EIN PUBLIC-KEY-KRYPTOSYSTEM

T. Hürlimann

Working Paper 94-03

Januar 1994

corrections Feb 94

INSTITUT D'INFORMATIQUE, UNIVERSITE DE FRIBOURG

INSTITUT FÜR INFORMATIK DER UNIVERSITÄT FREIBURG

Institute of Informatics, University of Fribourg, site Regina Mundi

rue de Faucigny 2

CH-1700 Fribourg / Switzerland

Bitnet: HURLIMANN@CFRUNI51

phone: (41) 37 21 95 60 fax: 37 21 96 70

Ein Public-Key-Kryptosystem

Tony Hürlimann, Dr. rer. pol.

Key-words: prime numbers, RSA, cryptography

Abstract: This paper gives a introduction into the RSA-public-key cryptosystems. A complete software package has been developed and implemented, which might be useful for educational purposes. All operations concerning huge integers have also been implemented (in Pascal). Furthermore, a brief idea will be given about the modern theory of prime numbers, which is a fascinating realm of the theory of numbers.

Stichworte: Primezahlen, RSA-Kryptosysteme

Zusammenfassung: Dieser Artikel führt systematisch in das Gebiet der RSA-Public-Key-Kyptosysteme ein. Es wurde eine vollständiges Softwarepaket entwickelt, welches für den Informatikunterricht geeignet sein könnte, um die verschiedenen Aspekte zu beleuchten. Alle dazu notwendigen Operationen auf grossen Zahlen wurden ebenfalls implementiert (in Pascal). Es sollen auch einige Eindrücke vermittelt werden von der gegenwärtigen Forschung der Primzahlentheorie, einem faszinierenden Gebiet der Zahlentheorie, auf dem vor 20 Jahren nur eine Handvoll Mathematiker gearbeitet haben.

> “So kann man denn nie wissen – die wichtigsten Beziehungen sind ja in der Wissenschaft wie im Leben oft die unerwartesten, – wie bald, vielleicht von heute auf morgen, eine Erkenntnis auf entlegenstem Gebiet plötzlich entscheidende, ja lebenswichtige praktische Bedeutung bekommt. Wie der berühmte Physiker L. Boltzmann einmal gesagt hat, ist nichts praktischer als die Theorie. Aber sie muss – von sich aus – auf Vorrat gearbeitet werden, wenn sie der Praxis in jenem Ausmass zur Verfügung stehen soll, wie sie es bisher getan hat. Anders zu verfahren und ihre Zielsetzung auf das augenblicklich praktisch Verwertbare zu beschränken, hiesse die Entwicklung lähmen und wäre eine schwere Schädigung derer, die nach uns wirken sollen, und denen wir nicht nur das erworbene Wissensgut übermitteln wollen, sondern auch die Fähigkeit, es zu mehren.”
>
> Heinrich Tietze

EINLEITUNG

Einer der wohl wortkargsten, mathematischen Vorträge wurde auf der Jahrestagung 1903 der Amerikanischen Mathematikervereinigung von F. Cole gehalten. Nur das Schaben der Keide war zu hören. Er begann fein säuberlich den Wert von 2^{67} auszurechnen, indem er 2*2*2*... berechnete. Hernach zog er Eins vom Resultat ab. An der Tafel stand das Resultat:

2^{67}-1 = 147'573'952'589'676'412'927

Hernach begann er auf der zweiten Tafelhälfte die Multiplikation von

193'707'721 * 761'838'257'287

auszurechnen. Das Resultat schrieb er ebenfalls an die Tafel. Es war wiederum dieselbe 21-stellige Zahl

147'573'952'589'676'412'927

Cole sagte kein Wort, legte die Kreide weg und setzte sich wieder. Das war's. Doch das Publikum brach in stehenden Applaus aus – wahrscheinlich das einzige Mal während der ganzen Tagung! Niemand stellte eine Frage, niemand fügte eine Bemerkung hinzu. Der nächste Redner trat ans Pult.

Was war geschehen? Bereits 1644 vermutete Mersenne, ein Minoritenbruder, dass 2^{67}-1 eine Primzahl sei. Es gelang jedoch während 230 Jahre nicht, die Vermutung zu bestätigen noch sie zu widerlegen. Erst 1876 konnte Lucas, ein französischer Mathematiker, mit einem genialen Kunstgriff nachweisen, dass 2^{67}-1 sicher keine Primzahl sein könne und widerlegte damit Mersenne's Vermutung. Eine Zerlegung konnte er jedoch auch nicht angeben. Cole gab an, dass er die Sonntage dreier Jahre damit verbracht habe, die beiden Primfaktoren zu suchen. Und der Mann, wenn er systematisch durchprobiert hat, muss Glück gehabt haben: Denn es gibt etwa 6'000'000 Primzahlen, die Kandidaten für eine Teilung sind – denn soviele Primzahlen gibt es bis zur kleineren Teilerzahl 193'707'721. Wenn er jedes Wochenende 2'000 Teilungen schaffte (eine pro Minute an einem 16.6 Stundentag), dann hat er im Verlaufe der drei Jahre 312'000 Primzahlen betrachtet. Die Wahrscheinlichkeit, dass er

zufällig auf die richtige Zahl getroffen ist, beträgt also 1/19. Wir wissen natürlich, dass er nicht systematisch probiert hat, sondern eine Methode verwendet hat, welche wir unten noch vorstellen werden.

Als Mathematica, ein komplexes Softwarepaket für die Mathematik, von unserem Institut gekauft wurde, habe ich nach der Installation als erstes folgenden Befehl eingegeben

FactorInteger[2^67-1]

Der Macintosh SE30 (Motorola 68030) brauchte ganze 11 Sekunden, bis er ausgab:

{{193'707'721, 1}, {761'838'257'287, 1}}

Dies ist eine Rechenleistungssteigerung seit 1903 von 1:1'701'818.

Vor drei Jahren machte wiederum eine Sensation die Runde. Es gelang zwei Forschern, A. Lenstra von Bell Communications Research in Morristown und M. Manasse vom Research Center der Digital Equipment Corp. in Palo Alto, den spektakulären Durchbruch, die 155-stellige Monsterzahl $2^{512}+1$ in drei Primzahlen zu zerlegen. Auch diese Zahl hat eine wechselvolle Geschichte. Denn Fermat glaubte 1640, mit $2^{512}+1$ eine Riesenprimzahl gefunden zu haben. Aber bereits 1903 wurde diese Vermutung zunichte gemacht. Da man zeigen konnte, dass sie durch 2'424'833 teilbar ist. Die restliche 148-stellige Zahl, nämlich

55293737465394924514694517099552200615379969757061180616246815528004460637386355995657739308921082102107781683053991969153149444980114382913931182 09

konnte aber bis 1990 nicht zerlegt werden. Nun haben die genannten Forscher die Zahl geknackt. Ihre Faktoren sind:

7416400626275308015247871419019374740599407810975190239058213161444157595047050080928187116939407374556028256478842083373957362004549187833663426 57 *

Es war ein veritabler Kraftakt. Denn die Lösung wurde durch ein parallelisierbares Computerprogramm, welches von Tausenden von rechenwilligen Computern über den ganzen Erdball häppchenweise verteilt parallel ausgeführt wurde. 500 Grosscomputer-**Jahre** Rechenleistung wurden dazu verwendet, bis einer die Lösung ausspuckte. Später wurde nachgewiesen, dass die beiden Faktoren in der Tat Primzahlen sind, sich die beiden Zahlen also nicht mehr weiter faktorisieren lassen (Fricker F. 1990).
Selbstverständlich habe ich es unterlassen – ja nicht einmal im Traume versucht, das Problem auf meinem Macintosh mit Mathematica zu lösen!

Bis vor kurzem wurde diese esoterische Tätigkeit der Zahlenjonglierei als vollkommen nutzlos betrachtet. Seit den 70er Jahren, als die Public-Key Kryptoverfahren entdeckt wurden, sind grosse Zahlen – und insbesondere

Primzahlen – zu einem weitläufigen Forschungsfeld geworden. Die Zahlentheorie ist aus ihrem Schattendasein herausgetreten, und spielerisch anmutende Zahlenakrobatik nimmt heute eine wichtige Rolle in der Kryptographie und der Datensicherung ein.

PRIMZAHLEN

Die Sicherheit moderner Kryptosysteme hängt von zwei Fragen ab:

1. Wie einfach ist es, eine Zahl als Primzahl zu identifizieren?
2. Wie schwierig ist es, eine Zahl zu faktorisieren?

Beide Fragen sind bis heute offen, obwohl es wahrscheinlich ist, dass das Problem der Primzahlenerkennung sehr viel leichter als das Problem der Faktorisierung ist. Auf dieser Annahme beruhen jedenfalls einige, moderne Kryptosysteme.

Primzahlen sind Zahlen, die nur duch sich und eins teilbar sind. Sie spielen eine zentrale Rolle in der Zahlentheorie. In der Tat besteht jede zusammengesetzte Zahl (also eine Nicht-Primzahl) aus einer eindeutigen Menge von Primfaktoren, d.h. sie kann als Multiplikation von einer Menge von Primzahlen ausgedrückt werden (Hauptsatz der Zahlentheorie). Die Anzahl der Primzahlen ist unendlich. Dies lässt sich leicht beweisen und war bereits Euklid bekannt: Angenommen, die Primzahlen seien eine endliche Menge $\{p_1, p_2, \mathrm{K}\, , p_n\}$. Dann ist die Zahl $q = p_1 \cdot p_2 \cdot \mathrm{K} \cdot p_n + 1$ durch keine der n Primzahlen teilbar. Dies steht aber im Widerspruch zur Annahme, dass die Primzahlenmenge endlich ist (Euklid).

(Nach diesem Beweis liefert also die Zahlensequenz $q_i = p_1 p_2 \mathrm{K}\ p_i + 1$ Primzahlen oder Faktoren, die nicht in $\{p_1, p_2, \mathrm{K}\, , p_i\}$ sind. Wenn wir mit i=1 beginnen, so erhalten wir:

q_1 = 2 + 1 = 3
q_2 = 2*3 + 1 = 7
q_3 = 2*3*7 + 1 = 43
q_4 = 2*3*7*43 + 1 = 1807 = 13*139
q_5 = 2*3*7*43*139 + 1 = 251085 = 5*50207
q_6 = 2*3*7*43*139*50207 + 1 = 12603664039 = 23*1607*340999
q_7 = 2*3*7*43*139*50207*340999 + 1 = 429836833293963 = 23*79*2365247734339
q_8 = 2*3*7*43*139*50207*2365247734339 + 1 = 1016587861619057545906876111 9
= 17*127770091783*46802225641471129

Zwei kuriose Zahlen sind

1'111'111'111'111'111'111 und
1000'000'000'000'000'000'000'000'000'000'000

Die erste ist eine Primzahl, während sich die zweite nur in die beiden

Primzahlen 8'589'934'592 und 116'415'321'826'934'814'453'125 zerlegen lässt (Anmerkung: das kann nicht sein!).)

Es ist keine Formel bekannt, welche die *n*-te Primzahl – ausgedrückt als P_n – berechnen könnte, oder angeben könnte, wieviele Primzahlen – bezeichnet mit $\pi(n)$ – es bis zu einer bestimmten Zahl *n* gibt. Hingegen sind sehr profunde Kenntnisse über die Primzahlenverteilung bekannt. So ist die *n*-te Primzahl P_n ungefähr *n* mal der natürliche Logarithmus von *n*, also:

$$P_n \approx n \cdot \ln(n)$$

Ferner ist die Anzahl Primzahlen bis zur Zahl *n* gegeben durch

$$\pi(n) \approx \frac{n}{\ln(n)}$$

Die folgende Tabelle 1 gibt die Näherung für einige Werte wider

n	π(n)	n/ln(n)	π(n)/(n/ln(n))
2	1	2.885	0.347
10	4	4.343	0.921
10^2	25	2.174*10	1.150
10^3	168	1.449*10^2	1.159
10^4	1229	1.086*10^3	1.132
10^5	9592	8.695*10^3	1.103
10^6	78498	7.238*10^4	1.085
10^7	664579	6.204*10^5	1.071
10^8	5761455	5.429*10^6	1.061
10^9	50847534	4.825*10^7	1.054
10^10	455052512	4.343*10^8	1.048
10^11	4118054813		
10^12	37607912018		
10^13	346065536839		
10^14	3204941750802		
10^15	29844570422669		
10^16	279238341033925		

Tabelle 1

Es gilt der Primzahlensatz:

$$\lim_{n->\infty}\left(\frac{\pi(n)}{n/\ln(n)}\right)=1$$

Es gelten sogar die Rosser-Schoenfeld Formeln

$$\ln(n)-\frac{3}{2}<\frac{n}{\pi(n)}<\ln(n)-\frac{1}{2} \quad (\text{für alle } n \geq 67)$$

und

$$n(\ln(n)+\ln(\ln(n))-\frac{3}{2}<P_n<n(\ln(n)+\ln(\ln(n))-\frac{1}{2} \quad (n \geq 20)$$

(aus Graham, Knuth, Patashnik S. 111).

Andererseits kann der Abstand zwischen zwei aufeinanderfolgenden Primzahlen beliebig gross werden, wie man durch die folgende Zahlensequenz, wenn *n* ein ganze Zahl ist und beliebig gross gewählt wird, leicht sieht:

$$n!+2,\ n!+3,\ n!+4,\text{K},\ n!+n.$$

Denn die erste Zahl in der Liste ist durch zwei, die zweite durch drei, die dritte durch vier, usw. und die letzte durch *n* teilbar.
Ferner konnte gezeigt werden, dass die Primzahlenfolge durch kein Polynom ausdrückbar ist, obwohl bereits Euler zwei einfache Polynome gefunden hatte, welche 17 bzw. 41 Primzahlen erzeugen.

$$n^2+n+17 \quad (0 \le \text{n} < 17)$$
$$n^2+n+41 \quad (0 \le \text{n} < 41)$$

Im Zusammenhang mit Primzahlen gibt es viele offene Fragen. So konnte die sogenannte Goldbachsche Vermutung immer noch nicht bewiesen werden. Sie besagt, dass jede gerade Zahl grösser vier sich als Summe zweier ungeraden Primzahlen (das sind alle Primzahlen ausser die Zahl 2) ausdrücken lässt. Eine abgeschwächte Form der Vermutung besagt, dass jede ungerade Zahl grösser sieben sich als Summe dreier ungerader Primzahlen darstellen lässt. Beide Vermutungen sind unbewiesen, wenn auch die russischen Mathematiker Winogradow und Borodzin bewiesen haben, dass die schwache Form der Goldbachschen Vermutung für alle Zahlen grösser als $3^{3^{15}}$ – einer Zahl mit mehr als sechs Millionen Stellen – zutrifft. Diese Vermutung ist also 'fast' richtig. Man braucht sie nur noch für eine *endliche* – wenn auch grosse – Anzahl von Fällen zu zeigen.
Auch die Frage, ob es unendlich viele Primzahlenpaare – Primzahlen, deren Differenz zwei beträgt – gibt, ist offen. Gegenwärtig scheint das Primzahlenpaar $1639494*(2^{4423}-1) +/- 1$ das grösste Primzahlenpaar zu sein.
Ferner ist die Frage offen, welche Zahlen perfekt sind. (Eine Zahl ist perfekt, wenn sie die Summe all ihrer Teiler ist.) Die ersten perfekten Zahlen sind:

$$p_1 = 2(2^2-1) = 1+2+3 = 6$$
$$p_2 = 2^2(2^3-1) = 1+2+4+7+14 = 28$$
$$p_3 = 2^4(2^5-1) = 496$$
$$p_4 = 2^6(2^7-1) = 8128$$
$$p_5 = 2^{12}(2^{13}-1) = 33550336$$

Der Primzahlensatz kann dazu verwendet werden, um die Anzahl der zufällig generierten, *n*-stelligen Zahlen zu finden, die nötig sind, bis eine unter ihnen im Mittel eine Primzahl ist. Diese Zahl ist $1/\ln(n)$, d.h. um eine 100-stellige Primzahl zu generieren, müssen etwa $\ln(10^{100})$ = 230 zufällige Zahlen betrachtet werden.

Die grössten bekannten Primzahlen sind eine Untermenge der Mersennesche Zahlen, welche definert sind als $M_k = 2^k - 1$, wobei *k* eine Primzahl ist. Durch Einsatz elektronischer Rechner weiss man, dass es für Indizes *k*<12000 genau 23 Mersennesche Primzahlen gibt, nämlich für *k*=2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607, 1279, 2203, 2281, 3217, 4253, 4423, 9689, 9941, 11213. Seit 1971 sind weitere 8 hinzugekommen: 19937, 21701, 23209, 44497, 86243, 110503, 132049 und 216091. (Graham, Knuth, Patashnik S. 109, siehe auch Peterson S. 47). Die letzte besitzt 65'050 Ziffern und war 1985 die grösste bekannte Primzahl. Sie wurde mit Hilfe des Supewrcomputers CRAY X-MP/24 zufällig gefunden. Der Computer brauchte für die 1.5 Billionen Rechenoperationen ganze drei Stunden. Würde man sie Ziffer an Ziffer in der Grösse der Bildschirmzeichen aufschreiben, so wäre sie etwa 200 Meter lang! Kürzlich wurde die bisher wohl grösste Primzahl bekannt: $2^{859433} - 1$. Sie hat 258716 Stellen. (NZZ vom 12. Januar 1994).
(Die grösste nicht-Mersennesche bekannte Primzahl ist 391581*2^216193-1).

Die Suche nach Mersenneschen Primzahlen kann als ein Spiegel für die gesteigerte Rechenkapazität der letzten zweihundert Jahren dienen. Die ersten 12 Zahlen waren bereits vor dem Zweiten Weltkrieg bekannt. Alle andern wurden durch Computereinsatz gefunden.

Es ist relativ einfach, Mersennesche Zahlen auf ihre Primzahleneigenschaft zu prüfen. Der heute immer noch gebräuchliche Lucas-Lehmer Test verwendet die folgende Rekurrenzgleichung

$$u_1 = 4$$

$$u_i = (u_{i-1}{}^2 - 2) \bmod M_k \quad (\text{für } 1 > i \geq k-1)$$

Wenn sich dabei herausstellt, dass u_{k-1} gleich Null ist, dann ist M_k eine Primzahl. Damit lässt sich zum Beispiel leicht feststellen, dass $M_5 = 2^5 - 1 = 31$ eine Primzahl ist. Wir brauchen nur die Folge zu berechnen:

$$u_1 = 4$$
$$u_2 = (4^2 - 2) \bmod 31 = 14$$
$$u_3 = (14^2 - 2) \bmod 31 = 8$$
$$u_4 = (8^2 - 2) \bmod 31 = 0$$

Etwas schwieriger ist das Erkennen einer beliebigen Primzahl *n*. Der gegenwärtige Rekord (1992) liegt etwa bei einer Zahl mit 1505 Ziffern (Morain F. zitiert nach Maurer U.). Eine einfache Methode ist natürlich, alle Zahlen bis zur Quardartwurzel von *n* systematisch auf Teilbarkeit zu untersuchen. Eine bessere Methode ist das Sieb des Eratosthenes. Es beschränkt die Divisionen auf die Primzahlen, die kleiner als die Quadratwurzel von *n* sind. Diese Methoden sind allerdings sehr aufwendig und nur für kleine Zahlen anwendbar. Zudem erhalten wir mehr Information als wir verlangten: neben der Information, ob die Zahl eine Primzahl ist oder nicht, bekommen wir nämlich die Primfaktoren der Zahl *n*, falls diese keine Primzahl ist.

Verschiedene, heute gebräuchliche Methoden um festzustellen, ob eine Zahl eine Primzahl ist, beruhen auf dem kleinen Fermatsatz, der folgendermassen lautet:

> Ist *p* eine Primzahl und *b* eine zu *p* teilerfremde natürliche Zahl kleiner *p*, so gilt
>
> $$b^{p-1} \bmod p = 1$$

Beispiel: $2^{7-1} \bmod 7 = 64 \bmod 7 = 1$, denn 7 ist eine Primzahl.

Leider gilt die Umkehrung nicht: Aus dem Satz folgt nicht automatisch die Primzahleigenschaft. So ist zum Beispiel $2^{340} \bmod 341 = 1$. 341 ist jedoch keine Primzahl, denn es gilt: 341 = 11*31. Auch $3^{90} \bmod 91 = 1$. Wiederum aber ist 91 keine Primzahl, denn es gilt: 91 = 7*13.
Zahlen, die den Fermattest bestehen, aber keine Primzahlen sind, heissen Pseudoprimzahlen. Demnach sind 341 und 91 Pseudoprimzahlen. Bei Pseudoprimzahlen gibt man noch die Basis an: so ist 341 eine Pseudoprimzahl zur Basis 2 und 91 eine zur Basis 3.
Allerdings besteht 341 den Fermattest nicht zur Basis 3 und 91 besteht den Test nicht zur Basis 2 ($3^{340} \bmod 341 \neq 1$ und $2^{90} \bmod 91 \neq 1$), sodass beide sofort als zusammengesetzte Zahlen erkannt werden.

Leider gibt es auch Zahlen (die sogenannten Carmichael-Zahlen), welche den

Fermattest für mehrere Basen bestehen und trotzdem keine Primzahlen sind. Die kleinsten Carmichael-Zahlen sind 561, 1105, 1729, 2465, 2821, 6601. Wenig ist bekannt über diese Zahlen, ausser dass sie sehr rar sind (unter 100'000'000 gibt es gerade 255 Carchimael-Zahlen). Allerdings gibt es nur 13 Zahlen, die kleiner als 25'000'000'000 sind, und den Fermattest für die Basen 2, 3 und 5 gleichzeitig bestehen. Diese sind in Tabelle 2 zusammengestellt. Davon bleibt nur die fünfte (3'215'031'751) in Tabelle 2 übrig, wenn zusätzlich die Basis 7 hinzugenommen wird (see Salomaa A., p.142).

Zahl	Faktorization
25326001	= 2251*11251
161304001	= 7333*21997
960946321	= 11717*82013
1157839381	= 24061*48121
3215031751	= 151*751*28351
3697278427	= 30403*121609
5764643587	= 37963*151849
6770862367	= 41143*164569
14386156093	= 397*4357*8317
15579919981	= 88261*176521
18459366157	= 67933*271729
19887974881	= 81421*244261
21276028621	= 103141*206281

Tabelle 2

Obwohl der Fermat-Test kein absoluter Test für die Primzahleneingenschaft einer Zahl ist, so ist er ein sehr zuverlässiger Test, um eine Zahl sofort als zusammengesetzt zu erkennen. Umgekehrt ist die Wahrscheinlichkeit, dass eine Zahl den Fermattest zu mehreren Basen gleichzeitig besteht, sehr klein. Für eine 100-stellige Zahl, welche den Fermat-Test für 50 zufällig ausgewählte Basen besteht und zusammengesetzt ist, ist die Wahrscheinlichkeit viel kleiner als eine fehlerhafte Berechnung durch den Computer auf Grund der kosmischen Strahlung. Für viele praktische Zwecke genügt ein solcher Test. Der Miller-Rabin-Test, welche heute wohl der meist verwendete Primzahlentest ist, basiert auf diesen Überlegungen.

Um nun eine Primzahl zu generieren, werden eine Reihe von Zahlen in einem Intervall zufällig ausgewählt. Dann wird jede Zahl mit Hilfe des Miller-Rabin Tests analysiert. Wird der Test bestanden, so wird die Zahl als Primzahl betrachtet. Es ist wichtig zu sehen, dass ein solcher Test nicht mit absoluter Sicherheit eine Primzahl liefert, die Wahrscheinlichkeit dafür ist jedoch sehr gross.

In Maurer U. [1993] wird ein Algorithmus vorgestellt, der *beweisbare* Primzahlen erzeugt. Der Algorithmus ist erst noch schneller als beim probabilistischen Test.

Pratt bewies 1975, dass die Komplexität der Primzahlenerkennung höchstens nicht-deterministisch polynomial ist. Adleman, Pomerance und Rumely zeigten 1983, dass die Komplexität der Primzahlenerkennung $O(\log(n)^{c\ln(\ln(\ln(n)))})$ ist – für $n = 10^{1000000}$ ist $\ln(\ln(\ln(n)))$ gerade 2.68.

Schwieriger scheint das Problem der Faktorisierung – die Zerlegung einer Zahl in seine Primfaktoren – zu sein. Auf der einen Seite möchten die Kryptographen, dass das Problem schwierig bleibt, möchte dafür aber auch die Gewissheit, andererseits versuchen Mathematiker und Informatiker immer grössere Zahlen zu faktorieren. Die Faktorierung 'schwieriger' Zahlen – nämlich solcher, die keine kleinen Faktoren haben – machte in den 80-iger Jahren eine Sprung von 50-stelligen auf 80-stellige Zahlen. Heute scheint die Faktorisierung von 80-stelligen, 'schwierigen' Zahlen auf Supercomputern etwa einen Tag zu dauern. Das Feststellen, ob eine solche Zahl eine Primzahl ist oder nicht, dauert hingegen nur Bruchteile von Sekunden. Daher konzentriert sich die Forschung darauf abzuschätzen, wie schwierig das Faktorisieren tatsächlich ist. Das halbe Dutzend Faktorisierungsalgorithmen, die heute bekannt sind, haben eine Komplexität in der gleichen Grössenordnung. Diese Feststellung sowie die Tatsache, dass die Fortschritte in der Faktorisierung zwar erheblich aber doch nicht so spektakulär wie bei der Primzahlenerkennung sind, könnten daraufhin deuten, dass die Faktorisierung im Gegensatz zur Primzahlerkennung ein schwieriges Problem ist. Stellt sich das als wahr heraus, so können die Kryptographen aufatmen – nicht zurücklehnen, denn es ist noch nicht eine ausgemachte Sache, dass die Schwierigkeit des Knackens ihrer Public-Key Kryptosysteme von der Schwierigkeit der Faktorisierung abhängt.....

VERSCHLÜSSELUNGSTECHNIK (KRYPTOGRAPHIE)

> Menschliche Erfindungskraft kann kein Verschlüsselungssystem schaffen, das durch menschliche Erfindungskraft nicht gebrochen werden könnte.
> E.A. Poe

Verschlüsselungstechniken sind seit dem Altertum bekannt. Sie wurden vor allem für die militärische Geheimhaltung eingesetzt. Heute in einer hochtechnisierten Kommunikationsgesellschaft, finden diese Techniken ein weitaus grösseres Anwendungdfeld. Man denke nur an die elektronische Übermittlung von Buchgeld oder Verträge. Hier sind sichere Verschlüsselungstechniken unerlässlich.

Gebräuchlich waren früher die einfachen Substitution (Ersetzung von Zeichen durch andere), die einfachen Transformation (Verwürfelung) und die polyalphabetische Substitution. Diese können mit Maschinen sehr leicht geknackt werden. Jüngere Verfahren sind die Wegwerf-Schlüssel Technik und die Ein-Weg-Schüssel Technik. Die Nachteile all dieser Techniken liegen auf der Hand: der Schlüssel muss auf einem sicheren Weg dem Empfänger der geheimen Nachricht übermittelt werden.

Alle genannten Verfahren beruhen also darauf, dass sowohl der Sender (derjenige, der die Nachricht verschlüsselt) als auch der Empfänger (derjenige, der die Nachricht wieder entschlüsselt) einen Schüssel gemeinsam haben, und dass dieser Schüssel irgendwie sicher zwischen Nachrichensender und -empfänger transportiert werden muss. Dadurch verlagert sich das Problem des Austausches von geheimen Nachrichten auf den Austausch des Schlüssels. Insbesondere bei elektronischen Kommunikationssystemen gibt es aber eine grosse Anzahl von Teilnehmern mit ständig wechselnden Verbindungen, die zuverlässig geschützt werden wollen. Bei n Teilnehmern gibt es $O(n^2)$ Verbindungen und entsprechend viele Schlüssel, die sicher übertragen werden müssten. Solche Systeme, welche für das Verschlüsseln und das Entschlüsseln der Nachricht denselben Schlüssel verwenden, sind daher im Bereiche der elektronischen Kommunikation völlig unannehmbar. Gesucht wäre also ein kryptographisches Verfahren, bei welchem der Schlüsseltausch entfällt. Wie ist dies möglich? Dies könnte etwa so funktionieren: Der Empfänger gibt öffentlich einen Schlüssel bekannt, mit dem eine Nachricht, die an ihn geschickt wird, verschüsselt werden kann. Er behält einen zweiten Schlüssel, mit dem einzig die Nachricht wieder entschlüsselt werden kann, geheim bei sich. Aber ist ein Verfahren denkbar, welches für die Verschlüsselung einer Nachricht einen anderen Schüssel verwendet wie für deren Entschlüsselung? Das heisst, wäre es möglich, dass derselbe Schlüssel, welcher die Nachricht verschlüsselt, diese nicht mehr entschlüsseln könnte, und es dazu einen zweiten Schlüssel, welcher dann vom Nachrichtenempfänger geheim gehalten werden könnte, bräuchte?
Dies scheint auf den ersten Blick ein Ding der Unmöglichkeit zu sein. Denn für jede Transformationsfunktion (Verschlüsselung) muss es eine inverse Transformationsfunktion (Entschlüsselung) geben, die eindeutig aus der ersten ableitbar ist. Welches Privileg sollte der Empfänger gegenüber alle andern Teilnehmer haben, um die Inverse zu berechnen? Doch solche Verfahren gibt es in der Tat, und sie heissen Public-Key-Systeme. 1975 ist hier ein

sensationeller Durchbruch erzielt worden. Am MIT und an der Stanford University wurden Verschlüsselungsverfahren mit öffentlicher Schüssel-bibliothek entwickelt. Eines davon arbeitet mit grossen Primzahlen. (Ein zweites bekanntes Verfahren beruht auf dem Knapsack-Problem – dies soll aber hier nicht weiter verfolgt werden, da verschiedene solcher Verfahren bereits geknackt worden sind).

Die Technik funktioniert folgendermassen: Jeder Empfänger von verschlüsselten Nachrichten besitzt ein elektronisches Gerät (die RSA Box), welches zwei grosse – sagen wir 100-stellige – Primzahlen p und q gespeichert hat. Das Gerät ist so gebaut, dass niemand – nicht einmal der Besitzer der Box – die beiden Primzahlen erfahren kann. Die Primzahlen werden in der Box durch einen Zufallsgenerator erzeugt. Nach den obigen Ausführungen wissen wir, dass es für die Box einfach ist, zwei solche Primzahlen zu *generieren*. Die Box teilt gegen aussen jedoch das Produkt der beiden Primzahlen, die Zahl $m=pq$ mit. Zudem berechnet die Box eine Zahl d, welche grösser als p und q jedoch kleiner als $(p\text{-}1)(q\text{-}1)$ und dazu teilerfremd ist. d ist der Dechiffrierschüssel des Empfängers und von diesem geheim zu halten. Das Tripel (p,q,d) bezeichnet man den privaten Schüssel des Empfängers (wobei dem Empfänger selbst nur d bekannt sein muss). In die öffentliche Schüsselbibliothek lässt er die Zahl m und eine Zahl e eintragen, die durch die Gleichung $(de) \bmod (p-1)(q-1) = 1$ definiert ist, für die also $de = 1 + k(p-1)(q-1)$ mit einem ganzzahligen k gilt. Das Zahlenpaar (e,m) ist der öffentliche Schüssel. Wir nehmen natürlich an, dass die RSA Box seinem Besitzer neben der Zahl d, die er selbstverständlich für sich behält, auch die öffentliche Zahl e ausgibt. Die Box kann die Zahlen e und d aus den Zahlen p und q leicht berechnen, indem sie zunächst eine Zahl d grösser p und q wiederholt generiert, bis diese teilerfremd zu $(p\text{-}1)(q\text{-}1)$ ist, d.h. bis der grösste gemeinsame Teiler Eins ist. Der grösste gemeinsame Teiler zweier Zahlen kann mit Hilfe der Euklidschen Algorithmus auch für grosse Zahlen schnell festgestellt werden. Sodann muss die Box die Modulo-Gleichung $1 = ed + k(p-1)(q-1)$ lösen, um die Zahl e zu finden. Dies kann mit Hilfe des erweiterten Euklidschen Algorithmus effizient gelöst werden (siehe Knuth S. 301). Der Besitzer der Box erhält also von der RSA-Box drei Zahlen: die Zahlen m und e, welche er in der öffentlichen Schüsselbibliothek eintragen lässt, und welche von irgendeinem Absender dazu verwendet werden eine Nachricht an ihn zu verschlüsseln, sowie die Zahl d, die er für sich behält, und die dazu dient, die Nachricht an ihn zu entschlüsseln.

Mit Hilfe des öffentlichen Schüssels kann nun jeder eine Nachricht für den Empfänger verschlüsseln, die nur mit Hilfe des privaten Schüssels entschlüsselt werden kann. Dieser ist ausschliesslich im Besitz des Empfängers. Die Verschüsselungsoperation kann durch folgende Funktion *Encrypt*() beschrieben werden:

encryptedMessage = Encrypt(m,e,Message)

Der Empfänger wendet die Funktion *Decrypt*() an, um die Nachricht wieder zu entschlüsseln:

Message = Decrypt(d,m,encryptedMessage)

Wie sehen nun diese beiden Funktionen aus? Zunächst muss die Nachricht (Message) in geeignete Blöcke zerlegt werden – sagen wir in Blöcke von 4 Bytes. Diese Blöcke von vier Bytes werden nun als Zahlen zwischen 0 und $2^{32}-1$ interpretiert, nennen wir sie die Zahl *Z*. Dann kann die *Encrypt* Funktion folgendermassen definiert werden:

$$X = Encrypt(Z, e, m) = Z^e \bmod m$$

Die *Encrypt* Funktion produziert aus der Zahl *Z* eine Zahl *X*, welche den Geheimcode – also die verschlüsselte Nachricht – repräsentiert. Diese Zahl *X* wird nun dem Empfänger über das öffentliche Kommunikationsnetz übermittelt. Der Empfänger entschlüsselt den Code *X*, indem er die folgende *Decrypt* Funktion anwendet

$$Z = Decrypt(X, d, m) = X^d \bmod m$$

Denn es gilt die mathematische Identität

$$Z \equiv Decrypt(Encrypt(Z, e, m), d, m)$$

Um die Zahl *d* (der Geheimcode des Empfängers) aus dem öffentlichen Schlüsselpaar (*e*,*m*) zu knacken, müsste man die Zahl *m*, die ja öffentlich bekannt ist, in seine Primzahlen *p* und *q* zerlegen. Aber genau diese Aufgabe ist für grosse – sagen wir 100-stellige – 'schwierige' Zahlen auch für Supercomputer ein enormer Rechenaufwand, der in nützlicher Zeit nicht zu bewältigen ist. Da die Zahl *m* aus zwei etwa gleichgrossen Primzahlen besteht, ist sie enorm schwierig zu knacken. Hingegen ist es relativ einfach für die RSA-Box zwei grosse Primzahlen zu generieren und daraus die geheime Zahl *d* zu gewinnen. Das Public-Key-Verfahren beruht gerade auf diesem Unterschied: Es ist einfach mit einem probabilistischen (oder deterministischen) Verfahren, eine Zahl als Primzahl zu identifizieren, bzw. eine solche zu generieren, jedoch ist es wesentlich schwieriger eine zusammengesetzte Zahl in seine Primzahlen zu zerlegen, vor allem dann, wenn die zusammengesetzte Zahl keine kleinen Primfaktoren enthält.

Bei der geheimen Übermittlung von Nachrichten – man denke nur an Geldtransferaufträge – ist die Identifikation des Absenders ebenfalls von entscheidender Bedeutung. Auf den ersten Blick scheint es, dass Public-Key-Verfahren dieses Problem nicht lösen könnten, denn irgendein Absender Alice kann mit Hilfe des öffentlichen Schlüssels eine geheime Nachricht an Bob senden und vorgeben diese Nachricht hätte Clara geschickt. Das heisst, Bob hat keine Möglichkeit nachzuweisen, von welchem Absender die Nachricht kommt. Obwohl die Nachricht geheim ist und nur von Bob entschlüsselt werden kann, hätte sie von irgendeinem Absender geschickt werden können, da ja gerade der Verschlüsselungsschlüssel öffentlich bekannt ist.
Das Problem kann gelöst werden, wenn man feststellt, dass ebenfalls folgende Identität gilt:

$$Z \equiv Decrypt(Encrypt(Z,e,m),d,m) \equiv Encrypt(Decrypt(Z,d,m),e,m)$$

Das heisst, wenn die Nachricht Z zuerst verschlüsselt und anschliessend entschlüsselt wird, so erhält man wiederum die Nachricht Z. Aber auch das Umgekehrte gilt, und das ist der entscheidende Punkt: Wenn man die Nachricht Z zuerst entschlüsselt – das heisst die Entschlüsselungsfunktion aud die (noch gar nicht verschlüsselte) Nachricht anwendet – und anschliessend verschlüsselt, so erhält man ebenfalls wiederum die Nachricht Z. Wenn nun Alice an Bob eine geheime Nachricht übersenden will, so entschlüsselt er zuerst seine Unterschrift mit seinem eigenen geheimen Schlüssel. Dann verschlüsselt er die Nachricht zusammen mit seiner entschlüsselten Unterschrift mit dem öffentlichen Schlüssel von Bob. Der Empfänger Bob entschlüsselt die Nachricht zusammen mit der Unterschrift mit seinem geheimen Schlüssel und verschlüsselt anschliessend die Unterschrift von Alice mit dem öffentlichen Schlüssel von Alice. Die Nachricht und die Unterschrift liegen jetzt beide im Klartext vor. Allerdings darf die entschlüsselte Unterschrift von Alice nicht einfach eine Blanko-Unterschrift sein, sonst könnte Bob an Clara eine Nachricht zusammen mit der entschlüsselten Unterschrift von Alice schicken und so vorgeben, die Nachricht sei von Alice. Die Unterschrift muss mit dem Inhalt der Nachricht selbst verknüpft sein.

OPERATIONEN AUF GROSSEN ZAHLEN

Damit es möglich ist, mit grossen Zahlen zu arbeiten, müssen die Grundoperationen für diese Zahlen implementiert sein. In der Sprache LISP oder Mathematica sind alle Operationen selbstverständlich für grosse Zahlen verfügbar. Mathematica ist hier sogar besonders effizient.

Die Operationen können allerdings auch in jeder andern Sprache implementiert werden. Für unsere Tests wurden die Operationen in Pascal implementiert. Effiziente Algorithmen der vier Grundoperationen (Addition, Subtraktion, Multiplikation und Division) sind in Knuth Kapitel 4.3.1 ausführlich beschrieben. Die Modulo Operation ist ein Nebenprodukt der Division. Ziel der Implementation war es auch herauszufinden, wie schwierig dies zu bewerkstelligen ist. Die Addition und Subtraktion waren schnell implementiert. Die Multiplikation bot keine grösseren Schwierigkeiten, besitzt aber einige subtile Fälle. Die Division allerdings war nicht ganz so einfach. Es galt verschiedene Klippen zu umschiffen, bis die Prozedur einwandfrei funktionierte. Die Modulo-Operation musste vervollständigt werden.
Die Operationen wurden so implementiert, dass es einfach ist von der Basis 10 zu irgendeiner andern Basis zu wechseln oder beliebig grosse Zahlen zu definieren. Die konkrete Implementation beschränkt sich allerdings auf 255 Ziffern, die als ARRAY OF CHAR (*string*) definiert sind. Die nullte Position der Zeichenkette gibt deren Länge an. Ab der ersten Position beginnen die Ziffern in umgekehrter Reihenfolge. Jede Ziffer ist als ein Byte mit ASCII Character zwischen 0 und 9 abgelegt (da die Basis 10 verwendet wurde). Der Typ wurde als *BigInteger* bezeichnet.

```
type
  BigInteger = string;   { stored from right to left }
```

Zuerst sind vier Umformungsoperationen definiert, die es erlauben, BigInteger Zahlen in lesbare Ziffernsequenzen (oder *longint*-Zahlen, falls die Zahl nicht zu gross ist) umzuwandeln und umgekehrt.

```
function Str2Big (a: string): BigInteger;
function Big2Str (a: BigInteger): string;
function Long2Big (a: longint): BigInteger;
function Big2Long (a: BigInteger): longint;
```

Anwendung:

```
var a: BigInteger; b:string; c:longint;

a:=Str2Big('1489346785');
b:=Big2Str(a);
a:=Long2Big(342865);
c:=Big2Long(a);
```

Str2Big() wandelt eine Ziffernsequenz in das interne Format von BigInteger um. *Big2Str*() ist die inverse Operation dazu. *Long2Big*() wandelt eine Zahl im longint-Format in das interne Format von BigInteger um, *Big2Long*() ist wiederum die inverse Operation dazu. Bei einer Basisänderung müssen alle

vier Operationen natürlich neu geschrieben werden.

Die nächste sechs Operationen sind der Kern der Implementation. Es sind dies die Addition *BigAdd*(), Subtraktion *BigSub*(), Subtraktion mit negativen Zahlen *BigSubNeg*(), Multiplikation *BigMul*(), Division *BigDiv*() und Modulo-Operation *BigMod*().

```
function BigAdd (a, b: BigInteger): BigInteger;
function BigSub (a, b: BigInteger): BigInteger;
function BigSubNeg (a, b: BigInteger; na, nb: boolean; var nc: boolean):
BigInteger;
function BigMul (a, b: BigInteger): BigInteger;
function BigDiv (a, b: BigInteger): BigInteger;
function BigMod (a, b: BigInteger): BigInteger;
```

Die Operationen wurden als Funktionen implementiert, um die darauf aufbauenden Operationen möglichst transparent gestalten zu können. Hätte die Effizienz im Vordergrund gestanden, so wären diese Operationen als Prozeduren mit Referenzparameterrückgabe implementiert worden. Nur positive Zahlen sind erlaubt, sodass die Subtraktion einen Fehler liefert, wenn eine grössere von einer kleineren Zahl abgezogen wird. Da der erweiterte Algorithmus von Euklid auch negative Zahlen liefern kann, wurde die Subtraktion für negative Zahlen *BigSubNeg()* separat implementiert: Die Funktion übernimmt zwei positive *BigInteger*-Zahlen *a* und *b*. Die boolschen Parameter *na* und *nb* geben an, ob diese beiden Zahlen als negative oder positive Zahlen zu betrachten sind: Ihre Werte sind TRUE für negative Zahlen. Die Funktion gibt eine *BigInteger*-Zahl zurück, sowie eine boolsche Variable *nc*, welche angibt, ob das Resultat positiv oder negativ ist.
Die Verwendung der Operationen ist einfach. So kann zum Beispiel die arithmetische Operation $e \leftarrow (a * b - c) \bmod d$ folgendermassen geschrieben werden:

```
e := BigMod(BigSub(BigMul(a,b),c),d);
```

Drei weitere Operation sind die Potenz von ganzen Zahlen, die Modulopotenz und die ganzzahlige Wurzel.

```
function BigPower (base: BigInteger; expo: integer): BigInteger;
function BigPowerMod (base, expo, modu: BigInteger): BigInteger;
function BigSqrt (a: BigInteger): BigInteger;
```

Die erste Operation *BigPower*() liefert das Resultat zu $base^{expo}$, die zweite Operation *BigPowerMod*() ist die zentrale Operation zur Implementation des Fermat-Tests sowie der *Encrypt*() und *Decrypt*() Funktionen im Public-Key-Kryptosystem. Sie liefert das Resultat von $base^{expo} \bmod modu$. Die dritte

Funktion *BigSqrt*() liefert die ganzzahlige Wurzel. Dezimalstellen werden abgeschnitten).
Alle drei Operationen beruhen auf den obigen sechs Kernoperationen.

BigPower() ist rekursiv implementiert wie folgt:

```
function BigPower (base: BigInteger; expo: integer): BigInteger;
  var   i: integer;   result: BigInteger;
begin
  if expo = 0 then  BigPower := Str2Big('1')
  else if expo mod 2 = 0 then begin
    result := BigPower(base, expo div 2);
    BigPower := BigMul(result, result);
   end
  else  BigPower := BigMul(base, BigPower(base, expo - 1));
end;
```

Die Funktion *BigPowerMod*() is folgendermassen implementiert:

```
function BigPowerMod (base, expo, modu: BigInteger): BigInteger;
  var  result: BigInteger;
 begin
  result := Str2Big('1');
  while expo <> Str2Big('0') do begin
    if BigIsEven(expo) then begin
      base := BigMul(base, base);
      base := BigSub(base, BigMul(modu, BigDiv(base, modu)));
      expo := BigDiv(expo, Str2Big('2'))
    end
    else begin
      result := BigMul(result, base);
      result := BigSub(result, BigMul(modu, BigDiv(result, modu)));
      expo := BigSub(expo, Str2Big('1'));
     end;
  end;
  BigPowerMod := result;
end;
```

BigIsEven(a) ist eine lokale Funktion, welche testet, ob die Zahl a gerade ist oder nicht. Ist a gerade, so liefert diese Funktion TRUE und sonst FALSE zurück.
Die Wurzeloperation *BigSqrt*() wurde durch die Rekurrenzformel $x_{n+1} = (x_n + a / x_n) / 2$ implementiert:

```
function BigSqrt (a: BigInteger): BigInteger;
  var x: BigInteger; {approximation}  i: integer;
begin
  x := a;  x[0] := chr(ord(x[0]) div 2 + 1);
  for i := 1 to 10 do  x := BigDiv(BigAdd(x, BigDiv(a, x)), Str2Big('2'));
  BigSqrt := x;
end;
```

Drei weitere Funktionen implementieren den Euklidschen Algorithmus zum Auffinden des grössten gemeinsamen Teilers $\gcd(a,b)$ zweier Zahlen a und b, den erweiterten Euklidschen Algorithmus zum Lösen der Modulo-Gleichung $ax + by = \gcd(a,b)$ und die Generierung einer Zufallszahl mit *digits* Anzahl

Ziffern. Die *BigGcd*() ist der Algorithmus von Euklid zum Auffinden des grössten gemeinsamen Teilers. Die Function *BigGcd_Ext*() ist der erweiterte Euklidsche Algorithmus. Die Eingabeparameter sind zwei positive Zahlen *a* und *b*. Das Resultat ist der grösste gemeinsame Teiler $\gcd(a,b)$. Ausserdem werden zwei Zahlen *x* und *y* zurückgegeben, welche der Gleichung $ax+by=\gcd(a,b)$ genügen. Da diese Zahlen negativ sein können, wird dies in den beiden Parametern *nx* und *ny* angezeigt. Die Funktion ist so geschrieben, dass jede Zahl zusammen mit einer boolschen Variable vorkommt, welche angibt, ob die entsprechende Zahl positiv oder negativ ist. Die Funktion *BigRand*() produziert eine Zufallszahl der Länge *digits*.

```
function BigGcd (a, b: BigInteger): BigInteger;
  var  x: BigInteger;
begin
  repeat
   x := a;  a := b;  b := BigMod(x, b);
  until b = Str2Big('0');
  BigGcd := a;
end;



function BigGcd_Ext(u,v:BigInteger; var x,y:BigInteger; var
nx,ny:boolean):BigInteger;
  var  v1,  v2,  v3,  t1,  t2,  t3,  d,  q,  x1,  y1: BigInteger;
       nv1, nv2, nv3, nt1, nt2, nt3, nd, nq, nx1, ny1: boolean;
 begin
  nv1 := false;  nv2 := false;  nv3 := false;
  nt1 := false;  nt2 := false;  nt3 := false;
  nd := false;  nq := false;  nx1 := false;  ny1 := false;

  x1 := Str2Big('1');  y1 := Str2Big('0');  d := u;
  v1 := Str2Big('0');  v2 := Str2Big('1');  v3 := v;
  while v3 <> Str2Big('0') do begin
    q := BigDiv(d, v3);  nq := (nd = nv3);
    t1 := BigSubNeg(x1, BigMul(v1, q), nx1, (nv1 = nq), nt1);
    t2 := BigSubNeg(y1, BigMul(v2, q), ny1, (nv2 = nq), nt2);
    t3 := BigSubNeg(d, BigMul(v3, q), nd, (nv3 = nq), nt3);

    x1 := v1;  nx1 := nv1;
    y1 := v2;  ny1 := nv2;
    d := v3;  nd := nv3;
    v1 := t1;  nv1 := nt1;
    v2 := t2;  nv2 := nt2;
    v3 := t3;  nv3 := nt3;
   end;
  x := x1;  nx := nx1;
  y := y1;  ny := ny1;
  BigGcd_Ext := d;
 end;



function BigRand (digits: integer): BigInteger;
  var  a: BigInteger;  i: integer;
begin
  a[0] := chr(digits);
  for i := 1 to digits do  a[i] := chr(abs(Random) mod BASE);
  BigRand := a;
end;
```

Die letzten zwei Funktionen betreffen Primzahlen:

```
function BigIsPrime (a: BigInteger): boolean;
function BigPrime (digits: integer): BigInteger;
```

Die erste Funktion *BigIsPrime*(*a*) testet mit Hilfe des probabilistischen Rabin-Miller-Tests, ob die Zahl *a* eine Primzahl ist oder nicht. Besteht die Zahl *a* den Test, so liefert die Funktion TRUE und sonst FALSE zurück. Die Funktion *BigIsPrime*(*a*) generiert als erstes alle Primzahlen unter 1000 und teilt die Zahl *a* versuchsweise durch alle generierten Primzahlen. Wenn sie durch keine dieser Primzahlen teilbar ist, wird der Rabin-Miller-Test angewendet. Die letzte Funktion *BigPrime*(*digits*) generiert eine Primzahl bestehend aus *digits* Ziffern. Diese Funktion generiert zuerst eine ungerade Zufallszahl. Dieser wird solange zwei oder vier hinzugefügt, bis sie den Primzahltest besteht.

IMPEMENTATION EINES PUBIC-KEY-KRYPTOSYSTEMS

Auf der Basis der obigen Operationen auf grossen Zahlen ist es einfach, ein Verschlüsselungssystem zu bauen. Allerdings wurde nicht in erster Linie auf Effizienz sondern auf Transparenz Rücksicht genommen. Ein Kryptosystem enthält einerseits eine Verschlüsselungs- und eine Entschlüsselungsfunktion und andererseits ein Schlüsselgenerierungssystem. Auf der Basis der obigen Operationen sind die Verschüsselungs- und Entschlüsselungsoperationen fast trivial:

```
function EnCode (w: Tword; PublicKey1, PublicKey2: BigInteger): BigInteger;
function DeCode (base, PrivateKey, PublicKey2: BigInteger): Tword;
```

Die *EnCode*() Funktion verwendet das öffentlich zugängliche Schlüsselpaar (*PublicKey1*,*PublicKey2*) – oben mit (*e*,*m*) bezeichnet – sowie das zu verschlüsselnde Wort *w*, wobei *Tword* definiert ist als

```
type  Tword = string[2];
```

Der zu verschlüsselnde Text wird also zunächst in Blöcke von 2 Bytes zerlegt. Diese werden in den *longint*-Typ – bzw. in den *BigInteger*-Typ – umgewandelt und anschliessend wird die Modulopotenz Operation angewendet, welche als Resultat eine *BigInteger*-Zahl liefert. Diese letztere Zahl ist der kodierte Text der 2 Bytes. (Besser als zwei Bytes sind natürlich grössere Blöcke. Allerdings muss die dazu entsprechende *BigInteger* Zahl kleiner als *m* sein!)

```
function EnCode (w: Tword; PublicKey1, PublicKey2: BigInteger): BigInteger;
  var
   base: BigInteger;
```

```
  x: longint;
begin
 x := 100 * ord(w[1]) + ord(w[2]);
 base := Long2Big(x);
 EnCode := BigPowerMod(base, PublicKey1, PublicKey2);
end;
```

Die Rückumwandlung geschieht analog, wobei jetzt der private Schüssel verwendet werden muss. *base* bezeichnet das Kodewort und die Funktion *DeCode*() liefert eine Zeichenkette zurück.

```
function DeCode (base, PrivateKey, PublicKey2: BigInteger): Tword;
 var
  result: longint;
  w: Tword;
begin
 result := Big2Long(BigPowerMod(base, PrivateKey, PublicKey2));

 w[1] := chr(ord(result div 100));
 w[2] := chr(ord(result mod 100));
 w[0] := chr(2);
 DeCode := w;
end;
```

Mit Hilfe dieser Funktionen konnte der kodierte Text

```
83129 2476499 2103986 399934 1282001 2491450 897949 759449 2800024
554169 2657364 2994888 1716407 2476499 3037504 33504 1865337 2411177
909733 745719 2103986 101974
```

in Baumann S. 81 geknackt werden. (Der Autor versprach DM 100.-- für den ersten, welcher den Text knackt. Leider wird daraus nichts, denn der Kode im Klartext lautet: «WIDERRUF: SIE BEKOMMEN LEIDER NICHTS, SORRY.»)
Sein öffentlicher Schlüssel ist (2490193,3261347). Da die Zahlen klein sind, war es einfach m=3261347 in seine Primzahlen zu zerlegen. Diese lauten p=1789 und q=1823. Genau diese Aufgabe ist aber für grosse Zahlen ein schwieriges Unterfangen, wie wir bereits wissen. Dann muss man die Gleichung $e \cdot x + (p-1)(q-1) \cdot y = 1$ lösen, was mit Hilfe des erweiterten Euklidschen Algorithmus gelingt (denn e und (p-1)(q-1) müssen teilerfremd sein). Diese liefert x=3769 und y=-2881. Also ist x(=d) der private Schlüssel.

Etwas aufwendiger ist das Generieren eines Schüsselpaars durch die Prozedur *GenerateAKey*(). Diese Funktion liefert einen öffentlichen Schlüssel (e,m) sowie den entsprechenden privaten Schüssel d. Zunächst werden zwei Primzahlen (p,q) der Länge L und L-2 produziert. Daraus wird $m = pq$ sowie (p-1)(q-1) berechnet. Dann wird eine Zufallszahl der Länge L-1 gebildet, die von (p-1)(q-1) abgezogen wird. Von der erhaltenen Zahl wird solange eins

abgezogen, bis diese teilerfremd zu (p-1)(q-1) ist. Die erhaltene Zahl ist unser private Schlüssel d. Um jetzt e zu finden, muss die Modulogleichung $(e \cdot d) \bmod (p-1)(q-1) = 1$ gelöst werden. Dies geschieht mit Hilfe des erweiterten Euklidschen Algorithmus.

```
procedure GenerateAKey(var
PrivateKey,PublicKey1,PublicKey2:BigInteger;L:integer);
  var
   PrivateKey2, PrivateKey3, e, m, d, gcd, y, pq_1: BigInteger;
   ne, ny: boolean;
 begin
  PrivateKey2 := BigPrime(L);                          {p}
  PrivateKey3 := BigPrime(L - 2);                      {q}
  m := BigMul(PrivateKey2, PrivateKey3);               {m}

pq_1:=BigMul(BigSub(PrivateKey2,Str2Big('1')),BigSub(PrivateKey3,Str2Big('1')));

  d := BigAdd(PrivateKey2, BigRand(L - 1));    {BigSub(pq_1, BigRand(L + 2));}
  repeat                                       {find d such that gcd(pq_1,d) =1}
   d := BigSub(d, Str2Big('1'));
  until BigGcd(d, pq_1) = Str2Big('1');

  gcd := BigGcd_Ext(d, pq_1, e, y, ne, ny);    {find e such that d*e +pq_1*y = 1}
  if ne then  writeln('error in Key');

  PrivateKey := d;
  PublicKey1 := e;
  PublicKey2 := m;
 end;
```

Die Prozedur *GenerateAKey*() kann natürlich unendlich variert werden. Es wurde hier ein relativ starres Schema implementiert. Aber dies sollte nach den Ausführungen leicht sein.

REFERENCES

BAUMANN R., [1983], Spiel, Idee und Strategie programmiert in Pascal, Vogel Buchverlag, Würzburg.

FRICKER FRANÇOIS, [1990], Mathematischer Rekord macht Geheimdienst Sorgen, Tages Anzeiger vom 23. Oktober 1990.

GRAHAM R.L., KNUTH D.E., PATASHNIK O., [1989], Concrete Mathematics, a Foundation for Computer Science, Addison-Wesley Publ. Comp., Mass.

KNUTH D.E. [1969], The Art of Computer Programming, Vol 2, Seminumerical Algorithms, Addison-Wesley Publ., Massachusetts.

MAURER U., [1993], Fast Generation of Prime Numbers and Secure Public-Key Cryptographic Parameters, November 1993, Working Paper 201, Eidgenössische Technische Hochschule, Zürich, Departement Informatik.

PETERSON I., [1992], Mathematische Expeditionen, Ein Streifzug durch die moderne Mathematik, Spektrum, Akademischer Verlag, Heidelberg.

REINHARDT F., SOEDER H., [1976], dtv Atlas zur Mathematik, Band 1, dtv Verlag, S.127.

RIESEL H., [1985], Prime Numbers and Computer Methods for Fatorization, Birkhäuser, Boston.

SALOMAA A., [1990], Public-Key-Cryptography, EATCS Monographs on Theoretical Computer Science, Springer-Verlag, Berlin.

TIETZE H., [1982], Gelöste und ungelöste Probleme aus alter und neuer Zeit, Bd 1+2, dtv wissenschaft. (erste Ausgabe 1959).

ZAGIER D. [1977], Die ersten 50 Millionen Primzahlen, Beihefte zur Zeitschrift “Elemente der Mathematik”, Nr. 15, Birkhäuser Verlag, Basel.

Nachtrag 5/96: the definite guide to cryptography is:

SCHNEIDER B., [1996], Applied Cryptography, (2nd ed), Wiley.