RSA & Co. in der Schule

Moderne Kryptologie, alte Mathematik, raffinierte Protokolle

Neue Folge – Teil 2: RSA für große Zahlen

von Helmut Witten und Ralph-Hardo Schulz

Das RSA-Verfahren wurde 1977 von Ronald L. Rivest, Adi Shamir und Leonard Adleman entwickelt (siehe Bild 1). Für diese Erfindung wurden sie im Jahr 2002 mit dem Turing-Preis der ACM geehrt (vgl. Schmidt, 2003). Diese Auszeichnung ist die angesehenste im Fach Informatik, quasi der Nobelpreis (den es für Informatik leider nicht gibt).

Obwohl das Verfahren Bestandteil vieler offizieller Standards wie PEM (***P**rivacy **E**nhanced **M**ail*) oder SWIFT (***S**ociety for **W**orldwide **I**nterbank **F**inancial **T**elecommunications*) ist, wurde es selbst nicht als Standard normiert. Wegen seiner großen Verbreitung und Akzeptanz, (z. B. in SSL-fähigen WWW-Browsern) wird RSA aber als De-facto-Standard für asymmetrische Verschlüsselung angesehen (vgl. Eckert, 2005, S. 161).

In den hier vorgelegten Beiträgen der neuen Folge von „RSA & Co." sollen die mathematischen Grundlagen erarbeitet werden, die zu einem tieferen Verständnis von RSA benötigt werden. Dabei soll es nicht (wie z. B. bei Bartholomé u. a., ^{2}1996) um einen Kurs zur elementaren Zahlentheorie mit RSA als krönendem Abschluss gehen. Vielmehr wird hier jeder Schritt mit einer kryptologischen Fragestellung verknüpft, die das Thema im Unterricht motivieren kann.

Es handelt sich also bei dieser Artikelfolge nicht um fertige Unterrichtseinheiten, obwohl Fragen der Praxis und der Methodik immer bedacht werden. Es bleibt aber die Aufgabe der Unterrichtenden, mit den hier dargestellten Elementen eine auf die jeweilige Lerngruppe zugeschnittene Unterrichtssequenz zu entwickeln.

Im ersten Teil dieser neuen Folge haben wir einen „experimentellen" Zugang zu RSA beschrieben, der die Schülerinnen und Schüler die Probleme beim Einsatz von RSA „am eigenen Leibe" spüren lässt. Bei diesem Zugang mussten wir den RSA-Algorithmus als gegeben voraussetzen (Witten/Schulz, 2006).

In dieser Folge wollen wir einen genetischen Weg zu RSA beschreiben. Dabei werden zunächst der erweiterte euklidische Algorithmus und das Rechnen mit sehr großen Zahlen („Monsterzahlen") im Zentrum stehen.

Andere offene Probleme wie die Korrektheit von RSA, schnelle Primzahltests und die Schwierigkeiten bei der Faktorisierung, die für das Verständnis und die Sicherheit von RSA fundamental sind, werden in weiteren Folgen behandelt.

http://www.usc.edu/dept/molecular-science/pictures/Len-Adi-Ron.jpg

Bild 1:
Rivest, Shamir und Adleman (von rechts nach links).

Von der additiven zur multiplikativen Verschlüsselung

Die Caesar-Verschlüsselung ist nach wie vor das bekannteste (und unsicherste) kryptologische Verfahren. Es basiert auf der Addition modulo *26* (oder allgemeiner: modulo der Anzahl der Buchstaben des verwendeten Alphabets, vgl. z. B. Witten/Letzner/Schulz, 1998). Die Entschlüsselung funktioniert dabei genauso wie die Verschlüsselung. Wenn man die Buchstaben von A bis Z – beginnend bei Null – durchzählt, muss die Summe der Nummern der bei Ver- und Entschlüsselung zu ad-

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
B	2	4	6	8	10	12	14	16	18	20	22	24	26	1	3	5	7	9	11	13	15	17	19	21	23	25
C	3	6	9	12	15	18	21	24	0	3	6	9	12	15	18	21	24	0	3	6	9	12	15	18	21	24
D	4	8	12	16	20	24	1	5	9	13	17	21	25	2	6	10	14	18	22	26	3	7	11	15	19	23
E	5	10	15	20	25	3	8	13	18	23	1	6	11	16	21	26	4	9	14	19	24	2	7	12	17	22
F	6	12	18	24	3	9	15	21	0	6	12	18	24	3	9	15	21	0	6	12	18	24	3	9	15	21
G	7	14	21	1	8	15	22	2	9	16	23	3	10	17	24	4	11	18	25	5	12	19	26	6	13	20
H	8	16	24	5	13	21	2	10	18	26	7	15	23	4	12	20	1	9	17	25	6	14	22	3	11	19
I	9	18	0	9	18	0	9	18	0	9	18	0	9	18	0	9	18	0	9	18	0	9	18	0	9	18
J	10	20	3	13	23	6	16	26	9	19	2	12	22	5	15	25	8	18	1	11	21	4	14	24	7	17
K	11	22	6	17	1	12	23	7	18	2	13	24	8	19	3	14	25	9	20	4	15	26	10	21	5	16
L	12	24	9	21	6	18	3	15	0	12	24	9	21	6	18	3	15	0	12	24	9	21	6	18	3	15
M	13	26	12	25	11	24	10	23	9	22	8	21	7	20	6	19	5	18	4	17	3	16	2	15	1	14
N	14	1	15	2	16	3	17	4	18	5	19	6	20	7	21	8	22	9	23	10	24	11	25	12	26	13
O	15	3	18	6	21	9	24	12	0	15	3	18	6	21	9	24	12	0	15	3	18	6	21	9	24	12
P	16	5	21	10	26	15	4	20	9	25	14	3	19	8	24	13	2	18	7	23	12	1	17	6	22	11
Q	17	7	24	14	4	21	11	1	18	8	25	15	5	22	12	2	19	9	26	16	6	23	13	3	20	10
R	18	9	0	18	9	0	18	9	0	18	9	0	18	9	0	18	9	0	18	9	0	18	9	0	18	9
S	19	11	3	22	14	6	25	17	9	1	20	12	4	23	15	7	26	18	10	2	21	13	5	24	16	8
T	20	13	6	26	19	12	5	25	18	11	4	24	17	10	3	23	16	9	2	22	15	8	1	21	14	7
U	21	15	9	3	24	18	12	6	0	21	15	9	3	24	18	12	6	0	21	15	9	3	24	18	12	6
V	22	17	12	7	2	24	19	14	9	4	26	21	16	11	6	1	23	18	13	8	3	25	20	15	10	5
W	23	19	15	11	7	3	26	22	18	14	10	6	2	25	21	17	13	9	5	1	24	20	16	12	8	4
X	24	21	18	15	12	9	6	3	0	24	21	18	15	12	9	6	3	0	24	21	18	15	12	9	6	3
Y	25	23	21	19	17	15	13	11	9	7	5	3	1	26	24	22	20	18	16	14	12	10	8	6	4	2
Z	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

Tabelle 1: Multiplikative Verschlüsselung mit den Nummern der Schlüsselbuchstaben (Verknüpfungstafel).

dierenden Schlüsselbuchstaben immer *26* ergeben, beim *13.* Buchstaben („M“) sind beide identisch. Für praktische Experimente mit dem Caesar-Code gibt es zahlreiche Applets und Programme im Netz. Hier sollen beispielhaft nur das Krypto-Programm von Michael Kühn (http://www.kuehnsoft.de/krypto.php), die Seiten vom Mathe-Prisma der Uni Wuppertal (http://www.matheprisma.uni-wuppertal.de/Module/Caesar/index.htm) und das sehr empfehlenswerte freie Programm *Cryptool* (http://www.cryptool.de/) erwähnt werden.

Wenn man durch modulare Addition verschlüsseln kann, sollte dies auch mit der modularen Multiplikation möglich sein. Wegen der Übersichtlichkeit benutzen wir in der Tabelle 1 nur die Nummern der Buchstaben. Alle möglichen multiplikativen Verschlüsselungen erhalten wir durch die folgende Verknüpfungstafel modulo 27 (die Null wurde hier weggelassen, da sie für die multiplikative Verschlüsselung ungeeignet ist).

Den Schülerinnen und Schülern fällt bald auf, dass C, F, I, L, O, R, U und X als Schlüsselbuchstaben offenbar unbrauchbar sind, da die Verschlüsselung in diesen Fällen nicht mehr umkehrbar ist. In diesen Fällen treten in den Zeilen und Spalten jeweils Nullen auf.

Als nächstes wird analog zur additiven Verschlüsselung der Zusammenhang der Schlüsselbuchstaben zum Ver- und Entschlüsseln untersucht. Man findet heraus, dass das Produkt der beiden zugehörigen Zahlen Eins sein muss. Aus der Tabelle liest man dann z. B. ab, dass S und J zusammengehörige Schlüsselbuchstaben sind. Man sagt, dass die entsprechenden Nummern (*19* und *10*) modulo *27* invers zueinander sind. In der Tat liefert $19 \cdot 10 = 190$ dividiert durch *27* den Rest *1*. Wir bemerken noch, dass Zeilen und Spalten, die eine Eins enthalten, keine Null haben (und umgekehrt).

Um herauszufinden, unter welchen mathematischen Voraussetzungen eine modulare Inverse existiert, werden andere „Alphabete“ mit variierender Buchstabenzahl betrachtet, z. B. *9* oder *10*. Wir benutzen dafür die Verknüpfungstafeln modulo *10* bzw. *11*:

m = 10

	A	B	C	D	E	F	G	H	I
A	1	2	3	4	5	6	7	8	9
B	2	4	6	8	0	2	4	6	8
C	3	6	9	2	5	8	1	4	7
D	4	8	2	6	0	4	8	2	6
E	5	0	5	0	5	0	5	0	5
F	6	2	8	4	0	6	2	8	4
G	7	4	1	8	5	2	9	6	3
H	8	6	4	2	0	8	6	4	2
I	9	8	7	6	5	4	3	2	1

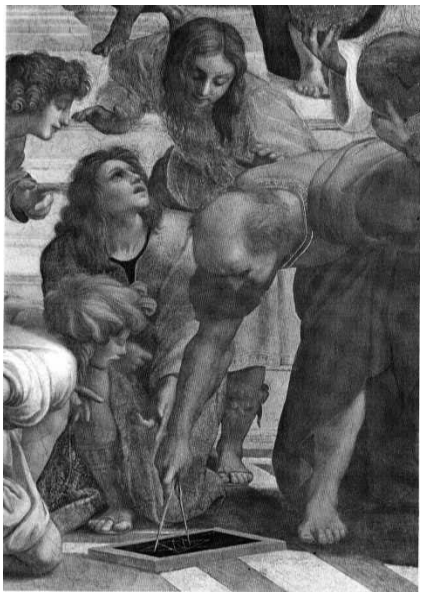
m = 11

	A	B	C	D	E	F	G	H	I	J
A	1	2	3	4	5	6	7	8	9	10
B	2	4	6	8	10	1	3	5	7	9
C	3	6	9	1	4	7	10	2	5	8
D	4	8	1	5	9	2	6	10	3	7
E	5	10	4	9	3	8	2	7	1	6
F	6	1	7	2	8	3	9	4	10	5
G	7	3	10	6	2	9	5	1	8	4
H	8	5	2	10	7	4	1	9	6	3
I	9	7	5	3	1	10	8	6	4	2
J	10	9	8	7	6	5	4	3	2	1

Hierbei fällt auf, dass für $m = 10$ alle Buchstaben mit geraden Nummern sowie mit der Nummer *5* ungeeignet sind. Für $m = 11$ eignen sich dagegen alle Buchstaben für die multiplikative Verschlüsselung. Die Lernenden vermuten, dass der größte gemeinsame Teiler (ggT) der Schlüsselzahl und des Moduls Eins sein muss. Man sagt in diesem Fall, dass die Zahlen relativ prim zueinander sind (vgl. dazu auch den Kasten „Regeln für modulares Rechnen“ aus Witten/Schulz 2006, S. 49).

Die Bestimmung des ggT kann in den aufgeführten Beispielen nach der Grundschulmethode erfolgen: Man bestimmt für beide Zahlen die Primteiler, das Produkt der gemeinsamen auftretenden Primteiler ist dann der ggT.

Dieses Verfahren ist sehr bequem, wenn man die Primteiler im Kopf bestimmen kann. Wir werden uns aber weiter unten mit sehr großen Zahlen beschäftigen,



Quelle: LOG-IN-Archiv

Bild 2: Euklid (ca. 365 v. Chr. bis ca. 300 v. Chr.) – ein Ausschnitt aus Raffaels „Die Schule von Athen“ (La scuola di Atene), Fresko in der Stanza della Segnatura des Vatikans.

und dafür ist dieses Verfahren völlig ungeeignet. Zum Glück gibt es ein sehr effizientes Verfahren, mit dem man den ggT auch ohne Kenntnis der Primfaktorzerlegung bestimmen kann. Das ist der berühmte euklidische Algorithmus.

Ehrenrettung für Euklid

Norbert Breier, der den Leserinnen und Lesern dieser Zeitschrift sicherlich nicht unbekannt ist, hat auf einem Vortrag vor Lehrerinnen und Lehrern in Braunschweig die Behandlung des euklidischen Algorithmus im Informatikunterricht als negatives Beispiel angeführt. Seiner Ansicht nach hat dieser Algorithmus nichts in einem zeitgemäßen Unterricht zu suchen.

Nun ist es sicherlich so, dass dieser Algorithmus häufig losgelöst von Anwendungen behandelt wird. Im Besonderen ist der euklidische Algorithmus ein beliebtes Beispiel, um iterative und rekursive Lösungen zu vergleichen. Im Fall der Behandlung von RSA im Unterricht ist aber das Verständnis dieses Algorithmus zum einen zwingend erforderlich, zum anderen wird er eingebettet in Sinn- und Sachzusammenhänge behandelt.

Der euklidische Algorithmus ist der älteste bekannte nicht-triviale Algorithmus. Er findet sich in den „Elementen“ (Buch VII, Proposition 1 und 2, Veröffentlichung um 300 v. Chr). Das Verfahren wurde jedoch wahrscheinlich nicht von Euklid erfunden, sondern war schon bis zu 200 Jahre früher bekannt (vgl. auch Wikipedia, Stichwort „Euklidischer Algorithmus“).

Euklid nannte das Verfahren „Wechselwegnahme“ und formulierte es als geometrisches Problem: „Wenn CD aber AB nicht misst, und man nimmt bei AB, CD abwechselnd immer das Kleinere vom Größeren weg, dann muss (schließlich) eine Zahl übrig bleiben, die die vorangehende misst“ (siehe Bild 3).

Heute wird der euklidische Algorithmus nicht mehr durch fortgesetzte Subtraktion ausgeführt, sondern durch Teilen mit Rest (s. Kasten „Der euklidische Algorithmus“, nächste Seite). Als PYTHON-Programm (zur Programmiersprache PYTHON vgl. Arnhold, 2001) lässt er sich folgendermaßen formulieren (iterative Version):

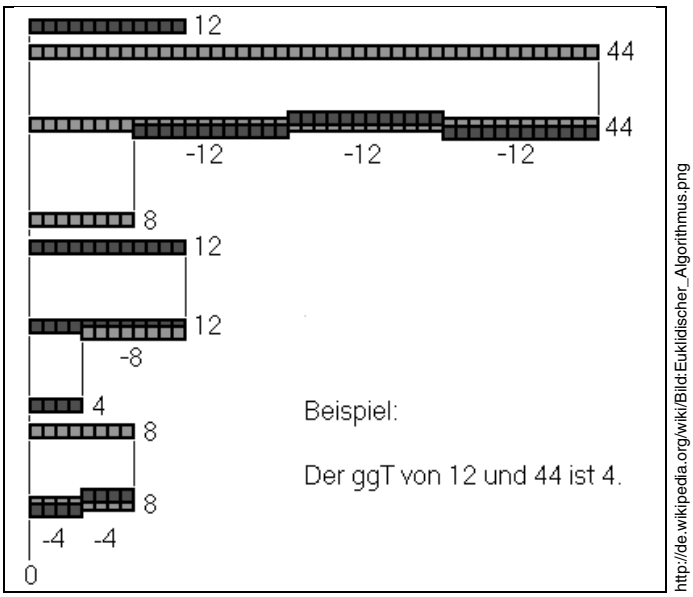
```
def ggT (a,b):
    '''Berechnung des ggT mit dem euklidischen
                        Algorithmus.'''
    aalt, amitte = a, b
    while amitte <> 0:
        aneu = aalt % amitte
        aalt, amitte = amitte, aneu
    return aalt
```

Eleganter ist die rekursive Variante im funktionalen Programmierstil (hierbei wird ggT*(a, b)* = ggT*(b, a%b)* verwendet, s. Kasten „Der euklidische Algorithmus“). Allerdings muss man hierfür, wenn die Funktion nicht nur einfach angewandt werden soll, das Konzept der Rekursion verstanden haben:

```
def ggT_rek (a,b):
    '''Rekursive Berechnung des ggT mit dem
            euklidischen Algorithmus.'''
    if b == 0: return a
    return ggT_rek (b, a%b)
```

Man kann zeigen, dass die Zahl der Iterationsschritte bzw. der rekursiven Aufrufe von der Größenordnung *O(*log*(b))* ist, wobei *b* der zweite Eingabeparameter ist (vgl. Schöning, 1997, S. 171). Der höchste Rechenaufwand ergibt sich im Übrigen, wenn *a* und *b* zwei aufeinander folgende Fibonacci-Zahlen sind, da sich dann als Rest immer die nächst kleinere Fibonacci-Zahl ergibt (vgl. auch z. B. Wikipedia, Stichwort „Euklidischer Algorithmus“).

Mit diesen Vorbereitungen können wir die multiplikative Verschlüsselung auch bei sehr großen Zahlen („Monsterzahlen“) durchführen. Nach einer Idee von Hermann Puhlmann kann man damit auch die asym-



http://de.wikipedia.org/wiki/Bild:Euklidischer_Algorithmus.png

Bild 3: Euklidischer Algorithmus zur Bestimmung des größten gemeinsamen Teilers durch Wechselwegnahme.

metrischen Kryptosysteme einführen (vgl. Puhlmann, 1998). Außerdem motiviert dieser Weg die Frage, ob es effiziente Verfahren zur Bestimmung der modularen Inversen gibt.

Asymmetrische Kryptographie durch multiplikative Verschlüsselung mit Monsterzahlen

Wir wählen als Modul
$n = 10000100001000010000100001000010000100001000010000100001000010000$
und als Schlüsselzahl
$s = 77551795926122526531387762857220408938783265383674244 9$.

Man überzeugt sich mit der Funktion ggT, dass n und s teilerfremd sind. Damit ist s als Schlüsselzahl geeignet. Die Botschaft „KRYPTOLOGIE_MACHT_SPASS“ wird mit der Darstellung _ = 00, A = 01, B = 02, …, Z = 26 zu
$m = 11182516201512150709050013010308200019160119 19$,
wenn die Zahlen einfach hintereinander geschrieben und als „Monsterzahl“ gelesen werden. Da $m < n$ gilt, kann man m mit $m \cdot s$ modulo n verschlüsseln. Es ergibt sich als Chiffre
$c = 22821461635739083079693904102669795957469631$.

Wie kann man sich überzeugen, dass sich c wieder zu m entschlüsseln lässt? Dazu benötigen wir die modulare Inverse k mit $k \cdot s \equiv 1 \bmod n$. Der naive Ansatz ist, k durch simples Ausprobieren zu finden. Wir verwenden dazu ein kleines PYTHON-Programm:

```
k = 1
while True: # Endlosschleife!
    if k*s%n == 1:
        print k
        break
    k = k+1
print "gefunden!"
```

In der Tat wird nach kurzer Zeit $k = 49$ ausgegeben; hier war die modulare Inverse einfach zu finden. Zur Kontrolle wird c mit k *modulo* n multipliziert und man erhält tatsächlich wieder die ursprüngliche Botschaft m.

Offenbar wäre es geschickter gewesen, k als Schlüsselzahl zu wählen, denn dann ist die modulare Inverse vermutlich schwieriger zu finden. Nachdem k und s vertauscht wurden, liefert der naive Test nach einigen Minuten kein Ergebnis. Dies legt den Gedanken nahe, den benötigten Zeitaufwand experimentell abzuschätzen. Dazu wird in der Schleife ein print-Befehl eingebaut, der nach einer Million Versuchen jeweils eine Meldung ausgibt:

```
if k%1000000 == 0: print "Anzahl Versuche:", k
```

Man misst auf einem durchschnittlichen PC für $10000000 = 10^7$ Versuche ca. 10 s. Die gesuchte Schlüsselzahl hat aber 54 Stellen, sodass die benötigte Zeit etwa 10^{47}-mal so lang ist, also in der Größenordnung von 10^{48} s liegt. Unter dieser Zahl kann man sich nicht allzu viel vorstellen, sie soll deshalb in Jahre umgerechnet werden. Ein Jahr hat etwa $60 \cdot 60 \cdot 24 \cdot 365{,}24$ s = 31556736 s. 10^{48} geteilt durch diese Zahl ergibt dann ungefähr $3{,}17 \cdot 10^{40}$ Jahre. Spätestens wenn man bedenkt, dass das Alter des Universums etwa $1{,}5 \cdot 10^{10}$ Jahre beträgt, sieht man ein, dass dieses Vorgehen total sinnlos ist, selbst wenn die verwendeten Rechner millionenfach schneller würden.

Man könnte also mit der multiplikativen Chiffre ein asymmetrisches Kryptosystem aufbauen. Man würde die kleine Zahl (in unserem Beispiel 49) als öffentlichen Schlüssel verwenden, die große Schlüsselzahl in den Tresor einschließen und nur für die Entschlüsselung geheimer Nachrichten herausholen. Außerdem könnte man diese geheime Schlüsselzahl auch zum Sig-

Der euklidische Algorithmus

1. Teilen mit Rest

Zu $a, b \in \mathbb{N}$ existieren $q, r \in \mathbb{N} \cup \{0\}$ mit
$a = q \cdot b + r$ und $r < b$.
Hierbei ist $r = R_b(a)$
(in anderer Schreibweise $a \bmod b$ oder $r = a\%b$).

Anmerkung: Es gilt $\mathrm{ggT}(a, b) = \mathrm{ggT}(b, r) = \mathrm{ggT}(b, a \bmod b)$.

Beweisskizze zur Anmerkung:
$d = \mathrm{ggT}(a, b)$ teilt a und b, somit auch $r = a - q \cdot b$.
Umgekehrt ist jeder Teiler von b und r auch Teiler von a.

2. Idee des euklidischen Algorithmus

Fortgesetzte Division mit Rest – bis der Rest 0 ist.

Ursprung: Geometrische Kommensurabilitäts-Betrachtungen, schon bei Euklid: Die Strecken der Längen a und b lassen sich mit einem Maßstab der Länge $\mathrm{ggT}(a, b)$ ausmessen.

Algorithmus:

$$\begin{aligned}
a &= q_1 b + a_3 \\
b &= q_2 a_3 + a_4 \\
a_3 &= q_3 a_4 + a_5 \\
&\vdots \\
a_{\text{alt}} &= q \cdot a_{\text{mitte}} + a_{\text{neu}} \\
&\vdots \\
a_{n-1} &= q_{n-1} \cdot a_n + a_{n+1} \quad \text{mit } a_{n+1} \neq 0 \\
a_n &= q_n \cdot a_{n+1} + 0
\end{aligned}$$

$$\boxed{a_{\text{neu}} = a_{\text{alt}} \% a_{\text{mitte}}}$$

Ergebnis: $a_{n+1} = \mathrm{ggT}(a, b)$

Beweisskizze:
Der Rest 0 tritt schließlich wegen $b > a_3 > a_4 \ldots \geq 0$ auf.
Ferner gilt
$\mathrm{ggT}(a, b) = \ldots = \mathrm{ggT}(a_{\text{alt}}, a_{\text{mitte}}) = \mathrm{ggT}(a_{\text{mitte}}, a_{\text{neu}}) = \ldots$
$= \mathrm{ggT}(a_{n+1}, 0) = a_{n+1}$.

Beispiel

Gesucht ist ggT(48, 18).

$$\begin{aligned}
&(1)\quad \underline{48} = 2 \cdot \underline{18} + \underline{12} \\
&(2)\quad \underline{18} = 1 \cdot \underline{12} + \underline{\underline{6}} \\
&(3)\quad \underline{12} = 2 \cdot \underline{\underline{6}} + \underline{\underline{0}} \\
&(4)\quad \underline{\underline{6}}
\end{aligned}$$

Also gilt ggT(48, 18) = 6.

Bild 4:
Claude Gaspard Bachet de Méziriac (1581–1638).

http://www.academie-francaise.fr/images/immortels/portraits/meziriac.jpg

nieren von Nachrichten verwenden – wenn es nicht doch einen effizienteren Algorithmus zur Bestimmung der modularen Inversen gäbe.

Der erweiterte euklidische Algorithmus

Leider oder zum Glück gibt es diesen effizienten Algorithmus – er beruht auf dem erweiterten euklidischen Algorithmus und hat ebenfalls logarithmische Laufzeit (der naive Algorithmus hat offenbar lineare Laufzeit – und das ist bei Monsterzahlen schon viel zu viel).

Der erweiterte euklidische Algorithmus geht auf das erste Buch zur Unterhaltungsmathematik „problèmes plaisans et délectables qui se font par les nombres“ (Lyon, 1612) zurück. Es wurde von Claude Gaspar Bachet de Méziriac verfasst. Die Idee war, dass man den ggT*(a, b)* als Linearkombination von *a* und *b* darstellen kann (zum Lemma von Bachet siehe Kasten „Der erweiterte euklidische Algorithmus“). Dieses Lemma wird in der Literatur häufig Étienne Bézout (1730–1783) zugeschrieben, der es von Zahlen auf Polynome verallgemeinerte (vgl. z. B. Wikipédia, Stichwort „Identité de Bézout“).

Betrachtet man bei festen vorgegebenen positiven Zahlen *a* und *b* die Menge aller Zahlen, die sich als Linearkombination $x \cdot a + y \cdot b$ mit beliebigen ganzen Zahlen *x* und *y* darstellen lassen, ergibt sich der ggT von *a* und *b* als kleinste positive Zahl aus dieser Menge (zum Beweis siehe z. B. Schöning, 1997, S. 171 f.). *x* und *y* sind dabei nicht eindeutig bestimmt. Dies zeigt das folgende Beispiel: Der ggT von 12 und 42 ist 6, hierfür gilt u. a. $6 = (-3) \cdot 12 + 1 \cdot 42 = 4 \cdot 12 + (-1) \cdot 42$.

Für die Berechnung eines der Paare *(x, y)* zur Darstellung von ggT*(192, 41)* gehen wir wieder vom (normalen) euklidischen Algorithmus aus.

a $192 = 4 \cdot 41 + 28$
b $41 = 1 \cdot 28 + 13$
c $28 = 2 \cdot 13 + 2$
d $13 = 6 \cdot 2 + 1$
e $2 = 2 \cdot 1 + 0$

Also ist ggT*(192,41) = 1*.

Der erweiterte euklidische Algorithmus

Ziel: Auffinden von Zahlen $x, y, \in \mathbb{Z}$ mit $\text{ggT}(a, b) = x \cdot a + y \cdot b$.

Diese existieren gemäß *Vielfachsummensatz* (Lemma von Bachet, 1612).

Idee: Bei jedem Schritt des euklidischen Algorithmus ist der neue Rest Summe von Vielfachen von *a* und *b*, sodass man nur Buch zu führen braucht.

Start: $a = 1 \cdot a + 0 \cdot b$ und $b = 0 \cdot a + 1 \cdot b$.

Sei $a_{\text{alt}} = q \cdot a_{\text{mitte}} + a_{\text{neu}} = x_{\text{alt}} a + y_{\text{alt}} b$
und $a_{\text{mitte}} = x_{\text{mitte}} a + y_{\text{mitte}} b$!

Dann folgt

$$\begin{aligned} a_{\text{neu}} &= a_{\text{alt}} - q \cdot a_{\text{mitte}} \\ &= x_{\text{alt}} \cdot a + y_{\text{alt}} \cdot b - q(x_{\text{mitte}} \cdot a + y_{\text{mitte}} \cdot b) \\ &= \underbrace{(x_{\text{alt}} - q \cdot x_{\text{mitte}})}_{x_{\text{neu}}} \cdot a + \underbrace{(y_{\text{alt}} - q \cdot y_{\text{mitte}})}_{y_{\text{neu}}} \cdot b. \end{aligned}$$

Daher setzt man:

$x_{\text{neu}} = x_{\text{alt}} - q \cdot x_{\text{mitte}}$ und $y_{\text{neu}} = y_{\text{alt}} - q \cdot y_{\text{mitte}}$

für $q = \left\lfloor \frac{a_{\text{alt}}}{a_{\text{mitte}}} \right\rfloor$

Erweiterter euklidischer Algorithmus zur Berechnung von ggT(a, b) und von ganzen Zahlen x, y mit $\text{ggT}(a, b) = x \cdot a + y \cdot b$:

```
def Bachet(a, b):
    aalt, amitte = a, b
    xalt, xmitte = 1, 0
    yalt, ymitte = 0, 1
    While amitte <> 0 :
        q= aalt / amitte
        aneu= aalt % amitte
        xneu= xalt - xmitte *q
        yneu= yalt - ymitte *q
        xalt, xmitte = xmitte, xneu
        yalt, ymitte = ymitte, yneu
        aalt, amitte = amitte, aneu
return aalt, xalt, yalt
```

Ausgabe: ggT$(a, b), x, y$ (mit $\text{ggT}(a, b) = x \cdot a + y \cdot b$).

Beispiel

$60 = 2 \cdot 23 + 14$ $\quad x_3 = 1 - 2 \cdot 0 = 1$ $\quad y_3 = 0 - 2 \cdot 1 = -2$
$14 = x_3 \cdot 60 + y_3 \cdot 23$
$= 1 \cdot 60 - 2 \cdot 23$

$23 = 1 \cdot 14 + 9$ $\quad x_4 = 0 - 1 \cdot 1 = -1$ $\quad y_4 = 1 - 1 \cdot (-2) = 3$
$9 = x_4 \cdot 60 + y_4 \cdot 23$
$= -1 \cdot 60 + 3 \cdot 23$

usw.

Ergebnis: $\text{ggT}(60, 23) = 1 = 5 \cdot 60 + (-13) \cdot 23$.

Die Gleichungskette wird jetzt von unten nach oben aufgelöst. Dabei werden die Reste so lange durch die aus den vorangehenden Zeilen ersetzt, bis man bei den Ausgangszahlen angelangt ist:

Aus d folgt $1 = 13 - 6 \cdot 2$,
mit c folgt $1 = 13 - 6(28 - 2 \cdot 13)$,
zusammengefasst $1 = 13 \cdot 13 - 6 \cdot 28$,
mit b folgt $1 = 13(41 - 1 \cdot 28) - 6 \cdot 28$,
zusammengefasst $1 = 13 \cdot 41 - 19 \cdot 28$,
mit a folgt $1 = 13 \cdot 41 - 19(192 - 4 \cdot 41)$,
zusammengefasst $1 = 89 \cdot 41 - 19 \cdot 192$.

Damit haben wir die Darstellung
ggT$(192, 41) = -19 \cdot 192 + 89 \cdot 41$ gefunden.

Diese Vorgehensweise lässt sich direkt mit der folgenden eleganten rekursiven Funktionen nachbilden (s. Schöning, 1997, S. 172; dort findet man auch einen formalen Korrektheitsbeweis):

```
def Bachet_rek (a,b):
    if b == 0: return (a,1,0)
    ggT, x, y = Bachet_rek(b, a%b)
    return ggT, y, x - (a/b)*y
```

Zunächst wird der ggT mit dem üblichen (rekursiven) Algorithmus bestimmt. Kehrt die Funktion aus den jeweiligen rekursiven Aufrufen zurück, werden die Parameter *x* und *y* schrittweise wie im oben stehenden Beispiel berechnet. Wenn man geeignete print-Anweisungen einbaut, kann man dies auch experimentell nachvollziehen.

Im Kasten „Der erweiterte euklidische Algorithmus“ (siehe vorige Seite) findet man die etwas umständlichere iterative Version zur Berechnung der Vielfachsummendarstellung des ggT. Die wichtigste Anwendung dieses Algorithmus (der manchmal auch als „Algorithmus von Berlekamp“ bezeichnet wird) ist die Berechnung der modularen Inversen.

Die Bestimmung der multiplikativen Inversen

Wir wollen jetzt die modulare Inverse zu *41* bezüglich des Moduls *192* bestimmen. Wie wir beim letzten Beispiel gesehen haben, ist dies möglich, weil beide relativ prim zueinander sind. Wenn wir die oben gewonnene Gleichung $1 = -19 \cdot 192 + 89 \cdot 41$ modulo *192* betrachten, ergibt sich die Kongruenz $89 \cdot 41 \equiv 1 \bmod 192$, d. h. *89* ist die modulare Inverse zu *41*, da $-19 \cdot 192$ modulo *192* zu Null wird.

Eine Funktion zur Bestimmung der modularen Inversen könnte also so aussehen:

```
def modinv(k,n):
    ggT, x, y = Bachet(n,k)
    if ggT > 1 : return -1
    if y < 0: y = y + n
    return y
```

Falls der ggT größer als *1* ist, kann keine modulare Inverse berechnet werden. In diesem Fall wird der Wert *–1* zurückgegeben. Da der Parameter *y* auch negativ sein kann, muss ggf. *n* addiert werden. Mit dieser Funktion kann man dann – wegen der logarithmischen Laufzeit – problemlos die modulare Inverse zu *49* bezüglich des Moduls
$m = 10000100001000010000100001000010000100001000010000100001000010000$

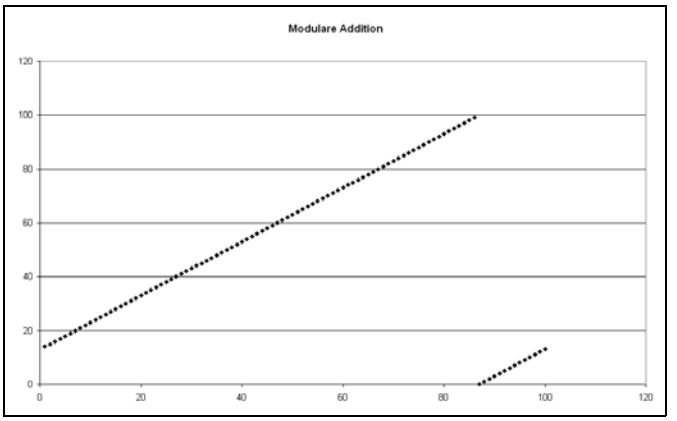
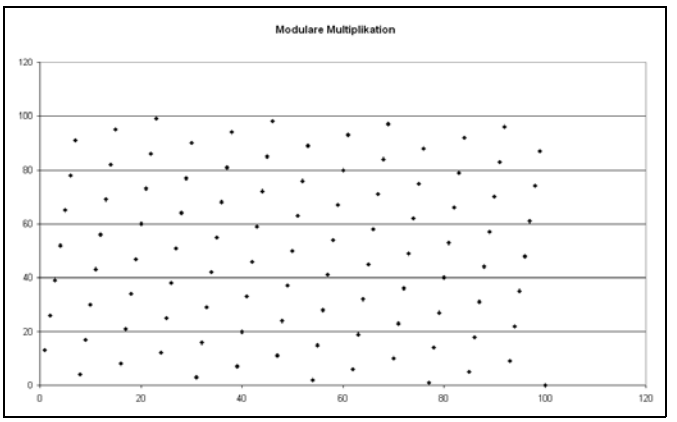


Bild 5 (oben): Der Funktionsgraph von $x+13$ modulo 100.

Bild 6 (unten): Der Funktionsgraph von $x \cdot 13$ modulo 100.



berechnen. Dabei ist für die iterative oder die rekursive Variante der Funktion Bachet kein Unterschied der Laufzeit zu bemerken. Auf jeden Fall ist damit die Zukunft der multiplikativen Verschlüsselung als asymmetrisches Kryptosysten schon vorbei, bevor sie richtig begonnen hat.

Rivest, Shamir und Adleman hatten die Idee, dass man für ein sicheres asymmetrisches Verschlüsselungssystem modulares Potenzieren verwenden kann. Man kann sich mit den Funktionsgraphen von $x+13$, $x \cdot 13$ und x^{13} (jeweils modulo *100*) leicht klarmachen, dass die Schwierigkeit, eine Umkehrfunktion zu finden, mit der nächst höheren Rechenart dramatisch zunimmt („modulares Chaos“, siehe Bilder 5 und 6, sowie Bild 7, nächste Seite). Der Definitionsbereich dieser Funktionen besteht jeweils aus den natürlichen Zahlen kleiner gleich 100.

Penible Lämmergeier

RSA erblickte im August 1977 das Licht der Öffentlichkeit (Gardner, 1977), obwohl der wissenschaftliche Artikel von Rivest, Shamir und Adleman erst 1978 erschien (Rivest/Shamir/Adleman, 1978). Martin Gard-

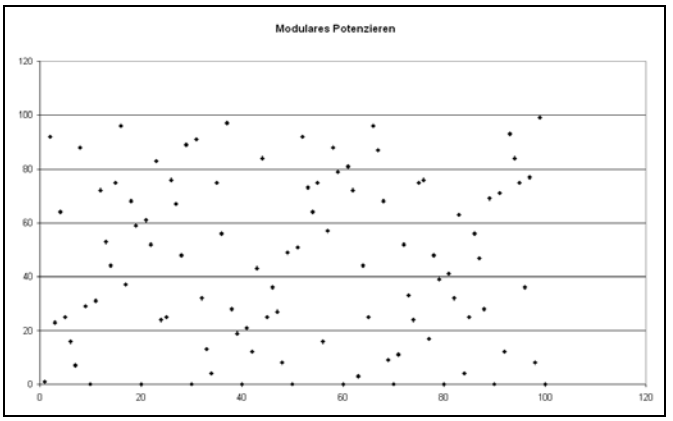


Bild 7: Der Funktionsgraph von x^{13} modulo *100*.

ner hatte von dieser Entwicklung gehört und stellte das Verfahren in seiner Kolumne „Mathematische Spiele“ für den „Scientific American“ (deutsch: „Spektrum der Wissenschaft“) vor. Der Artikel hatte die Überschrift „A new kind of cipher that would take millions of years to break“. Es wird berichtet, dass Vertreter der mächtigen NSA (***N**ational **S**ecurity **A**gency*) versucht haben, Ron Rivest von der Veröffentlichung abzuhalten, weil angeblich das Sicherheitsbedürfnis der USA verletzt würde – ein Unterfangen, das nicht zuletzt durch den Artikel von Martin Gardner gescheitert ist (vgl. Schmeh, 2004, S. 220 ff.).

Gardner druckte den öffentlichen Schlüssel, bestehend aus einer 129-stelligen Zahl und dem Exponenten $e = 9007$, in seiner Kolumne ab. Die 129-stellige Zahl ist heute unter dem Namen RSA-129 bekannt (vgl. z. B. Wikipedia, Stichwort „RSA Factoring Challenge“) und lautet

114381625757888867669235779976146612010218296721242362562561842935706935245733897830597123563958705058980751475992900026879543541.

Die Texte wurden nach dem gleichen Schema – wie oben bei der multiplikativen Chiffre angegeben – in große Zahlen verwandelt. Der zu entschlüsselnde Geheimtext lautet

c = 96869613754622061477140922254355882905759991124574319874695120930816298225145708356931476622883989628013391990551829945157815154.

Außerdem wurde noch eine Signatur mitgeteilt, die mit dem geheimen Schlüssel chiffriert war und von jedermann mit dem öffentlichen Schlüssel verifiziert werden konnte. Die Signatur lautete

s = 16717861150380844246015271389168398245436901032358311217835038446929062655448792237114490509578608655662496577974840004057020373.

Durch Berechnung von s^e modulo der Zahl RSA-129 mit dem in der letzten Folge vorgestellten Verfahren zur schnellen Potenzierung (vgl. Witten/Schulz, 2006) ergibt sich (mit einer vorangestellten Null):

0609181920001915122205180023091419001514050008211404180504000415121201 1819.

Nach dem oben angegebenen Schema wird damit der Text

FIRST_SOLVER_WINS_ONE_HUNDRED_DOLLARS

dargestellt.

Es hat dann doch nicht Millionen Jahre, sondern „nur“ 17 Jahre gedauert, bist dieser Code geknackt wurde (vgl. z. B. Wikipedia, Stichwort „RSA-129“), die eigentliche Projektdauer betrug mehr als 6 Monate. Einem Team unter der Leitung von Derek Atkins, Michael Graff, Arjen K. Lenstra und Paul Leyland gelang 1994 mit 600 Freiwilligen und 1600 über das Internet zusammengeschalteten Computern die Faktorisierung von RSA-129 in die beiden Primzahlen

p = 3490529510847650949147849619903898133417764638493387843990820577

und

q = 3276913299326670954996198819083446141317764296799294253979828853 3.

Die 100 Dollar Preisgeld wurden im Übrigen der *Free Software Foundation* gespendet.

Dass diese Faktorisierung korrekt ist, kann man leicht mit der Langzahlarithmetik des PYTHON-Interpreters nachprüfen; entsprechende Experimente mit DERIVE werden bei Johann Wiesenbauer (1999) beschrieben.

Wir sind jetzt in der Lage, den privaten Schlüssel von Rivest, Shamir und Adleman zu berechnen und damit den Code zu knacken. Für den öffentlichen Schlüssel e und den privaten Schlüssel d gilt die Beziehung $e \cdot d \equiv 1 \mod \varphi(n)$ mit $\varphi(n) = (p-1)(q-1)$, mit anderen Worten: e und d sind bezüglich $\varphi(n)$ zueinander modular invers. Mit der Funktion modinv ergibt sich dann sofort

d = 106698614368578024442868771328920154780709906633937862801226224496631063125911774470873340168597462306553968544513277109053606095,

und man erhält damit die entschlüsselte Botschaft

m = 200805001301070903002315180419000118050019172105011309190800151919090618010705.

In Buchstaben umgewandelt ergibt sich der kryptische Text

THE_MAGIC_WORDS_ARE_SQUEAMISH_OSSIFRAGE

(übersetzt: Die magischen Worte sind penible Lämmergeier – vgl. z. B. auch Wikipedia, Stichwort „The Magic Words are Squeamish Ossifrage“).

Seit 1993/94 werden die „peniblen Lämmergeier“ häufig bei kryptoanalytischen Wettbewerben verwendet. Man fragt sich, was sich Rivest, Shamir und Adleman bei der Wahl dieses Klartextes gedacht haben, was Lämmergeier mit der Kryptologie zu tun haben. Beim Adjektiv „squeamish“ (penibel, zimperlich) ist das ziemlich klar: Ohne große Sorgfalt scheitert man bei der Kryptoanalyse.

„Ossifrage“ ist ein altertümliches Wort für Lämmergeier, für das sich inzwischen im Englischen der Germanizismus „Lammergeier“ durchgesetzt hat (vgl. auch Wikipedia, Stichwort „Lammergeier“). Im Deutschen spricht man heute allerdings eher vom Bartgeier (vgl. Wikipedia, Stichwort „Bartgeier“). Ossifrage könnte man wörtlich mit „Knochenbrecher“ übersetzten. Damit wird auf die spezielle Art der Nahrungsaufnahme dieser Aasfresser angespielt: Sie lassen Knochen (und mitunter auch Schildkröten) aus großer Höhe fallen, um sie aufzubrechen und dann das Mark zu fressen. Damit dürfte eine weitere Beziehung zum Codebrechen klar sein.

Von dem griechischen Dichter Aischylos, der 456 v. Chr. in Gela starb, erzählt die von Plinius dem Älteren überlieferte Legende, dass er dadurch zu Tode kam,

Bild 8: Ein Bart- oder Lämmergeier (Gypaetus barbatus) – als Ossifrage Gegenstand eines berühmten Beispiels aus der Kryptoanalyse.

Quelle: Johann Andreas Naumann, „Naturgeschichte der Vögel Mitteleuropas", Band V, Tafel 60; Gera, ²1899.

dass ein Lämmergeier eine Schildkröte auf seinen kahlen Kopf herabfallen ließ. Diese Geschichte passt dazu, dass in den USA kryptologische Verfahren als Waffen eingeschätzt werden und daher Exportkontrollen unterliegen.

RSA Challenge

Es ist deutlich geworden, dass die Sicherheit von RSA vor allem von der Größe des Moduls *n* abhängt. Während es sehr einfach ist, *n* aus seinen beiden Primfaktoren *p* und *q* zu berechnen, ist die Umkehrung bei genügend großem *n* praktisch unmöglich – wenigstens so lange noch kein effizienteres Verfahren zur Faktorisierung gefunden wurde. RSA ist leicht zu „knacken", wenn die Primfaktoren von *n* bekannt sind. Daher ist es unsinnig, für *n* kleine Zahlen zu verwenden.

Um zu demonstrieren, welche Zahlen als sicher gelten könnten, wird von der Firma *RSA Laboratories* seit 1991 ein Faktorisierungswettbewerb durchgeführt (vgl. auch Wikipedia, Stichwort „RSA Factoring Challenge"). Der Rekord aus dem Jahr 2005 liegt zurzeit bei einer Zahl mit 200 Stellen im Dezimalsystem bzw. 663 Bit (vgl. RSA Laboratories). Die benötigte CPU-Zeit entsprach bei dieser Faktorisierung 55 Jahren auf einer einzelnen 2,2 GHz Opteron-CPU. Wenn man in der Presse liest „RSA-200 geknackt", bedeutet dies also keineswegs, dass RSA insgesamt unsicher geworden wäre. Es heißt nur, dass man keine 512-Bit-Schlüssel mehr verwenden sollte.

Mit einem 1024-Bit-Schlüssel ist man dagegen auf absehbare Zeit auf der sicheren Seite, für sehr hohe Sicherheitsanforderungen kann man 2048-Bit-Schlüssel verwenden. Auf der Seite zum RSA-Challenge (http://www.rsasecurity.com/rsalabs/node.asp?id=2093) findet man u.a. eine noch nicht faktorisierte Zahl mit 1024 Bit (das entspricht 309 Dezimalstellen):

135066410865995223349603216278805969938881475
605667027524485143851526510604859533833940287150571909441798207282164471551373680419703964191743046496589274256239341020864383202110372958725762358509643110564073501508187510676594629205563685529475213500852879416377328533906109750544334999811150056977236890927563

Wem es gelingt, diese Zahl zu zerlegen, erhält 100000 Dollar Preisgeld. Ist das nicht einen Versuch wert? Allerdings ist die Wahrscheinlichkeit, einen der Faktoren durch Probieren zu finden, wesentlich kleiner als diejenige, den Jackpot beim Lotto zu knacken.

Fazit und Ausblick

Der Zugang zu RSA über die multiplikative Verschlüsselung verlangt von den Schülerinnen und Schülern einen längeren Atem. Die Programmierung des erweiterten euklidischen Algorithmus ist eher für die Sekundarstufe II geeignet, während die multiplikative Verschlüsselung und die Falltüreigenschaft beim RSA-Kryptosystem anhand des Faktorisierungswettbewerbs „RSA-Challenge" auch in der Sekundarstufe I veranschaulicht werden kann.

Die Behandlung von RSA im Unterricht verbindet mathematische, politische und informatische Fragestellungen und erfüllt damit die Forderung nach fachübergreifendem und fächerverbindendem Unterricht.

In der nächsten Folge werden wir uns mit der Korrektheit von RSA und mit schnellen Primzahltests befassen. Für beides bildet der kleine Satz von Fermat die Grundlage.

Nachtrag zur letzten Folge

Rüdeger Baumann hat uns als aufmerksamer Leser unseres Artikels in einem freundlichen Brief darauf hingewiesen, dass der Algorithmus zum schnellen Potenzieren durch Quadrieren und Multiplizieren („Square and Multiply") auf Adrien-Marie Legendre (vgl. u.a. Wikipedia, Stichwort „Adrien-Marie Legendre") zurückgeht.

Dies hat unseren Forscherehrgeiz angespornt. Die Urheberschaft von Legendre konnten wir dabei leider nicht verifizieren, dafür sind wir bei Donald Ervin Knuth („The art of computer programming", Band 2, Abschnitt 4.63, S.399; Reading (MA) u.a., Addison Wesley, 1981) fündig geworden: Diese Methode wird schon 200 v. Chr. in Pingalas „Hindu classic Chandahsûtra" beschrieben (vgl. B. Datta und A.N. Singh, „History of Hindu Mathematics", Band 1; Bombay, 1935).

Prof. Dr. Ralph-Hardo Schulz
Freie Universität Berlin
Fachbereich Mathematik und Informatik
Institut für Mathematik II
Arnimallee 3
14195 Berlin

E-Mail: schulz@math.fu-berlin.de

StD Helmut Witten
Fachseminar für Informatik
1. Schulpraktisches Seminar
Charlottenburg-Wilmersdorf (S)
Walther-Rathenau-Schule (Gymnasium)
Herbertstraße 4
14193 Berlin

E-Mail: helmut@witten-berlin.de

Literatur und Internetquellen

Arnhold, W.: Lieben Sie PYTHON? In: LOG IN, 21. Jg.. (2001), H. 2, S. 18–24.

Bartholomé, A.; Rung, J.; Kern, H.: Zahlentheorie für Einsteiger. Braunschweig, Wiesbaden: Vieweg, ²1996.

Eckert, C.: IT-Sicherheit – Konzepte, Verfahren, Protokolle (Studienausgabe). München: Oldenbourg, 2005.

Gardner, M.: Mathematical Games – A New Kind of Cipher That Would Take Millions of Years to Break. In: Scientific American, Vol. 237, Nr. 2 (August 1977), S. 120–124.
http://www.totse.com/en/technology/science_technology/sciamer1.html

Puhlmann, H.: Kryptographie verstehen – Ein schülergerechter Zugang zum RSA-Verfahren. TU Darmstadt (1998). Als gezippte PostScript-Datei online verfügbar unter:
http://wwwbib.mathematik.tu-darmstadt.de/Math-Net/Preprints/Listen/files/2000.ps.gz
Eine PDF-Version erhält man bei
http://www.matheraetsel.de/texte/kryptographie.pdf

RSA-Laboratories: RSA-200 is factored!
http://www.rsasecurity.com/rsalabs/node.asp?id=2879

Rivest, R.; Shamir, A.; Adleman, L.: A Method for Obtaining Digital Signatures and Public Key Cryptosystems. Communications of the ACM, Vol. 21, Nr. 2 (1978), S. 120-126.
http://theory.lcs.mit.edu/~rivest/rsapaper.pdf

Schmeh, K.: Die Welt der geheimen Zeichen – Die faszinierende Geschichte der Verschlüsselung. Herdecke; Dortmund: W3L-Verlag, 2004.

Schmidt, M.: Späte Ehre für RSA-Erfinder Rivest, Shamir und Adleman. Heise online news, 15.04.2003.
http://www.heise.de/newsticker/meldung/36136

Schöning, U.: Algorithmen – kurz gefasst. Heidelberg; Berlin: Spektrum Akademischer Verlag, 1997.

Singh, S.: Geheime Botschaften – Die Kunst der Verschlüsselung von der Antike bis in die Zeiten des Internet. München; Wien: Hanser, 2000.

Wiesenbauer, J.: Public Key Kryptosysteme in Theorie und Programmierung. In: Schriftenreihe zur Didaktik der Österreichischen Mathematischen Gesellschaft, Heft 30 (1999), S. 144–159.

Wikipedia (deutsch): Stichwort: „Adrien-Marie Legendre“.
http://de.wikipedia.org/wiki/Adrien-Marie_Legendre

Wikipedia (deutsch): Stichwort „Bartgeier“.
http://de.wikipedia.org/wiki/Bartgeier

Wikipedia (deutsch): Stichwort „Euklidischer Algorithmus“.
http://de.wikipedia.org/wiki/Euklidischer_Algorithmus

Wikipedia (englisch): Stichwort „RSA Factoring Challenge“.
http://en.wikipedia.org/wiki/RSA_Factoring_Challenge

Wikipedia (englisch): Stichwort „Lammergeier“
http://en.wikipedia.org/wiki/Lammergeier

Wikipedia (englisch): Stichwort „RSA-129“.
http://en.wikipedia.org/wiki/RSA-129

Wikipedia (englisch): Stichwort „The Magic Words are Squeamish Ossifrage“.
http://en.wikipedia.org/wiki/The_Magic_Words_are_Squeamish_Ossifrage

Wikipédia (französisch): Stichwort „Identité de Bézout“.
http://fr.wikipedia.org/wiki/Identit%C3%A9_de_B%C3%A9zout

Witten, H.; Letzner, I.; Schulz, R.-H.: RSA & Co. in der Schule (Teil 2) – Von Cäsar über Vigenère zu Friedman. In: LOG IN, 18 Jg. (1998), H. 5, S. 31–39.

Witten, H.; Schulz, R.-H.: RSA & Co. in der Schule (Neue Folge – Teil 1: RSA für Einsteiger). In: LOG IN, 26. Jg. (2006), H. 140, S. 45–54.

Alle URLs der Internetquellen wurden am 20. Dezember 2006 geprüft.