifikation einer *Hash-Funktion* wurde bereits im Kapitel über die Anwendung von RSA aufgeführt. Eine Hash-Funktion ist eine *Einwegfunktion*, die nicht umkehrbar ist, deren Rückgabewert aber auch keine Informationen über den Eingabewert liefert und für die ein Rückgabewert sehr schwer reproduzierbar ist. SHA1 ist dabei zur Zeit die empfehlenswerteste Funktion, wobei dennoch MD5 am weitesten verbreitet ist. Bezeichnen wir unsere Funktion also als h(x). Wir wissen, dass es so gut wie unmöglich ist, zwei Werte für x zu finden, die den selben Rückgabewert erzeugen. Wir wissen auch, dass wir h(x) nicht in absehbarer Zeit auf x zurückführen können (im Optimalfall können wir das gar nicht). Dies können wir nutzen, um einen Zufallszahlengenerator zu schreiben.

Wir haben einen Status e (entropy pool) und einen Inkrementen i. Wir initialisieren e mit einem zufälligen Startwert (random seed), der nicht bekannt sein darf. Eine Zufallszahl ermitteln wir dadurch, dass wir h(e) berechnen. Der Rückgabewert ist unsere Zufallszahl. Danach addieren wir i zu e. Die beiden Werte stellen den *geheimen Zustand* des Generators dar, dürfen also nicht öffentlich bekannt sein. Erfüllt die Funktion h(x) die oben aufgeführten Anforderungen, so können wir h(e) nicht auf e zurückführen. Auch i können wir nicht ermitteln, denn dazu müsste h(e) Informationen über e preisgeben.

Als Angreifer sehen wir immer nur den Rückgabewert der Funktion h(e) und diese gibt, wenn h(x) die oben gestellten Forderungen erfüllt, keinerlei Informationen über e. Sollten wir jemals e ermitteln können, haben wir ein weiteres Problem. Wir können die nächste Zufallszahl voraussehen, aber nicht die übernächste, denn wir kennen e + i nicht. Es ist eine gute Idee, i während der Laufzeit nach Systemereignissen zu verändern. Auch, wenn die

Veränderung von i bekannt ist, kann der Angreifer die weiteren Zufallszahlen nicht voraussehen, da er weder den aktuellen Wert von i, noch e kennt.

Der Generator ist vollständig kompromittiert, sobald der Angreifer e und i kennt. In diesem Fall kann er nämlich jede weitere Zufallszahl voraussehen. Die Periode des Generators hängt von der benutzten Hash-Funktion ab. Erfüllt sie die oben gestellten Anforderungen, so ist die Periode unendlich, sofern weder e, noch i (wenn eine Veränderung von i stattfindet) irgendwann umklappen. Dies ist nur rein mathematisch möglich. Ein Computer hat immer seine Grenzen. Die genaue Periode ist kaum vorhersagbar. Die Sicherheit dieses Generators hängt von der benutzten Hash-Funktion und dem Startwert für e ab. Liefert h(x) irgendwelche Informationen über x, oder lässt sich ein Zusammenhang zwischen h(x) und h(x + 1) herstellen, so ist der Generator nicht sicher. Bei einer guten Hash-Funktion ist dies unmöglich und somit ist das Verfahren bei Verwendung einer guten Hash-Funktion als sicher einzustufen. Den Startwert für e können wir entweder vererben (wir führen e dauerhaft fort), oder wir verwenden zufällige Informationen, die sich ständig verändern. Die Uhrzeit kann mit einbezogen werden, sie sollte jedoch nicht allein benutzt werden, sonst kennt der Angreifer e, wenn er weiß, wann der Generator initialisiert wurde. Zusätzlich erschwert wird dem Angreifer das Handwerk, wenn wir i laufend verändern. In dem Fall kann er bei erfolgreicher Kompromittierung immer nur die nächste Zufallszahl voraussagen, aber nicht die übernächste. Allerdings sollte die Veränderung von i keine Informationen über das System bekannt geben. Optimal wäre es, wenn wir für diese Veränderung auch eine Hash-Funktion in Betracht ziehen. Empfehlenswert ist die Funktion SHA1.

2. Eulers phi-Satz

Wir können uns folgenden Satz zunutze machen:

```
// Wenn x in Z[n] und gcd(x, n) = 1, dann:
x^phi(n) = 1 (mod n)
```

Ähnlich wie bei RSA erweitern wir diesen Satz ein wenig:

```
x^phi(n) * x^phi(n) = 1 * 1 (mod n)

// Das heißt:
x^(t * phi(n)) = 1 (mod n)
x^(t * phi(n)) * x = 1 * x (mod n)
x^(t * phi(n) + 1) = x (mod n)

// Der Exponent:
k = t * phi(n) + 1

// Oder als modulare Kongruenz:
k = t * phi(n) + 1 (mod phi(n))
k = 0 + 1 (mod phi(n))
k = 1 (mod phi(n))

// Diesen teilen wir auf:
e * d = 1 (mod phi(n))
k = e * d (mod phi(n))

// Das heißt:
x^k = x (mod n)

// oder:
x^(e * d) = x (mod n)

// Aufgeteilt:
x^e = y (mod n)
y^d = x (mod n)
```

Wir wissen bereits, dass y keine Informationen über x
preisgibt. Anders als in RSA wählen wir eine große Primzahl als n.
Wenn n prim ist, gilt: phi(n) = n - 1. Also suchen wir
uns ein 1 ‹ e ‹ n - 1, das relativ prim zu n - 1 ist.
e muss ungleich 1 sein, sonst ist x^e = x (mod n) für
jedes beliebige x. Auch muss e ungleich 0 sein, sonst
gilt x^e = 1 (mod n) für jedes beliebige x. e
darf auch kein Vielfaches von phi(n) = n - 1 sein, sonst gilt:
x^e = 1 (mod n) für jedes x ungleich 0, wenn
gcd(x, n) = 1, denn es gilt: x^phi(n) = 1 (mod n) für
jedes x ungleich 0, wenn gcd(x, n) = 1. Dies trifft für
jedes x ungleich 0 in Z[n] zu, denn da n prim
ist, ist jedes x ungleich 0 in Z[n] relativ prim zu
n. Am einfachsten ist es also, ein e zu wählen, das
zwischen 1 und n - 1 und relativ prim zu n - 1 ist. Der
Status des Generators ist p. Wir brauchen keinen zufälligen
Startwert und keinen Inkrementen. Der Startwert von p ist 2.
e und n bleiben geheim.

Eine Zufallszahl ermitteln wir dadurch, dass wir
(p^e mod n) (mod 2) berechnen. Wir berechnen also
p^e (mod n) und nehmen nur das niederwertigste Bit als Ergebnis.
Hinweis: die Funktion lässt sich durch eine binäre *und*-Operation an
Stelle des Divisionsrests durch zwei optimieren. Also unsere optimierte
Variante: (p^e mod n) and 1. Nachdem wir das zufällige Bit erzeugt
haben, ersetzen wir p durch p + 1 (mod n). Resultiert
p = 0, so setzen wir p auf 1. Das garantiert eine
zentrische Verteilung (siehe weiter unten).

Aus dem Bit, das dieser Generator zurückgibt ermittelt der Angreifer überhaupt
keine Informationen über p, e oder n. Er weiß
weder, wie groß n ist, noch weiß er, welcher Exponent verwendet
wurde. Der Grund dafür, dass wir nur ein einziges Bit zurückgeben, ist, dass
der Angreifer sonst Informationen über n sammeln kann. Ist der
Rückgabewert x, so weiß der Angreifer, dass n auf jeden
Fall größer als x ist, denn ein Ergebnis modulo n ist
auf jeden Fall kleiner als n. Auch hier ist es wieder
empfehlenswert, p nach Systemereignissen abzuändern. Interessant
ist, dass der Angreifer p ruhig kennen darf, denn dadurch kennt er
p^e (mod n) trotzdem nicht und kann somit die nächste Zufallszahl
nicht voraussehen.

Erfolgreich kompromittiert ist das Verfahren, sobald der Angreifer
p, e und n kennt. Damit kann er jede weitere
Zufallszahl voraussehen. Das Verfahren ist als sicher einzustufen, denn es
leidet nicht unter den Schwächen von RSA. Es gibt keine Zahl, die man
faktorisieren müsste. Auch der Versuch, die Wurzel aus p^e (mod n)
zu ziehen, scheitert, denn der Angreifer kennt vom Ergebnis nur ein einziges
Bit. Die Periode dieses Verfahrens ist n - 1, da
p^e (mod n) in jedem Fall auf eine unterschiedliche Zahl abgebildet
wird, sofern e die Anforderungen für einen gültigen Exponenten
erfüllt (sonst würde auch RSA nicht funktionieren). Da p bei diesem
Generator nicht 0 werden kann, ist die Periode 1 weniger als n.
Wenn wir nun zusätzlich p nach Systemereignissen abändern, kann der
Angreifer selbst aus p, e und n nicht die
nächste Zufallszahl ermitteln, da sich p ständig verändert. Für die
Veränderung sollte ein sicheres Hash-Verfahren wie SHA1 in Betracht gezogen
werden.

Wichtig ist es also, einen großen Wert für n zu wählen. Dabei müssen wir allerdings nicht wie bei RSA übertreiben, doch je kleiner unser n ist, desto leichter ist es zu *erraten*. Der Angreifer hat noch ein Problem. Er braucht unbedingt alle drei Werte, um p^e (mod n) berechnen zu können. Wenn wir p nicht geheim halten, so muss der Angreifer nur noch zwei Werte erraten. Anders als bei anderen Verfahren sollte p nicht zu intensiv verändert werden, sonst herrscht die Gefahr, dass es nur noch auf bestimmten Zahlen landet. Optimal ist also ein 64 bis 128 Bits langer Wert für n und ein e, das auf purem Zufall basiert. Wählen wir einen 128 Bits großen Wert für n, so können e und n bis zu 2^127 * (2^127 - 3) verschiedene Wertekombinationen annehmen. Der Grund dafür ist, dass die effektive Länge von n nur 127 Bits beträgt, da das unterste Bit immer 1 ist (n ist prim; ist n ungleich 2, so ist es also in jedem Fall ungerade). Auch von e beträgt die effektive Länge nur 127 Bits, denn es muss auch in jedem Fall ungerade sein. Da n eine große Primzahl (also nicht 2) und damit ungerade ist, ist n - 1 auf jeden Fall gerade. e muss relativ prim zu n - 1 sein und kann daher nicht gerade sein. Also ist auch bei e das unterste Bit immer 1. Da wir 0, 1 und n - 1 für e nicht verwenden können, haben wir noch einmal drei Zahlen weniger.

3. Symmetrische Chiffren

Ein weiteres simples Verfahren ist die Anwendung symmetrischer Chiffren. Die momentan empfehlenswerteste Chiffre ist das Rijndael-Verfahren, das als Kandidat für den *Advanced Encryption Standard* (AES) am besten abgeschnitten hat. Oft wird das Verfahren einfach als *AES-Verfahren* bezeichnet. Die Schlüssellänge ist variabel. Hier nehmen wir dieses Verfahren als Beispiel.

Wir gehen ähnlich wie beim Hash-Verfahren vor. Erst generieren wir irgendeinen Rijndael-Schlüssel k und einen zufälligen Status e.
Diese Werte können aus Systemereignissen gebildet werden. Die Uhrzeit sollte dabei zwar in Betracht gezogen werden, aber auf keinen Fall alleine. Man kann einen systemweiten Zufallsgenerator wie unter Linux verwenden oder so viele Systeminformationen wie möglich auslesen. Hier ist Vorsicht geboten. Wir müssen die Informationen so kombinieren, dass ein Angreifer bei erfolgreicher Kompromittierung aus den beiden Startwerten keine Informationen über das

System ermitteln kann. Optimal ist die Verwendung einer bewährten Hash-Funktion wie SHA1. Der Wert von k bleibt geheim.

Eine Zufallszahl ermitteln wir, indem wir e mit dem Schlüssel k verschlüsseln. Das Resultat lässt sich weder auf e, noch auf k zurückführen (sonst wäre die die Chiffre unsicher). Danach ersetzen wir e durch e + 1. Es empfiehlt sich, e laufend nach Systemereignissen abzuändern. Kann der Angreifer den Wert von e ermitteln, nützt ihm das nichts, denn er braucht den Wert von k, um e verschlüsseln zu können. Es muss darauf hingewiesen werden, dass bei einer schwachen Chiffre eine Kryptanalyse möglich ist, denn wir wissen, dass e jedes mal durch e + 1 ersetzt wird. Eine solche Kryptanalyse hat sich beim Rijndael-Verfahren als besonders schwer erwiesen.

Erfolgreich kompromittiert ist das Verfahren, wenn der Angreifer k ermitteln kann. Dadurch kann er die Zufallszahlen entschlüsseln und erhält e. Diese Gefahr kann stark eingeschränkt werden, indem man wie beim obigen Verfahren vom Ergebnis nur ein einziges Bit zurückgibt. Es ist nicht notwendig, jedes mal ein anderes Bit des Ergebnisses zurückzugeben, denn alle Bits erscheinen zufällig. Im Prinzip würde das nur die Kryptanalyse erleichtern. Es sollte also jedes mal das gleiche Bit des Ergebnisses zurückgegeben werden. Dadurch hat der Angreifer ein Problem. Er kennt zwar den Schlüssel, aber es gibt keinen Chiffretext, den er entschlüsseln könnte. Wird obendrein e laufend verändert, so nützt ihm auch der Schlüssel k nichts, denn selbst, wenn er k und e kennt, kann er keine Zufallszahlen voraussehen.

Die Sicherheit des Verfahrens basiert einerseits auf der Auswahl der Startwerte und andererseits auf der verwendeten Chiffre. Sie wird zusätzlich erhöht, wenn man vom Ergebnis nur ein Bit zurückgibt. Da Kryptanalytiker bisher noch keine Schwächen in Rijndael feststellen konnten, kann dieses Verfahren als sicher eingestuft werden. Nochmal wird die Sicherheit des Verfahrens dadurch sehr stark erhöht, dass wir e laufend verändern. Das heißt, der Angreifer kann das Verfahren selbst dann nicht kompromittieren, wenn er den kompletten geheimen Zustand des Generators kennt.

Handelt es sich um eine sichere Chiffre, so ist die maximale Periode von der Blockgröße der Chiffre abhängig. Arbeiten wir mit einer Blockgröße von 128 Bits, so ist die Periode maximal 2^128. Somit hängt die Periode vom Chiffreverfahren und von der Anzahl an Bits für e ab. Im Rijndael-Verfahren ist sie also 2^b, wobei b die Anzahl der Bits ist, die für e reserviert wurde. Sie kann jedoch nicht höher sein, als die Blockgröße der Chiffre zulässt. Ist die Blockgröße c Bits, so ist die maximale Periode 2^c, auch, wenn 2^b größer ist.

Die Zahlen, die diese Generatoren produzieren, sind keine echten Zufallszahlen, da sie sämtlich auf mathematischem Wege berechnet werden. Sie sind also systematisch. Wichtig ist es also, ein System zu finden, das dem Empfänger der Zufallszahlen und vor allem einem Angreifer zufällig erscheint. Jeder Pseudozufallszahlengenerator muss irgendwie initialisiert werden. Diese Initialisierung muss durch eine zufällige Quelle erfolgen. Auch wird darauf hingewiesen, dass es eine sehr gute Idee ist, den geheimen Zustand des Generators während der Laufzeit auf möglichst zufälliger Basis zu verändern. Hier stehen wir vor einem großen Problem. Wie sollen wir einen Zufallszahlengenerator schreiben, wenn dieser selbst schon Zufallszahlen benötigt? Hier werden einige Ideen für Quellen solcher Zahlen zusammengestellt:

- Aktuelles Datum und aktuelle Uhrzeit
- Bildschirminhalt
- Position der Festplattenköpfe
- Seriennummern momentan eingelegter Wechseldatenträger
- Tastatureingaben
- Mausposition und Mausbewegungen
- Rauschen eines Mikrofons
- Zeit seit Systemstart (uptime)
- Temporäre Dateien
- RAM-Inhalt
- Inhalt der Swap-Partition oder -Datei
- Systemweiter Zufallszahlengenerator
- Netzwerkverkehr
- Systemzustände (Sequenznummern o.ä.)
- Laufende Prozesse
- Gerätezustande und Schnittstellenverkehr
- Aufträge in der Druckerwarteschlange

Es zeigt sich, dass es jede Menge solcher Quellen gibt. All diese Informationen sollten zusammengestellt und durch eine bewährte Hash-Funktion in eine Prüfsumme umgewandelt werden, die dann verwendet werden kann. Der Grund dafür, dass wir nur die Prüfsumme verwenden, ist, dass ein Angreifer, der den Generator kompromittiert, sonst an alle möglichen Systeminformationen kommen könnte. Der RAM-Inhalt ist ein gefährliches Beispiel. Dadurch, dass wir eine Hash-Funktion verwenden, können wir jede beliebige Information für Initialwerte nutzen. Es sollten so viele Informationen wie möglich genutzt werden, damit die Startwerte so schwer wie möglich zu erraten sind.

Das ist nicht das einzige Problem, das wir mit Zufallszahlengeneratoren haben. Es verbergen sich eine Reihe Tücken in solchen Generatoren. Einige werden hier zusammengestellt und erklärt.

---

1. Verteilungsproblem

Viele Zufallszahlengeneratoren haben ein kleines Problem. Nehmen wir einen Generator an, der nur einzelne Bits liefert, also Nullen und Einsen. Liefert dieser eher Nullen oder eher Einsen? Im Optimalfall liefert er beides gleich oft, doch dies ist eher selten der Fall. Als Beispiel nehmen wir eine abgeänderte Version des 2. Generators, der oben vorgestellt wurde. Wir nehmen eine Primzahl n und ein e, das relativ prim zu n - 1 ist. Wir setzen p = 2 und ermitteln Zufallszahlen durch (p^e mod n) (mod 2). Danach ersetzen wir p durch p + 1 (mod n). Damit haben wir den selben Generator wie oben. Wir erweitern diesen Generator, indem wir zusätzlich ein m in Z[n] wählen und die Zufallsfunktion umschreiben: (p^e mod n) (mod m). Der Angreifer hat hier ein weiteres Problem. Er braucht erst m, um n überhaupt schätzen zu können.

Als Beispiel setzen wir n = 13, m = 10, e = 5 und unser p ist eben 2. Wir generieren eine Zufallszahl. Ist p^e (mod n) unter m, so erhalten wir eine Zahl zwischen 0 und 9. Ist p^e (mod n) über oder gleich m, so erhalten wir eine Zahl zwischen 0 und 2. Es zeigt sich deutlich, dass wir eher 0 bis 2 erhalten, als 3 bis 9, da diese Möglichkeit zwei mal besteht. Der Angreifer hat hier eine hervorragende Angriffsfläche. Er braucht sich nur die Verteilung

der generierten Zufallszahlen ansehen und wird nach einer gewissen Zeit feststellen, dass 0 bis 2 öfter generiert werden, als die anderen Zahlen. Er sieht sich die größte generierte Zahl an und diese ist 9. Er schätzt also, dass m = 10 ist. Also stellt er folgende Gleichung auf, mit der er die größtmögliche Zahl für p^e (mod n) berechnen können will:

```
9 + x = 2 (mod 10)
9 + x = k * 10 + 2
```

Er braucht also nur noch den Wert für k zu schätzen und hat damit den Wert von n. Das senkt den Schlüsselraum des Generators von 2^(x - 1) * (2^(x - 1) - 3) auf 2^(x - 1) - 3, da er den Wertebereich für n nicht mehr durchsuchen muss. Aus 254 effektiven Bits werden also nur noch 127 effektive Bits. Das Verteilungsproblem ist also durchaus ernst zu nehmen!

Das beste wäre es nun, den Generator umzuschreiben, sodass die Verteilung gleich ist, doch das können wir nicht immer. Vor allem bei echten Generatoren, die auf physikalischen Ereignissen basieren, ist das unmöglich. Also brauchen wir eine Möglichkeit, diese Verteilung wieder zu zentrieren. Wir können die erzeugten zufälligen Bits erst mit dem sogenannten *Newton-Filter* filtern. Das Prinzip ist simpel. Wir generieren zwei zufällige Bits a und b. Sind diese gleich, so verwerfen wir sie. Sind sie unterschiedlich, so verwerfen wir b und nehmen a als Zufallsbit. Dieser Filter funktioniert allerdings nur dann, wenn die Verteilung konstant bleibt, also beispielsweise, wenn wir konstant doppelt so viele Einsen wie Nullen erhalten. Die Ausgabe ist dann zentrisch verteilt.

Es muss darauf hingewiesen werden, dass der oben vorgestellte 2. Generator bereits zentrisch verteilt ist. Ist n prim, so ist die Anzahl der geraden Zahlen in Z[n] genau um eins höher als die Anzahl der ungeraden Zahlen. Wir liefern 0, wenn die Zahl gerade ist, ansonsten 1. Nach jeder Zufallszahl ersetzen wir p durch p + 1 (mod n).
Ist danach p = 0, so setzen wir p auf 1. Dadurch verbannen wir eine gerade Zahl (nämlich 0, da 0^e = 0 (mod n)) aus der Menge und haben somit genau so viele gerade wie ungerade Zahlen. Der achtsame Leser dürfte bereits festgestellt haben, dass diese Symmetrie verloren gehen könnte, wenn wir p nach Systemereignissen ändern. Die Wahrscheinlichkeit dazu ist jedoch sehr gering, denn selbst, wenn p nur auf geraden Zahlen landet, ist p^e (mod n) relativ

zufällig. Es muss nur darauf geachtet werden, dass p auf jeder Zahl in Z[n] (außer 0) landen kann.

In einigen seltenen Fällen ist es eher wünschenswert, dass die Verteilung nicht zentrisch ist. Doch in diesem Fall sollten wir ein gewisses Maß an Kontrolle über die Verteilung behalten. Also zentrieren wir die Verteilung erst mit dem Newton-Filter, sofern notwendig. Dann überlegen wir uns, wie die Verteilung aussehen soll. Beispielsweise sollen durchschnittlich doppelt so viele Einsen wie Nullen generiert werden. Das erreichen wir ganz einfach dadurch, dass wir jede zweite Null verwerfen. Damit haben wir nur noch die Hälfte an Nullen. Wichtig ist, anzumerken, dass wir nicht jede Eins doppelt zurückgeben sollten, denn sonst können immer nur zwei aufeinanderfolgende Einsen generiert werden und der Zufall nimmt stark ab.

2. Informative Zufallszahlen

Die Ausgabe vieler älterer Zufallszahlengeneratoren lässt sich auf den geheimen Zustand zurückführen. Als Beispiel lässt sich das weiter oben aufgeführte Exemplar eines unsicheren Generators nehmen. Da wir uns im 2. vorgestellten Generator auf pure Zahlentheorie verlassen, wissen wir bereits, dass die Ausgabe keine Informationen über die Eingabe liefert (sonst wäre RSA unsicher). Alle anderen Generatoren verlassen sich auf Hash- oder Chiffreverfahren. Die Sicherheit hängt also von diesen ab.

Zufallszahlen sind also ein deutlich umfangreicheres Thema, als viele annehmen. Es wurden bereits ganze Bücher über dieses Thema geschrieben. In diesem Text kann dem Leser nur eine kleine Einführung geboten werden, denn eine intensive Beschäftigung mit Zufallszahlen würde den Rahmen dieser Dokumentation sprengen.

Es gibt übrigens einen ganz interessanten Trick, mit dem man die Qualität eines Zufallszahlengenerators testen kann. Echte Zufallszahlen lassen sich nicht komprimieren. Also sucht man ein gutes Kompressionsprogramm wie bzip2 und versucht, eine Datei, die die Ausgabe eines Zufallszahlengenerators enthält, zu komprimieren. Im Optimalfall ist keine

Kompression möglich und die komprimierte Version ist genau so groß wie die unkomprimierte Version (z.T. sogar größer, wegen Metainformationen vom Kompressionsprogramm). Sobald die komprimierte Version kleiner als die unkomprimierte ist, ist das ein Zeichen dafür, dass eine bestimmte längere Folge von Zahlen wiederholt wurde oder das Kompressionsprogramm ein System in der Zahlenfolge entdecken konnte. Es sollten möglichst lange Folgen generiert werden. Am besten lässt man, sofern möglich, eine komplette Periode des Zufallszahlengenerators durchlaufen.

Arithmetik mit übergroßen Zahlen

Neben den ganzen kryptologischen und mathematischen Problemen, mit denen wir bei der Arbeit mit RSA konfrontiert werden, haben wir noch ein computertechnisches Problem. Das Problem an diesem Problem ist, im wahrsten Sinne des Wortes, die Größe des Problems. Es ist so groß, dass heutige Privatcomputer damit nicht umgehen können. Wir benötigen gigantische Zahlen. Die meisten Benutzer besitzen jedoch noch 32 Bit Computer, die mit Zahlen größer als 2^32 - 1 nicht klar kommen. Der Programmierer kennt dieses Problem. Glücklicherweise können wir jedoch die Unterstützung für solche Zahlen emulieren. Gehen wir nun von einem 8 Bit Computer aus, auf dem wir Arithmetik mit 32 Bit Zahlen durchführen wollen. Wir erstellen also vier solche 8 Bit Einheiten. Unsere Variablen haben also ein Subskript von 0 bis 3. 0 gibt dabei die niederwertigste Stelle und 3 die höchstwertige an. Heißt unsere Variable also v[], so hat sie folgende Komponenten:

```
v[0] (mod 256)  // Vielfaches von 256^0
v[1] (mod 256)  // Vielfaches von 256^1
v[2] (mod 256)  // Vielfaches von 256^2
v[3] (mod 256)  // Vielfaches von 256^3
```

Alle vier Komponenten von v[] haben einen Wert von 0 bis 255. Der Wert von v[] selbst lässt sich folgendermaßen berechnen:

```
v[] = v[0] + v[1] * 256 + v[2] * 256^2 + v[3] * 256^3
```

Die höchste darstellbare Zahl ist also:

```
  255 + 255 * 256 + 255 * 256^2 + 255 * 256^3
= 2^32 - 1
= 4294967295
```

Mit diesen Variablen können wir nun rechnen. Doch, bevor wir damit anfangen, müssen wir erst einmal verstehen, wie ein Computer mit Zahlen umgeht. Zunächst ist sehr wichtig, anzumerken, dass ein Computer negative Zahlen tatsächlich nicht kennt. Wir müssen jedoch nicht auf sie verzichten. Interessant ist die Implementierung von negativen Zahlen, denn sie sind völlig implizit. Dazu sehen wir uns erst einmal an, wie so ein 8 Bit Computer rechnet. Er führt alle Rechenoperationen in der Menge Z[2^8], also Z[256] durch. Wenn wir also 7 von 13 subtrahieren möchten, addieren wir das additive Inverse von 7 zu 13, also:

```
  13 - 7 (mod 256)
= 13 + -7 (mod 256)
= 13 + (256 - 7) (mod 256)
= 13 + 249 (mod 256)
= 6 (mod 256)
```

Es stellt sich heraus, dass wir echte negative Zahlen gar nicht brauchen. Wir erklären alle Zahlen, deren höchstes Bit gesetzt ist, zu negativen Zahlen, also alle Zahlen größer oder gleich 128. Um die negative Zahl -15 zu speichern, speichern wir also in Wirklichkeit 256 - 15 = 241. Also ist 241 unsere Darstellung von -15. Dass die Zahl negativ ist, erkennen wir am höchsten Bit. Ist es gesetzt, ist also die Zahl größer oder gleich 128 (2^7), dann ist die Zahl negativ. Dieses Vorgehen wird bei der Multiplikation und Division allerdings problematisch, da die negativen Zahlen in dem Fall nicht implizit sind. Und wir haben noch ein zweites Problem. Wir können eine negative Zahl -128 darstellen, aber keine positive Zahl 128, denn 128 ist schon die negative Darstellung für -128 (Erinnerung: bei 128 ist das höchste Bit gesetzt, die Zahl ist größer oder gleich 128 und damit negativ). Mehr dazu später.

Um nun mit diesen Zahlen rechnen zu können, betrachten wir sie etwas näher. Dazu versetzen wir uns kurz zurück in die Zeit in der Grundschule. Wir Menschen sind genau so begrenzt wie Computer. Zweistellige Zahlen addieren wir problemlos, sofern wir keine Konzentrationsprobleme haben. Doch schon bei dreistelligen Zahlen wird es bei manchen schwierig. Bei zwanzigstelligen Zahlen hört es auf. Um solche Zahlen selbst zu addieren, bedienen wir uns einem Blatt Papier und einem Stift. Das nennen wir *schriftliches Addieren*. Wenn wir schriftlich addieren, arbeiten wir im 10er

Zahlensystem. Wir haben also Ziffern von 0 bis 9. Mit den oben genannten Vektorvariablen arbeiten wir genau nach dem selben Prinzip, nur im 256er Zahlensystem. Wir haben also insgesamt 256 verschiedene "Ziffern", statt nur 10. Wir könnten auch im 10er Zahlensystem arbeiten, dann hätten die Komponenten der Variable v[] einen Wertebereich von 0 bis 9, statt 0 bis 255. Damit könnten wir vierstellige Dezimalzahlen in so einer Vektorvariablen darstellen. Wir möchten den Computer jedoch vollständig ausreizen und bedienen uns daher dem 256er Zahlensystem.

Genau so wie der Mensch mit zu großen Zahlen schriftlich rechnen kann, kann es auch der Computer und genau auf diese Art implementieren wir große Zahlen. Hier werden nun alle Grundrechenarten vorgestellt. So wie wir im Schulheft beim schriftlichen Addieren in der Menge Z[10] rechnen, so rechnen wir hier in der Menge Z[256], denn unsere Variablenkomponenten sind 8 Bit breit. Hier ist auch wichtig, anzumerken, dass das völlig implizit abläuft. Wir müssen die modulo-Operation also nicht selbst ausführen, denn wir können so oder so nicht mehr als 8 Bits speichern.

1. Addition

```
c[] = a[] + b[]
```

Um a[] und b[] zu addieren, addieren wir ausgehend von der niederwertigsten Komponente aufwärts bis zur höchsten und übertragen dabei immer den Übertrag. Der Übertrag ist der durch die modulare Arithmetik verloren gegangene Anteil der Zahl, also der Quotient. Dieser kann nicht größer als 1 sein. Zunächst setzen wir alle Komponenten von c[] auf 0. Danach berechnen wir c[0] = a[0] + b[0]. Nun prüfen wir, ob ein Übertrag stattfand. Ist die Summe a[0] + b[0] größer als 255, also außerhalb von Z[256], so fand ein Übertrag statt und wir setzen c[1] auf 1. Danach addieren wir a[1] + b[1] zu c[1] und führen wieder eine Übertragsprüfung durch. Ein Übertrag findet statt, wenn a[1] + b[1] + c[1] größer als 255 ist. In dem Fall setzen wir c[2] auf 1. Das führen wir fort bis zur höchstwertigen Komponente. Bei dieser brauchen wir keine Übertragsprüfung mehr durchführen, denn wir sind bereits an der Grenze angelangt. Beispiel:

```
a[] = { 42, 173, 26, 218 } = 3659181354
b[] = { 63, 162, 52,  17 } = 288662079

c[0] = a[0] + b[0]     =  42 +  63     = 105  // Kein Übertrag
```

```
c[1] = a[1] + b[1]     = 173 + 162     =  79  // Hier ein Übertrag
c[2] = a[2] + b[2] + 1 =  26 +  52 + 1 =  79  // Kein Übertrag
c[3] = a[3] + b[3]     = 218 +  17     = 235

c[] = { 105, 79, 79, 235 } = 3947843433
```

Die Addition wurde also im 256er-Zahlensystem "schriftlich" durchgeführt, wie wir das normalerweise im Dezimalsystem von Hand gemacht hätten.

2. Subtraktion

```
c[] = a[] - b[]
```

Die Subtraktion können wir ähnlich wie die Addition durchführen. Um b[] von a[] zu subtrahieren, subtrahieren wir, beginnend mit der niederwertigsten Komponente, schrittweise und führen dabei wieder eine Übertragsprüfung durch. Zunächst setzen wir c[] auf 0 und berechnen: c[0] = a[0] + -b[0]. Wir müssen also erst das additive Inverse zu b[0] ermitteln. In den meisten Programmiersprachen lässt sich dies durch eine ganz normale Subtraktion durchführen, aber wir bleiben hier bei der Zahlentheorie und damit plattformunabhängig. Das additive Inverse ist 256 - b[0] (mod 256), denn wir rechnen in der Menge Z[256]. Ein Übertrag fand statt, wenn b[0] größer als a[0] ist. Es gibt nun zwei Wege, den Übertrag durchzuführen. Entweder, wir manipulieren b[], oder wir manipulieren c[], wobei die Manipulation von b[] deutlich einfacher und schneller ist. Hier wird der Übertrag über die Manipulation von b[] vorgeführt. Fand bei der ersten Komponente (0) ein Übertrag statt, so addieren wir 1 zu b[1]. Bei dieser Addition könnte erneut ein Übertrag stattfinden. Wir benötigen also eine innere Schleife, die den Übertrag weiterreicht, bis kein Übertrag mehr stattfindet. Danach ist die nächste Komponente an der Reihe. Wir berechnen c[1] = a[1] + -b[1]. Fand ein Übertrag statt, so addieren wir 1 zu b[2]. Fand hier auch ein Übertrag statt, so übertragen wir weiter auf b[3]. Das führen wir bis zur letzten Komponente durch, bei der wir allerdings keine Übertragsprüfung mehr durchführen brauchen, denn der Übertrag landet im Nirvana. Hier ein Subtraktionsbeispiel:

```
a[] = { 62, 74, 251, 164 } = 2767931966
b[] = { 84, 12,  63, 153 } = 2571045972

c[0] = a[0] - b[0] =  62 +  -84 = 234  // Übertrag, wir inkrementieren b[1]
c[1] = a[1] - b[1] =  74 +  -13 =  61  // Kein Übertrag
```

```
c[2] = a[2] - b[2] = 251 +  -63 = 188  // Kein Übertrag
c[3] = a[3] - b[3] = 164 + -153 =  11

c[] = { 234, 61, 188, 11 } = 196885994
```

Das additive Inverse von x in Z[256] ist
256 - x. Hier stehen wir vor einem Problem, denn wir können diese Subtraktion nicht durchführen, da wir die Zahl 256 mit 8 Bits nicht darstellen können. Das Problem ist aber schnell gelöst, wenn wir uns die wichtigste Kongruenz der modularen Arithmetik ansehen:

```
// k beliebig
k * n = 0 (mod n)
```

Also können wir auch ganz einfach 0 - x berechnen. Das ist gleichbedeutend mit 256 - x, wenn uns 8 Bits zur Verfügung stehen, wir also in der Menge Z[2^8] arbeiten. Es soll jedoch erwähnt werden, dass diese Subtraktion arithmetisch gesehen nicht korrekt ist; sie ist nicht definiert. Wir haben aber das Glück, dass sie alle Computer dennoch richtig ausführen.

3. Schiebeoperationen

```
c[] = a[] ‹‹ b
c[] = a[] ›› b
```

Hier wird eine neue Operation eingeführt, die die Multiplikation mit Zweierpotenzen immens beschleunigt. Ein PC mit i486-Architektur braucht zum Multiplizieren beispielsweise etwa vierzig mal so lang wie zum Schieben! Das Prinzip ist sehr einfach. Haben wir im 10er-Zahlensystem (also im Dezimalsystem) eine Zahl 63, so können wir sie problemlos mit 10 multiplizieren, indem wir einfach eine 0 anhängen. Das Resultat ist 630. Wir können die Zahl auch problemlos durch 10 teilen, indem wir einfach die niederwertigste Ziffer entfernen (Erinnerung: wir arbeiten nur mit Ganzzahlen). Das Resultat ist also 6. Anders ausgedrückt: wir haben bei der Multiplikation die Ziffern nach links geschoben und an der frei gewordenen Stelle eine 0 platziert. Bei der Division haben wir die Ziffern nach rechts geschoben. Dabei wurde die niederwertigste Stelle hinter das imaginäre Komma geschoben und ist verschwunden, da wir ja nur Ganzzahlen haben. Mit dieser Schiebeoperation können wir, ohne rechnen zu müssen, mit jeder Potenz von 10 multiplizieren. Ist der Exponent negativ, so dividieren wir. Um mit 10^3, also 1000, zu multiplizieren, schieben wir einfach drei mal nach links. Um mit 10^(-4) zu multiplizieren, also durch 10000 zu dividieren, schieben wir

einfach vier mal nach rechts.

Diese Schiebeoperationen sind in jedem Zahlensystem gültig, also auch im Dualsystem. Um eine Dualzahl 1101011 (dez. 107) mit 10 (dez. 2) zu multiplizieren, schieben wir einmal nach links und erhalten 11010110 (dez. 214). Um durch 10 (dez. 2) zu dividieren, schieben wir einmal nach rechts und erhalten 110101 (dez. 53). Auch in diesem Zahlensystem kann man also, ohne rechnen zu müssen, mit jeder Potenz von 10 (dez. 2) multiplizieren. Um mit 10^1011 (dez. 2^11) zu multiplizieren, schieben wir 1011 (dez. 11) mal nach links und so weiter. Die Schiebeoperation im Dualsystem wird als *bitweise Schiebeoperation* (engl. bitwise shift operation) bezeichnet und kann durch den Computer sehr effizient durchgeführt werden. In C und C++ gibt es zum Linksschieben den Operator ‹‹ und zum Rechtsschieben den Operator ››. a ‹‹ b heißt also: a um b Stellen nach links schieben. Dabei handelt es sich immer um die bitweise Schiebeoperation. In den meisten anderen Programmiersprachen gibt es ähnliche Operatoren, die den gleichen Zweck erfüllen.

Die Linksschiebeoperation ist sehr simpel. Hier beginnen wir bei der höchstwertigen Komponente, in unserem Fall 3. Zunächst berechnen wir c[3] = (a[3] ‹‹ b) + (a[2] ›› (8 - b)). Damit schieben wir die aus a[2] herausgeschobenen Bits wieder in c[3] rein. Die 8 ist dabei die Anzahl der Bits. Dann berechnen wir c[2] = (a[2] ‹‹ b) + (a[1] ›› (8 - b)). Das führen wir fort, bis wir an der niederwertigsten Stelle angelangt sind. Dort können wir die Rechnung etwas abkürzen: c[0] = a[0] ‹‹ b. An der niederwertigsten Stelle gibt es nämlich nichts mehr hineinzuschieben.

Die Rechtsschiebeoperation verläuft genau anders herum. Wir beginnen mit der niederwertigsten Stelle, also 0 und berechnen: c[0] = (a[0] ›› b) + (a[1] ‹‹ (8 - b)). Hier schieben wir also erst b mal nach rechts und schieben danach die aus a[1] herausgeschobenen Bits in a[0] wieder hinein. Auch das führen wir fort, bis zur höchstwertigen Stelle, an der wir die Operation wieder simplifizieren: c[3] = a[3] ›› b.

Es sollten also bei Multiplikationen mit Zweierpotenzen immer die Schiebeoperationen eingesetzt werden, sofern möglich. Bei den Schiebeoperationen ist auf das Vorzeichen zu achten. Diese Schiebeoperationen lassen sich nur auf nicht negative Zahlen ausführen. Um negative Zahlen auf diese Art mit Zweierpotenzen zu multiplizieren, müssen wir sie vor der Schiebeoperation erst invertieren und nach der Schiebeoperation nochmal.

4. Multiplikation

```
c[] = a[] * b[]
```

Hier gibt es mehrere Wege. Für große Zahlen ist wohl der schulische Weg am besten geeignet. Hier soll jedoch ein Ansatz vorgestellt werden, der der modularen Exponentiation sehr ähnelt und für kleinere Zahlen sehr schnell durchführbar ist.

Zunächst stellen wir sicher, dass b[] kleiner als a[] ist. Falls das nicht der Fall ist, tauschen wir a[] und b[] aus. Danach machen wir uns die Schiebeoperationen zunutze, um a[] * b[] schnell ausrechnen zu können. Auch hier arbeiten wir nach einem sehr simplen Prinzip. Die Idee soll zunächst mit skalaren Werten verdeutlicht werden. Um zwei Zahlen x und y zu multiplizieren, gehen wir folgendermaßen vor. Ist y gerade, so führen wir x * y auf (2 * x) * (y / 2) zurück. Ist y ungerade, so führen wir x * y auf x * (y - 1) + x zurück. Nehmen wir als Beispiel die Multiplikation 634 * 736:

```
  x * y
= 634 * 736   // y ist gerade, also verdoppeln wir x und halbieren y
= 1268 * 368
= 2536 * 184
= 5072 * 92
= 10144 * 46
= 20288 * 23  // Hier wird y ungerade; wir subtrahieren 1 von y und addieren x
= 20288 * 22 + 20288
= 40576 * 11 + 20288
= 40576 * 10 + 40576 + 20288
= 81152 * 5 + 40576 + 20288
= 81152 * 4 + 81152 + 40576 + 20288
```

```
= 162304 * 2 + 81152 + 40576 + 20288
= 324608 * 1 + 81152 + 40576 + 20288
= 324608 + 81152 + 40576 + 20288
= 466624
```

Wir haben also die ursprüngliche Multiplikation in eine Reihe Additionen überführt. Bis dahin haben wir nur die Subtraktion und die Schiebeoperationen gebraucht, denn wir haben immer nur mit zwei multipliziert und dividiert. Hinterher haben wir noch einmal addiert. Das heißt, alle Operationen, die wir benötigen, stehen uns bereits zur Verfügung und wir müssen keine einzige echte Multiplikation oder Division durchführen. Da dieses Verfahren jedoch wiederholt auf der gesamten Zahl arbeitet, eignet es sich eher für kleinere Zahlen. Für große Zahlen sollte der normale schulische Weg eingesetzt werden.

Ähnlich wie bei der modularen Exponentiation führen wir eine zusätzliche Variable r[], die wir mit 0 initialisieren. Ist b[] gerade, so verdoppeln wir a[] und halbieren b[]. Ist b[] ungerade, so addieren wir a[] zu r[] und ersetzen b[0] durch b[0] - 1 (wir brauchen für b[] also nicht mal die Additionsfunktion aufrufen, da wir nur eine einzige Komponente von b[] manipulieren und, da b[] auf jeden Fall ungerade ist, wird auch garantiert kein Übertrag stattfinden). Die Schleife endet, sobald b[] = 0 ist. Dann enthält r[] das Ergebnis. Beispiel (Multipliaktion 527 * 373):

```
   a[]     b[]     r[]
------  ------  ------
   527     373       0  b[] ist ungerade; r[] = r[] + a[]; b[] = b[] - 1
   527     372     527  b[] ist gerade; a[] = a[] * 2; b[] = b[] / 2
  1054     186     527  b[] ist gerade
  2108      93     527  b[] ist ungerade
  2108      92    2635  b[] ist gerade
  4216      46    2635  b[] ist gerade
  8432      23    2635  b[] ist ungerade
  8432      22   11067  b[] ist gerade
 16864      11   11067  b[] ist ungerade
 16864      10   27931  b[] ist gerade
 33728       5   27931  b[] ist ungerade
 33728       4   61659  b[] ist gerade
 67456       2   61659  b[] ist gerade
134912       1   61659  b[] ist ungerade
134912       0  196571  b[] ist 0; r[] enthält das gewünschte Ergebnis
```

Dieser Multiplikationsalgorithmus lässt sich übrigens sehr stark optimieren. Wir lassen eine innere Schleife laufen, die wiederholt a[] verdoppelt und b[] halbiert, bis b[] ungerade wird. Ob b[] gerade ist, sehen wir am niederwertigsten Bit, also b[0] (mod 2). Ist es 0, so ist b[] gerade. Auch das lässt sich durch eine Bitoperation optimieren: b[0] and 1. Nachdem diese innere Schleife verlassen wird, ist garantiert, dass b[] ungerade ist. Also addieren wir a[] zu r[] und löschen das unterste Bit von b[] durch eine XOR-Operation. Das ist gleichbedeutend mit der Subtraktion von 1. Danach ist b[] garantiert gerade. Ein Übertrag kann nicht stattfinden, da die kleinste ungerade Zahl 1 ist. Ist b[] also ungleich 0, so betreten wir wieder die innere Schleife, ansonsten ist der Algorithmus abgeschlossen und r[] enthält unser Ergebnis.

Hier muss unbedingt angemerkt werden, dass dieser Algorithmus nur funktioniert, wenn b[] nicht negativ ist. Das ist aber kein Problem. Zunächst überprüfen wir, ob der Betrag von a[] kleiner als der Betrag von b[] ist. Wenn ja, vertauschen wir a[] und b[]. Danach überprüfen wir das Vorzeichen von b[], also das höchstwertige Bit. Ist es gesetzt, ist b[] negativ. In diesem Fall invertieren wir einfach die Vorzeichen von a[] und b[], ersetzen also a[] durch -a[] und b[] durch -b[]. a[] darf ruhig negativ sein.

Steht eine Multiplikation an, sollten die Größen der Faktoren überprüft werden. Übersteigen sie den Grenzwert, an dem die schulische Variante schneller arbeitet, sollte diese genutzt werden. Es gibt übrigens noch viele weitere Verfahren. Eine kurze Suche im Internet sollte hier hilfreich sein.

5. Division und Modulo

```
c[] = a[] / b[]
c[] = a[] mod b[]
```

Die Division ist ein wenig komplizierter. Es gibt viele Lösungsansätze. Die momentan schnellste Lösung für große Zahlen ist das Schätzverfahren von Knuth. Der Leser wird hier auf externe Literatur verwiesen. Jede dieser

Lösungsansätze hat ihre Geschwindigkeitsgrenzen. Für kleinere Zahlen sind andere Divisionsalgorithmen zu empfehlen. Hier soll nun ein solches Verfahren vorgestellt werden.

Wir subtrahieren von a[] so lange Vielfache von b[], bis a[] kleiner als b[] ist und zählen dabei, wie oft wir b[] von a[] subtrahiert haben. Sobald a[] kleiner als b[] ist, enthält der Zähler den Quotienten und a[] ist dann der Divisionsrest. Dazu führen wir drei zusätzliche Variable m[], f[] und r[]. Die Variable m[] enthält immer ein Vielfaches von b[] und f[] dokumentiert den Faktor, mit dem wir b[] multipliziert haben. r[] ist der Zählwert, der dokumentiert, wie oft wir b[] von a[] subtrahiert haben. Die Variablen werden folgendermaßen initialisiert:

```
m[] = b[]
f[] = 1
r[] = 0
```

Nun schieben wir m[] und f[] so lange nach links, bis wir den größten Wert m[] kleiner oder gleich a[] erreicht haben. Jetzt subtrahieren wir m[] von a[] und addieren den Wert von f[] zu r[]. Ist a[] danach kleiner als m[], so schieben wir m[] und f[] so lange nach rechts, bis m[] wieder kleiner oder gleich a[] ist. Dann beginnen wir die Schleife von vorn. Die Schleife endet, sobald f[] den Wert 0 erreicht. Hier ein Beispiel (Division 373457 / 9456):

```
// Ausgangszustand:
a[] = 373457
b[] = 9456
m[] = 9456
f[] = 1
r[] = 0

// Jetzt schieben wir m[] und f[] nach links, bis wir den größten Wert m[]
// kleiner als a[] erreicht haben:

   a[]     m[]     f[]
------  ------  ------
373457    9456       1
373457   18912       2
373457   37824       4
```

```
373457   75648       8
373457  151296      16
373457  302592      32

// Nun betreten wir den Algorithmus:

   a[]     m[]  f[]   r[]
------  ------  ---  ----
373457  302592   32     0  m <= a, also: a - m, r + f
 70865  302592   32    32  m > a, also: m / 2, f / 2
 70865  151296   16    32  m > a
 70865   75648    8    32  m > a
 70865   37824    4    32  m <= a
 33041   37824    4    36  m > a
 33041   18912    2    36  m <= a
 14129   18912    2    38  m > a
 14129    9456    1    38  m <= a
  4673    9456    1    39  a < b; r ist der Quotient und a der Rest
```

Je kleiner der Wert für b[] ist, desto länger dauert die Schleife. Sie lässt sich dadurch etwas optimieren, dass man die Startwerte für m[] und f[] innerhalb der Hauptschleife ermittelt. So laufen beide Prozesse gleichzeitig. Der Geschwindigkeitsunterschied ist jedoch nicht sehr groß und hängt allein vom Wert für b[] ab. b[] darf übrigens nicht 0 sein, denn sonst leiten wir eine Endlosschleife ein. Das ist jedoch kein Problem, denn durch 0 können wir ohnehin nicht teilen.

Dieser Algorithmus kommt nicht mit negativen Zahlen klar. Sowohl a[], als auch b[] müssen positiv sein. a[] darf auch 0 sein. In dem Fall bricht der Algorithmus sofort ab und liefert 0 als Quotient und Rest. Das Vorzeichen des Quotienten und des Rests sollte in einer temporären Variablen festgehalten werden. Unterscheiden sich die Vorzeichen von a[] und b[], so wird die temporäre Variable auf 1 gesetzt, ansonsten auf 0. Danach werden a[] und b[] durch ihre jeweiligen Beträge ersetzt. Nachdem der Algorithmus durchgelaufen ist, wird überprüft, ob die temporäre Variable 1 ist. Wenn ja, wird der Quotient invertiert. Der Rest berechnet sich ein wenig anders. Das Vorzeichen von b[] spielt in dem Fall keine Rolle. War a[] negativ, so kann man entweder einen negativen Rest zurückgeben oder etwas effektiver vorgehen und den Rest a[] durch b[] - a[] ersetzen. Dadurch muss der Benutzer nicht erst das additive Inverse zum Divisionsrest berechnen.

Ähnlich wie bei der Multiplikation sollte hier die Größe der Operanden festgestellt werden. Übersteigt sie einen gewissen Grenzwert, sollte ein anderer Algorithmus wie z.B. der weiter oben erwähnte Knuth-Algorithmus eingesetzt werden.

TO DO: Weitere Rechenoperationen, Tücken, Tipps und Tricks.

Wie geht es weiter?

Die Zukunft von RSA ist kaum vorhersagbar. Es könnte jederzeit ein Verfahren entdeckt werden, das die Sicherheit von RSA komplett vernichtet. Da RSA in der Kryptographie eine sehr wichtige Rolle spielt, wird es exzessiv studiert. Es ist Tatsache, dass die Sicherheit von RSA von sehr vielen unbestätigten Sätzen abhängt. Niemand weiß, ob wir jemals einen Weg finden werden, große Zahlen zu faktorisieren. Niemand kann sagen, ob wir nicht irgendwann in der Lage sein werden, auf schnellem Wege die modulare Wurzel zu ziehen. Obwohl Primzahlen seit mehreren Tausend Jahren studiert werden, steckt die Primzahltheorie gewissermaßen noch in den Kinderschuhen.

Inzwischen wurde RSA schon durch weitere Verfahren ersetzt, die aber unter genau den selben Lücken und zusätzlich noch weiteren leiden. Ein Beispiel ist die Arithmetik in der Menge der elliptischen Kurve. Elliptische Kurven sind ein sehr umfangreiches Thema; weitaus umfangreicher als RSA. Sie erlauben jedoch eine ähnliche Sicherheit wie bei RSA, aber mit viel kleineren Schlüsseln und deutlich schnelleren Rechenoperationen. Sie werden allerdings erst seit wenigen Jahrzehnten studiert und es gibt viel mehr unbestätigte Sätze als im RSA-Verfahren. Auch diesem Verfahren werde ich demnächst einen Text widmen.

Da der Mensch sehr naiv ist, verlässt er sich auf seinen Instinkt und seine Einschätzungen. Er baut und verwendet Dinge, mit denen er nicht umgehen kann. Das hebt die Natur des Menschen als Tier hervor. Bezogen auf den Menschen ist die Sicherheit ein Widerspruch in sich. Wer weiß, ob wir damit nicht irgendwann schmerzhaft auf die Schnauze fallen werden.

"Ich weiß nicht, welche Waffen beim dritten Weltkrieg zum Einsatz kommen. Beim vierten werden wir mit Pfeil und Bogen kämpfen." (Albert Einstein)

Nachwort

In diesem Text wurden die Funktionsweise und die Stärken und Schwächen von RSA vorgeführt. Es zeigt sich deutlich, dass jeder auf einem Blatt Papier mit einem Bleistift einen kleineren RSA-Schlüssel erstellen kann. Er kann auch die notwendigen Rechnungen alle auf diesem Blatt Papier durchführen, da die modulare Arithmetik alle Berechnungen extrem vereinfacht.

Die Stärke von RSA ist, dass es jeder verstehen und anwenden kann. Das Verfahren ist eine einzige simple mathematische Operation. Hauptschulwissen reicht aus, um RSA zu verstehen, sofern man ein wenig im logischen Denken geübt ist.

Die wichtigste Schwäche von RSA ist, dass die Sicherheit von RSA noch nicht einmal bewiesen ist. Es ist auch sehr unwahrscheinlich, dass sich diese jemals beweisen lassen wird. Falls jemals jemand einen Weg findet, große Zahlen direkt zu faktorisieren, ist RSA wertlos und die Sicherheit im halben Internet bricht zusammen, da inzwischen so gut wie jeder auf die Sicherheit von RSA setzt. Auch die Regierung und vor allem das Militär.

Eine weitere Schwäche ist, dass man extrem vorsichtig, geradezu paranoid programmieren muss. Welche Folgen ein falscher RSA-Schlüssel haben kann, wurde im Kapitel über Tücken des RSA-Verfahrens verdeutlicht. Es verbergen sich jede Menge solcher Gefahren in RSA. Man muss sehr vorsichtig programmieren und das Verfahren sehr gut kennen, um diese Tücken zu umgehen. Logisches Denken wird hier verlangt. Die letzte erwähnenswerte Schwäche von RSA ist die Performance. Wir müssen mit gigantischen Zahlen rechnen. Zahlen, die heutige Privatcomputer nicht unterstützen. Das heißt, wir müssen die Unterstützung für solch große Zahlen softwaremäßig emulieren. Dies ist natürlich wesentlich langsamer als eine native Unterstützung solch groß dimensionierter Zahlen durch den Prozessor. Außerdem ist eine solche Implementierung nicht unbedingt simpel und sehr fehleranfällig. Vor allem, wenn man mit dem Ziel, schnellen Code zu produzieren, programmiert.

Zusammenfassend kann man sagen, dass RSA heute als sicher eingeschätzt werden kann. Es ist jedoch nur dann sicher, wenn man eben vorsichtig programmiert. Dabei zählen sehr viele Faktoren, beispielsweise die Generierung von Zufallszahlen. Es wird auch nur so lange sicher bleiben, wie es keiner schafft, große Zahlen schnell zu faktorisieren. Dabei ist die größte Hürde, dass die Zeit zum Finden großer Primzahlen linear mit der benötigten Größe steigt, die Faktorisierung jedoch exponentiell. Sollte diese Hürde jemals überwunden werden, ist RSA nur noch ein mathematischer Satz wie jeder andere und nicht mehr zur Verschlüsselung geeignet.

Mit diesen Worten schließe ich diese Dokumentation ab und wünsche dem Leser viel Spaß und viel Erfolg beim Ausprobieren, Programmieren, Lernen und Verstehen.

Wissen ist Macht - Euer mm_freak.

Referenzen

Hier eine Zusammenstellung von Webseiten mit weiterführenden Informationen. Diese Liste ist noch relativ unvollständig. Informationen über weitere Referenzen, die hier angebracht wären, bitte an den Autor richten.

Bruce Schneier
Die Homepage des bekannten amerikanischen Kryptologen Bruce Schneier. Dort finden sich Informationen zu momentanen kryptographischen Verfahren und aktuellen Arbeiten.
GNU Privacy Guard (gnupg)
Populärste freie Implementierung der PGP-Mechanismen zur Verschlüsselung und Signatur von Emails.
OpenSSH
Open-Source Implementierung des bekannten Secure Shell (SSH) Protokolls, dem verschlüsselten Ersatz für Telnet.
OpenSSL
Die populärste und am weitesten verbreitete freie Implementierung der Secure Sockets Layer (SSL) und Transport Layer Security (TLS) Protokolle. OpenSSL enthält auch eine umfangreiche kryptographische Bibliothek (libcrypto) für Programmierer.
RSA Security
Homepage von der RSA Security Inc. Dort finden sich aktuellste Arbeiten in Bezug auf asymmetrische Verfahren. RSA Security veranstaltet auch immer wieder Faktorisierungswettbewerbe.
DDDDFF

Kontakt

Der Autor ist zur Zeit nur im IRC-Channel von
Team Unix
unter dem Nickname *mm_freak* zu erreichen:
#teamunix auf irc.phat-net.de.
Man benötigt einen der vielen verfügbaren IRC-Clients, um dort hin zu gelangen, z.B.
X-Chat (GUI) oder
irssi (konsolenbasiert). Mozilla- und Netscape-Benutzer können auch einfach auf den obigen Link klicken und gelangen sofort zum Chatraum.

Die angegebene Email-Adresse des Autors wird momentan (vorübergehend) nicht verwendet. Emails an diese Adresse werden verloren gehen. Im Übrigen lohnt sich auch ein Blick auf die Homepage von Team Unix.
Dort finden sich weitere interessante Dokumente, ein Forum und ein Wiki für jedermann.

Fehlerberichte und Verbesserungsvorschläge bitte anmm_freak ‹never@drwxr-xr-x.org›.

Vielen Dank.

Eindeutige ID: #1194
huschi
2006-09-19 12:43