Kryptographie

Alfred Nussbaumer, LSR für NÖ

11. - 15. April 2011, Talentezentrum Schloss Drosendorf

Kryptographische Verfahren werden seit langer Zeit zur Geheimhaltung von Daten eingesetzt. In welcher Weise sie für Vertraulichkeit, Integrität und Authentizität verwendet werden können, ist Inhalt dieses Skriptums. Dabei werden die Verfahren vorgestellt und ihre Arbeitsweise mit Hilfe des Computer-Algebra-Systems MuPAD untersucht, welches die zum Teil aufwändigen Funktionen und die erforderliche Modulo-Arithmetik zur Verfügung stellt.

Das Skriptum gliedert sich in verschiedene Abschnitte: Nach einer allgemeinen Einleitung in die Aufgaben und ***wichtige kryptographische Verfahren*** *dienen die Themen aus der* ***Zahlentheorie und Wahrscheinlichkeitsrechnung*** *dazu, sich mit einigen mathematischen Grundlagen für kryptographische Verfahren vertraut zu machen. Der* ***praktische Einsatz der Kryptographie*** *bezieht sich auf Hashverfahren (Passwörter, md5-Summen), sichere Server-Client-Verbindungen (ssh, vpn) und das Verschlüsseln von Daten*[1]*.*

Contents

[1] Die Verschlüsselung von E-Mails und das Signieren von Nachrichten wird - wegen der geringen Verbreitung - bei diesem Intensivkurs nicht berücksichtigt.

Dieses Skriptum[2] dient lediglich als Arbeitsbehelf zum Intensivkurs “Kryptographie” im Rahmen der Begabungs- und Begabtenförderung im Talentezentrum Schloss Drosendorf. Der Autor übernimmt keine Gewährleistung über die zitierten Verfahren, Methoden und Beispiele.

[2] Erstellt mit LaTeX.

1 Aufgaben der Kryptographie

Durch Kryptographie soll der Datenaustausch - etwa via Internet - effektiv geschützt werden. Dies betrifft beispielsweise:

- E-Mail
- Internet-Banking, Bestellwesen
- Befehle an einen entfernten Computer bertragen
- VPN - sichere Netzwerkverbindung über öffentliche Kanäle
- Benutzerauthentifizierung mittels verschlüsselter Passwörter
- Verschlüsseln von Datenträgern / Verzeichnissen / Dateien
- Bankomat-, Kreditkarten

Mit so genannten Schlüsseln wird das Verschlüsseln, Entschlüsseln und Signieren von Nachrichten möglich. Die Schlüssel sind entweder als - wiederum verschlüsselte - Datei auf der Festplatte, auf Chipkarten oder USB-Geräte gespeichert. In allen Fällen sollen Kryptosysteme sicherstellen:

- Vertraulichkeit - die Daten sollen nicht im Klartext übertragen werden
- Integrität - die Daten sollen vollständig und unveränderbar übertragen werden
- Authentizität - die Identität des Absenders soll exakt feststellbar sein

Die Kryptographie ist an sich eine jahrtausendealte ““Wissenschaft” - aber erst seit dem Ende des 2. Weltkrieges im vorigen Jahrhundert wurde die Kryptographie Ziel zahlreicher Forschungen. Neuartige Entwicklungen auf dem Gebiet der Quantenkryptographie lassen noch sicherere Übertragungsmethoden erwarten[3].

2 Grundbegriffe zur Kryptographie

Im Laufe der Zeit sind zahlreiche Verschlüsselungsverfahren entwickelt worden. Einige “wichtige” sollen hier besprochen werden. Allen gemeinsam sind folgende Definitionen:

Klartext - die unverschlüsselte Nachricht.

Schlüsseltext - die verschlüsselte Nachricht.

Verschlüsselung - der Vorgang des Umsetzens vom Klartext in den Schlüsseltext

Entschlüsselung - der umgekehrte Vorgang ;-)

Schlüssel - (oft geheimer) Parameter für einen Ver- oder Entschlüsselungsalgorithmus

Klartextalphabet - das Alphabet für den Klartext

Geheimtextalphabet - das Alphabet für den Geheimtext

[3] In Österreich befasst sich zur Zeit eine Forschergruppe um A. Zeilinger in Wien (*http://www.quantum.at*) experimentell mit Quantenkryptographie.

2.1 Klassische Verschlüsselungsverfahren

2.1.1 Verschiebechiffre

Eines der einfachsten Verfahren überhaupt besteht darin, die Buchstaben des Klartextes alphabetisch um eine bestimmte Anzahl von Buchstaben zu verschieben. Alte Quellen (Sueton) berichten davon, dass der römische Kriegsherr Caesar einen solchen Verchiebechiffre zum Übermitteln geheimer Nachrichten verwendet hat: Dabei wurde jeder Klartext mit dem Schlüssel 3 verschlüsselt: Jeder Buchstabe des Klartextes wurde um 3 Stellen im Alphabet nach hinten verschoben (aus “A” wird “D”, aus “B” “E” usw.), die letzten Buchstaben im Alphabet wurden zyklisch durch die ersten Buchstaben im Alphabet ersetzt.

Schwachstellen: Probiert man alle Schlüssel durch - im lateinischen Alphabet mit 20 Buchstaben brauchte man also höchstens 19, im Schnitt jedoch nur 10 Schlüssel auszuprobieren - findet man sehr schnell den richtigen Schlüssel und erhält den Klartext. Allerdings muss man zuerst noch erkennen, dass eine Verschiebechiffre verwendet wurde ;-)

Bemerkung: Die beschriebene Chiffre wird in der Literatur auch als “Caesarchiffre” bezeichnet. Später wurden so genannte **Chiffrierscheiben** verwendet: Auf zwei Scheiben steht das zyklisch angeordnete Alphabet, die kleinere Scheibe ist drehbar. Jede Stellung der kleineren (“inneren”) Scheibe ergibt einen anderen Schlüssel: Beim Verschlüsseln liest man die Buchstaben von außen nach innen, beim Entschlüsseln von innen nach außen.

2.1.2 Substitutionschiffre

Der Nachteil der geringen Anzahl verschiedener Schlüssel beim Caesar-Verfahren kann durch ein besseres Geheimtext-Alphabet verkleinert werden. Dabei wählt man nicht ein verschobenes Alphabet, sondern ein Alphabet, dessen Buchstaben beliebig vertauscht (permutiert) wurden. Jeder Buchstabe des Klartextalphabets wird durch seinen entsprechenden Vertreter im Geheimtextalphabet ersetzt (daraus erklären sich die Namen “Permutationsverfahren”, “Substitutionsverfahren”).

Beim Substitutionsverfahren wird also jedm Buchstaben im Klartextalphabet ein Buchstabe des Geheimtextalphabets eindeutig zugeordnet. Das Geheimtextalphabet ist jedoch weitgehend nicht alphabetisch sortiert.

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
Z	I	N	F	O	R	M	A	T	K	S	C	H	U	D	L	E	G	B	J	P	Q	V	W	X	Y

Im Beispiel wird das Schlüsselwort “INFORMATIK IST SCHON UND LEHRER LUGEN NICHT” verwendet: Die Buchstaben dieses Satzes werden der Reihe nach im Geheimtextalphabet angeordnet; wiederholte Buchstaben werden ausgelassen. Zuletzt kommen noch alle nicht verwendeten Buchstaben des Klartextalphabetes. Um an den letzten Stellen Klartextbuchstaben nicht unverändert zu lassen, wird vor das Schlüsselwort der Buchstabe “Z” gesetzt...

Für 26 Buchstaben (Großbuchstaben des deutschen Alphabets) gibt es 26! $\approx 10^{26}$ viele Permutationen. Ein einfaches Durchprobieren aller Schlüssel fhrt beim Knacken der Chiffre kaum zum Erfolg; dennoch führen einfache Methoden der Kryptoanalyse leicht zum Knacken der Substitutionschiffre.

2.1.3 Affine Chiffre

Es ist üblich, die Buchstaben der Alphabete an Hand ihres Index zu bezeichnen, beispielsweise für das Klartextalphabet A = 1, B = 2, C = 3, usf. Verschlüsselungsverfahren werden dann auf

diese Indizes angewendet (z.Bsp. Addition beim Caesar-Verfahren). Die Affine Chiffre verwendet Restklassen-Addition und -Multiplikation (und fußt somit auf der Ringstruktur von $\mathbf{Z}_m$).

Sei $m = 26$ und $\mathrm{ggT}(a, m) = 1$. Dann lässt sich das Verschlüsselungsverfahren mit dem Schlüssel (a, b) wie folgt definieren:

$X \to a \cdot X + b$

Das Entschlüsseln läuft invers:

$Y \to a^{-1}(Y - b)$

2.1.4 Vigenere Chiffre

Bei der Caesar- und Substitutions-Chiffre wurde jeweils ein Buchstabe des Klartextalphabets durch einen bestimmten Buchstaben des Geheimtextalphabets ersetzt. Solche Chiffren heißn **monoalphabetisch**. Um die Analyse der Klartext-Buchstabenhäufigkeiten im Geheimtext zu erschweren wurden ab 1500 **polyalphabetische** Chiffren entwickelt. Die Grundidee besteht darin, mehrere Geheimtextalphabete zu verwenden.

B. de Vigenére (1523 - 1585) entwickelte ein Verfahren, das auf einem Schlüsselwort beruht. Die Buchstaben des Schlüsselwortes geben das jeweilige Geheimtextalphabet vor.

2.1.5 Vernam Chiffre

1918 entwickelten Gilbert Vernam und Joseph O. Mauborgne eine unknackbare Chiffre: Als Schlüssel wird eine Zufallsfolge von Zeichen verwendet, die genauso lang wie der Klartext ist. Zunächst wird jeder Buchstabe des Alphabets durch eine (eindeutige) Zahl repräsentiert. Jedes Zeichen des Klartextes wird nun mit dem entsprechenden Zeichen des Schlüssels verknüpft, indem man die zugehörigen Zahlen (üblicherweise ohne Zehnerübertrag) miteinander addiert.

Verwendet man jeden Schlüssel nur einmal, so ist das Verfahren nachweisbar sicher. Allerdings muss für jede neue Nachricht ein neuer Schlüssel verwendet werden; die notwendige, sichere Übertragung des Schlüssels gestaltet sich aber kompliziert. Außerdem stellt sich die Frage, ob man statt der geheimnisvollen Übermittlung des Schlüssels nicht auch gleich die Nachricht weitergeben hätte können ...

2.1.6 Aufgaben

1. Realisiere eine Caesar-Chiffre-Scheibe: Entlang der Kreislinie einer Scheibe sind alle Buchstaben des Alphabets (und Ziffern, Sonderzeichen, etc.) alphabetisch angeordnet. Konzentrisch drehbar dazu ist eine etwas kleinere Kreisscheibe mit genau den gleichen Zeichen in der gleichen Reihenfolge montiert. Drehe die kleinere Scheibe um 13 Stellen weiter ... Wie funktioniert diese Verschlüsselung?

2. Verwende statt eines zweiten Alphabets eine “beliebige” Zeichenfolge auf der inneren Scheibe; beachte aber, dass **alle** Zeichen auftreten müssen. Wie wird mit dieser Scheibe verschlüsselt und entschlüsselt? Ist diese Scheibe grundsätzlich sicherer als die Caesar-Chiffre?

3. Recherchiere die historische Ausführung von Chiffrescheiben!

2.1.7 Symmetrische Verschlüsselungsverfahren

Alle bis jetzt besprochenen Chiffrierverfahren verwenden zum Verschlüsseln und Entschlüsseln einer Nachricht jeweils den gleichen Schlüssel. Solche Verfahren heißen **symmetrisch**. Wir besprechen hier weitere symmetrische Verschlüsselungsverfahren.

2.1.8 Hill Chiffre

Bei der Hill Chiffre wird eine $n \times n$ - Matrix A als Schlüssel verwendet. Dabei müssen die Determinante der Matrix A und der Modul m teilerfremd sein: $\mathrm{ggT}(\det A, m) = 1$ (vgl. Abschnitt.

Zur Verschlüsselung wird der Klartext in Blöcke der Länge n eingeteilt: Bei der Multiplikation eines Blockes mit der Matrix A entsteht ein gleichlanger Geheimtextblock.

2.1.9 Permutationschiffre

Die Permutationschiffre stellt einen Spezialfall der Hill Chiffre dar. Der Klartext wird in gleich lange Blöcke unterteilt. Die Buchstaben jedes Blocks werden nun nach der vorgebenene Permutationsvorschrift permutiert.

2.1.10 Stromchiffren

Die zuletzt besprochenen Chiffren sind alle so genannten **Blockchiffren**. Für jeden Block des Klartextes wird der gleiche Schlüssel angewendet. Der Klartext wird mit einer ganzen Reihe verschiedener Schlüssel verschlüsselt. Man kann zeigen, dass beispielsweise die Vigenére Chiffre eine periodische Stromchiffre mit der Periode n ist.

2.1.11 Autokey Chiffre

Bei der Autokey-Chiffre wird der Klartext selbst als Schlüssel verwendet. Im folgenden Beispiel wurden die (Großbuchstaben des Alphabets einfach durchnummeriert. Damit erhalten wir den Modul 26 für die folgenden Berechnungen.
Beispiel: $m = 26$, $K = 17$

Verschlüsseln: Wir addieren jeweils den vorhergehenden Buchstabencode.

I	N	F	O	R	M	A	T	I	K	I	S	T	S	C	H	O	E	N
9	14	6	15	18	13	1	20	9	11	9	19	20	19	3	8	15	5	14
17	9	14	6	15	18	13	1	20	9	11	9	19	20	19	3	8	15	5
0	23	20	21	7	5	14	21	3	20	20	2	13	13	22	11	23	20	19
Z	W	T	U	G	E	N	U	C	T	T	B	M	M	V	K	W	T	S

Entschlüsseln: Wir subtrahieren jeweils den vorhergehenden Buchstabencode.

Z	W	T	U	G	E	N	U	C	T	T	B	M	M	V	K	W	T	S
0	23	20	21	7	5	14	21	3	20	20	2	13	13	22	11	23	20	19
17	9	14	6	15	18	13	1	20	9	11	9	19	20	19	3	8	15	5
9	14	6	15	18	13	1	20	9	11	9	19	20	19	3	8	15	5	14
I	N	F	O	R	M	A	T	I	K	I	S	T	S	C	H	O	E	N

Beachte, dass eine Fehlentschlüsselung ausreicht, dass die Entschlüsselung des restlichen Geheimtextes misslingt.

2.1.12 DES

Der DES (*Data Encryption Standard*, vgl. Abschnitt 6) ist ebenfalls ein symmetrischer Algorithmus. Er wurde beispielsweise durch den Triple DES verbessert. In letzter Zeit wurde er durch den AES (*Advanced Encryption Standard*) abgelöst.

2.1.13 Aufgaben

1. Lies in Büchern, Zeitschriften oder “im Internet” zu symmetrischen Verschlüsselungsverfahren nach!

2. Beschreibe ein weitere symmetrische Verschlüesselungsverfahren mit eigenen Worten!

3. Recherchiere zu historischen Verschlüsselungsverfahren und deren Angriffe (zB. Enigma, M 209, TYPEX MK III, purple machine)!

4. Lies zu historischen Kryptographen nach (zB Cardano (1501 - 1576), G.B. Della Porta (1538 - 1615), J. Trithemius (1462 - 1516), B. de Vigenère (1523-1585)

5. Durch systematisches Durchprobieren aller Schlüssel kann der DES grundsätzlich geknackt werden. Diese prinzipielle Schwäche wurde durch den Triple-DES, durch den AES (*Advanced Encryption Standard*) und IDEA (*International Encryption Standard*) korrigiert und weiter entwickelt. Lies zu diesen Verfahren nach! Was hat dies mit dem Algorithmus von Rijndael zu tun?

6. Geldautomaten arbeiten mit dem DES-Algorithmus. Auf der Bankomatkarte ist eine eindeutige PIN (*Personal Identification Number*) codiert ... Beschreibe die mögliche Aufgabe der Bankomatkarte!

2.2 Asymmetrische Verschlüsselungsverfahren - Public-Key

Symmetrische Verschlüsselungsverfahren haben einige Nachteile: Erstens sind die Verschlüsselungs- und die Entschlüsselungsfunktion häufig sehr ähnlich, und zweitens muss der (geheime!) Schlüssel auf sicherem Weg zwischen Sender und Empfänger ausgetauscht werden. Weiters muss dieser geheime Schlüssel mit allen Kommunikationspartnern ausgetauscht werden ...

1976 entwickelten Whitfield Diffie & Martin Hellmann (und davon unabhängig Ralph Merkle) die Idee eines Public-Key-Kryptosystems: Beim Entschlüsseln einer Nachricht sollte ein anderer Schlüssel als zum Verschlüsseln verwendet werden. Weiters soll es sehr schwierig sein, aus der Kenntnis des einen Schlüssels den anderen berechnen zu können. Von diesen beiden Schlüsseln wird einer öffentlich bekannt gemacht (= öffentlicher Schlüssel), der andere Schlüssel bleibt (streng) geheim. Zu jedem privaten Schlüssel gehört genau ein öffentlicher Schlüssel (und umgekehrt; wir sprechen auch von einem “Schlüsselpaar”).

Will Alice nun eine vertrauliche Nachricht an Bob senden, so verschlüsselt sie ihre Nachricht mit dem öffentlichen Schlüssel von Bob. Dieser kann sie mit dem nur ihm bekannten privaten Schlüssel entschlüsseln. Alice sendet mit ihrem Mail-System verschlüsselte Nachrichten an alle Personen, deren öffentliche Schlüssel sie ihrem “Schlüsselbund” hinzugefügt hat...

Auch digitale Signaturen können mit Hilfe von asymmetrischen Schlüsseln realisiert werden. Möchte Alice ihre Nachricht signieren, so berechnet sie einen Hashwert aus ihrer Nachricht und verschlüsselt diesen Hashwert mit ihrem privaten Schlüssel. Das Ergebnis sendet Alice gemeinsam mit ihrer (unverschlüsselten) Nachricht an Bob. Bob entschlüsselt den Hashwert an Hand des öffentlichen Schlüssels von Alice, berechnet den Hashwert aus der Nachricht, die er von Alice empfangen hat und vergleicht. Stimmen beide Werte berein ist erstens sicher gestellt, dass die Nachricht unverändert übertragen wurde, und dass sie mit Sicherheit von Alice stammt.

2.2.1 Falltrap-Funktionen

Als Falltrap-Funktionen (*one-way-Funktionen*, Einweg-Funktionen, ...) bezeichnen wir Funktionen, die zwar sehr leicht einen Funktionswert aber nur sehr schwer die Umkehrung berechnen lassen. So kann beispielsweise sehr leicht das Produkt zweier (großer) Primzahlen berechnet werden. Die Primfaktorzerlegung des Ergebnisses ist im Allgemeinen jedoch schwierig.

2.2.2 Merlke-Hellmann-Knapsack

2.2.3 RSA, Rivest-Shamir-Adleman

Drei Professoren vom MIT, Ron Rivest, Adi Shamir und Leonard Adleman, veröffentlichten 1977 das erste Public-Key-Kryptosystem. Das Verfahren verwendet große Primzahlen.

Schlüsselerzeugung:
Zuerst braucht man zwei große Primzahlen p und q, die typischerweise ca. 1024 bis 2048 Bit groß sind, und berechnet das Produkt n der beiden Zahlen.

$$n = p * q$$

Anschließend berechnet man die Anzahl der teilerfremden Zahlen zu n mit der eulerschen Φ-Funktion.

$$\Phi_{(n)} = (p-1)(q-1)$$

Nun muss man zwei Zahlen e und d suchen, die folgende Bedingung erfüllen:

$$e * d \equiv 1 \ (\, mod\, \Phi_{(n)}\,)$$

In der Praxis wählt man eher zuerst eine Primzahl e und sucht dazu ein geeignetes d.

Das Zahlenpaar (e,n) bildet den "offentlichen Schlüssel, das Zahlenpaar (d,n) den geheimen Schlüssel.

Verschlüsselung:
Wenn man eine geheime Botschaft m übertragen will, muss man sich zuerst den öffentlichen Schlüssel seines Kommunikationspartners holen. Die Botschaft m muss dabei kleiner als n sein. Üblicherweise nimmt man eine Zahl die um eine Dezimalstelle kürzer ist.

Die verschlüsselte Botschaft c wird wie folgt berechnet:

$$c = m^e \, mod \, n$$

Dann kann man die Zahl c an den Empfänger schicken. Da nur er das geheime Schlüsselpaar (d,n) kennt kann auch nur er die geheime Botschaft m aus c berechnen.

$$m' = c^d \, mod \, n$$

Man kann zeigen das $m = m'$ ist. Der Empfänger hat damit aus der verschlüsselten Botschaft c, die geheime Botschaft m berechnet, das heißt er hat c entschlüsselt.

Die Sicherheit von RSA
Zur Sicherheit von RSA tragen vor allem zwei mathematische Probleme bei. Gelänge es eines dieser beiden in einem vernünftigen Zeitraum zu lösen wäre die Sicherheit nicht mehr gegeben.

Erstens ist es sehr schwer die sehr große Primzahl n zu zerlegen und so p und q zu bekommen. Kennt ein Angreifer p und q kann er leicht $\Phi_{(n)}$ berechnen und so ist es sehr leicht für ihn vom öffentlichen Schlüssel (e,n) auf den geheimen Schlüssel (d,n) zu schließen.

1977 hat Martin Gardner in seiner Kolumne "Mathematical Recreations" im "Scientific American" seinen Lesern aufgetragen die beiden Faktoren, die Primzahlen sind, der 129-stelligen Zahl 114 381 625 757 888 867 669 235 779 976 146 612 010 218 296 721 242 362 562 561 842 935 245 733 897 830 597 123 563 958 705 058 989 075 147 599 290 026 879 543 541 herauszufinden. Er musste bis in den April 1994, also 16 Jahre, warten bis ihm Paul Leyland von der Universität Oxford, Michael Graff von der Universitt von Iowa und Derek Atkins vom MIT (*Massachusetts Institute of Technology*) in Cambridge die Faktoren präsentierten. Das Programm

dazu stammte von Arjen K. Lenstra vom Zentrum Bell Communications Reserach in Norristown und wurde von ca. 600 Freiwilligen, die ihre Workstations nächtelang rechnen ließen, ausgeführt.

Die Firma RSA veröffentlicht heute noch Produkte zweier Primzahlen und lobt ein Preisgeld für deren Faktorisierung aus. Informationen zu den aktuellen “Challenge Nummbers” und dem ausgesetztem Preisgeld finden sich auf *http://www.rsasecurity.com/rsalabs/challenges/factoring/numbers.htm* die derzeit am kleinse nicht faktorisierte Primzahlen hat 174 Stellen und das ausgesetze Preisgeld für deren Faktorisierung beträgt 10000 Dollar.

Am 4. Feburar 2002 gaben Mathematiker der Universitt Bonn bekannt, mit dem Linux-Cluster Parnass2 (der aus 144 400-MHz PentiumII Prozessoren bestand), einen neuen Weltrekord in der Faktorisierung grosser Zahlen in Primzahlen aufgestellt zu haben. Die Zahl, die von den Mathemaktikern der Universität Bonn zerlegt wurde, hatte 158 Dezimalstellen.

Zweitens sichert hier das Problem des diskreten Algorithmus das RSA-Verfahren. Man könnte dann einfach irgendein m whlen und mit dem öffentlichen Schlüssel c berechnen und dann mithilfe des diskreten Algorithmus auf d schliessen, das heißt: Suche einen Wert d für den $m = c^d$ mod n richtig ist.

2.2.4 Aufgaben

1. Lies zu folgenden Personen nach: W. Diffie, M. Hellman, R. Rivest, A. Shamir, L. Adleman
2. Chiffrierverfahren werden zB folgendermaßen eingeteilt:

   Direkte Chiffrierverfahren - Substitutionsverfahren (Alphabetverfahren, Blockverfahren, Code(C)), Permutationsverfahren (Umstellung), Stromverfahren (i-Wurm, Buchchiffre, Zahlenwurm)

   Indirekte Chiffrierverfahren - Chaffing, Frequency Hopping, Versteckende Verfahren (Steganographie)
3. **Handverfahren** spielten in der Geschichte eine wichtige Rolle. Fasse mögliche Handverfahren zusammen!

2.2.5 Rechenbeispiel zum Verschlüsseln, erweiterter Euklidscher Algorithmus

Im Rechenbeispiel verwenden wir deutlich kleinere Zahlen ;-)

1. Wir verschlüsseln beispielsweise die streng geheime Nachricht “A”. Dieser Zahl ordnen wir beispielsweise den ASCII-Code $65 = m$ zu.
2. Fr das Verschlüsselungsverfahren wählen wir die Primzahlen $p = 13$ und $q = 17$. Auf bestimmte Weise - dies wir anschließend dargestellt - berechnen wir daraus den öffentlichen Schlüssel (7,221) und den geheimen Schlüssel (55,221).
3. Wir verschlüsseln die Nachricht “A” mit dem öffentlichen Schlüssel gemäß der oben dargestellten Formel:

   $$c = m^e \, mod \, n$$

   $$c = 65^7 \, mod \, 221 = 91$$
4. Die verschlüsselte Nachricht wird nun an den Empfänger gesendet, der die Nachricht mit dem nur ihm bekannten geheimen Schlüssel entziffern kann:

   $$m' = c^d \, mod \, n$$

   $$m' = 91^{55} \, mod \, 221 = 65$$

Erklärung:

Wir müssen nun den Zusammenhang zwischen den gewählten Primzahlen p, q und dem Schlüsselpaar (e,n) und (d,n) klären.

Zunächst gehen wir von der **Euler'schen Φ-Funktion** aus. Sie liefert die Anzahl aller zu einer Zahl teilerfremden Zahlen, die kleiner als die Zahl selbst sind. Da Primzahlen p keine echten Teiler besitzen, gilt sofort:

$$\Phi_{(p)} = (p-1)$$

Fr das Produkt $n = p \cdot q$ der beiden Primzahlen p und q gilt dann:

$$\Phi_{(n)} = (p-1)(q-1)$$

Nun suchen wir zwei Zahlen e und d mit der Eigenschaft

$$e \cdot d \equiv 1 \ (mod\, \Phi_{(n)})$$

Wegen des **Kleinen Satzes von Fermat** gilt:

$$e^{\Phi(n)} \equiv 1 \ (mod\, n)$$

Dies verwenden wir für das Bestimmen der Zahl d, die zu e modular invers sein soll ("modulare Inverse"):

Wählen wir für die Primzahlen $p = 13$ und $q = 17$, $n = p \cdot q$, $r = \Phi(n) = (p-1)(q-1) = 12 \cdot 16 = 192$, so müssen wir schließlich nur mehr die modulo inverse Zahl d zu e suchen. Zunächst wählen wir beispielsweise $e = 7$ und legen damit den öffentlichen Schlüssel (e,n) = (7, 221) fest. Dann bestimmen wir mit dem **Erweiterten Euklid'schen Algorithmus** d mit der Eigenschaft $e * d \equiv 1 \ (mod\, r)$:

192	7		
1	0	192	q
0	1	7	
1	-7	3	27
-2	55	1	2

In jeder Zeile kann der Wert in der dritten Spalte als Linearkombination der beiden Zahlen 192 und 7 berechnet werden; die dazu nötigen Linearfaktoren stehen in der ersten und zweiten Spalte[4]. Auf diese Weise lesen wir in der letzten Zeile:

$$-2 \cdot 192 + 55 \cdot 7 = 1$$

Insbesondere gilt dann

$$-2 \cdot 192 + 55 \cdot 7 \equiv 55 \cdot 7 = 1 \,(mod\, 192)$$

Somit ist 55 die Inverse zu $7 (mod 192)$. Wir erhalten damit den (geheimen) privaten Schlüssel (e, n) = (55, 221). Die Primzahlen $p = 13$ und $q = 17$ werden für alle weiteren Berechnungen und Umformungen nicht mehr benötigt und sollten (aus Sicherheitsgründen) jetzt gelöscht werden.

Wir zeigen nun, dass $c^d \equiv m \,(mod\, n)$ gilt:

$$c \equiv m^e \,(mod\, n)$$

Sei

$$m' = c^d \equiv (m^e)^d = m^{e \cdot d} \,(mod\, n)$$

[4] Dieser Eigenschaft liegt der folgende Satz zu Grunde: $ggT(a,b) = d \Rightarrow d = x \cdot a + y \cdot b$ für $x, y \epsilon \mathbf{Z}$. Insbesonders gilt dieser Satz, wenn die beiden Zahlen a und b teilerfremd sind.

Da e und d invers modulo $\Phi(r)$ sind ($r = (p-1) \cdot (q-1)$), erhalten wir:

$$e \cdot d = 1 + k \cdot (p-1) \cdot (q-1)$$

Daher berechnen wir:

$$m^{e \cdot d} = m^{1+k \cdot (p-1) \cdot (q-1)} = m \cdot (m^{p-1})^{k \cdot (q-1)} \equiv m \cdot 1^{k \cdot (q-1)} = m \, (mod \, p)$$

Die erwendete Kongruenz ergibt sich aus dem Kleinen Satz von Fermat. Analog erhalten wir für q:

$$m^{e \cdot d} = m^{1+k \cdot (p-1) \cdot (q-1)} = m \cdot (m^{q-1})^{k \cdot (p-1)} \equiv m \cdot 1^{k \cdot (p-1)} = m \, (mod \, q)$$

Da $p \mid (m^{ed} - m)$ und $q \mid (m^{ed} - m)$ gilt auch $p \cdot q \mid (m^{ed} - m)$ ($p \cdot q = n$). Damit haben wir gezeigt, dass

$$m^{ed} - m \equiv 0 \, (mod \, n)$$

und somit

$$m' = m$$

(da $m < n$ gewählt wurde).

2.2.6 Rechenbeispiel zum Signieren einer Nachricht

Der RSA-Algorithmus ist relativ langsam. Er eignet sich daher in erster Linie, um Nachrichten zu signieren. Im folgenden Beispiel wählen wir eine sehr kurze Nachricht ;-)

Wir signieren beispielsweise die Nachricht “JA” und stellen die beiden Buchstaben zunächst anhand ihrer ASCII-Codes dar: 6574. Aus diesen Zahlen bilden wir einen Hashwert (vgl. das folgende Kapitel). Zu Demonstrationszwecken addieren wir einfach alle Ziffern: $6+5+7+4 = 22$. Diesen Wert verschlüsselt der Sender mit dem geheimen Schlüssel (d,n) = (55,221) von vorigem Beispiel:

$$22^{55} \equiv 61 \, (mod \, 221)$$

Der Empfänger der Nachricht berechnet nun seinerseits den Hashwert aus der Nachricht $6 + 5 + 7 + 4 = 22$. Dann kann er den verschlüsselten Hashwert mit dem öffentlichen Schlüssel (d,n) = (7,221) entschlüsseln:

$$61^7 \equiv 22 \, (mod \, 221)$$

Stimmen beide Werte berein, dann wurde die Nachricht erstens vollständig und unverändert bertragen und außrdem stammt sie vom Sender, dessen öffentlicher Schlüssel zum Entschlüsseln des Hashwertes verwendet wurde.

2.3 Hash-Verfahren

Hash-Funktionen sind im Wesentlichen Einweg-Funktionen, die sich sehr leicht aus dem Klartext berechnen lassen. Um den Klartext eindeutig zu identifizieren müssen sie im Allgemeinen folgende Eigenschaften aufweisen:

1. Hash-Funktionen liefern für beliebig lange Klartexte ein Ergebnis von vorgegebener Länge.
2. Hash-Funktionen müssen von allen Teilnehmern leicht zu berechnen sein.
3. Hash-Funktionen müssen praktisch eindeutig sein, d.h. es muss praktisch ausgeschlossen sein, dass zwei verschiedene Klartexte den gleichen Wert der Hash-Funktionen liefern. Im Besonderen muss es ausgeschlossen sein, dass jemand zu einem vorgegebenen Hashwert einen Klartext zusammenstellen kann, der genau den gleichen Hashwert liefert.

Hash-Funktionen werden beispielsweise zum Test verschlüsselter Passwörter oder zum Test verwendet, ob eine Datenübertragung fehlerfrei war. In solchen Fällen wird eine so genannte **Prüfsumme** (*Message Digest*, MD) berechnet

2.3.1 MD5

MD5 wurde vom amerikanischen Kryptographen Ron Rivest entwickelt. Er dient zum Erstellen von HashWerten beliebig langer Nachrichten. Das Ergebnis ist ein “Fingerprint” aus 16 Bytes (128 Bits, 32 Hexadezimalziffern).

Bei der Berechnung wird zunächst die Nachricht mit Bits so verlängert, dass ihre Länge kongruent zu 448 (modulo 512) ist. Dazu wird das Bit “1” und dann entsprechend viele “0” angehängt. Zuletzt wird eine 64-bit Darstellung der Länge der Nachricht angehängt (damit erhöht die Nachricht exakt eine Länge von einem Vielfachen von 512 Bits, das sind 16 32-bit Wörter). Daher kann die Nachricht nun in der Form M[0 ... N-1] geschrieben werden, wobei N ein Vielfaches von 16 ist.

Zur Berchnung werden 4 32-Bit-Register verwendet, die vier Wörter (A, B, C, D) aufnehmen können. Sie werden folgendermaßen initialisiert:

A	01	23	45	67
B	89	ab	cd	ef
C	fe	dc	ba	98
D	76	54	32	10

Mit folgenden vier Funktionen werden jeweils 3 32-Bit-Wörter in ein 32-Bit-Wort umgewandelt:

```
F(X,Y,Z) = XY v not(X) Z
G(X,Y,Z) = XZ v Y not(Z)
H(X,Y,Z) = X xor Y xor Z
I(X,Y,Z) = Y xor (X v not(Z))
```

Zur folgenden Berechnung wird ein Datenarray T mit 64 Elementen verwendet: T[i] entspricht dem ganzzahligen Anteil des Ergebnisses von 4294967296 mal dem Absolutbetrag von sin(i), wobei i in Radiant angegeben wird.

Die folgende Berechnung ist dem Dokument rfc1321.txt [5] entnommen:

```
/* Process each 16-word block. */
For i = 0 to N/16-1 do

  /* Copy block i into X. */
  For j = 0 to 15 do
    Set X[j] to M[i*16+j].
  end /* of loop on j */

  /* Save A as AA, B as BB, C as CC, and D as DD. */
  AA = A
  BB = B
  CC = C
  DD = D

  /* Round 1. */
  /* Let [abcd k s i] denote the operation
       a = b + ((a + F(b,c,d) + X[k] + T[i]) <<< s). */
  /* Do the following 16 operations. */
  [ABCD  0  7  1]  [DABC  1 12  2]  [CDAB  2 17  3]  [BCDA  3 22  4]
  [ABCD  4  7  5]  [DABC  5 12  6]  [CDAB  6 17  7]  [BCDA  7 22  8]
  [ABCD  8  7  9]  [DABC  9 12 10]  [CDAB 10 17 11]  [BCDA 11 22 12]
  [ABCD 12  7 13]  [DABC 13 12 14]  [CDAB 14 17 15]  [BCDA 15 22 16]
```

[5] http://www.faqs.org

```
    /* Round 2. */
    /* Let [abcd k s i] denote the operation
         a = b + ((a + G(b,c,d) + X[k] + T[i]) <<< s). */
    /* Do the following 16 operations. */
    [ABCD  1  5 17]  [DABC  6  9 18]  [CDAB 11 14 19]  [BCDA  0 20 20]
    [ABCD  5  5 21]  [DABC 10  9 22]  [CDAB 15 14 23]  [BCDA  4 20 24]
    [ABCD  9  5 25]  [DABC 14  9 26]  [CDAB  3 14 27]  [BCDA  8 20 28]
    [ABCD 13  5 29]  [DABC  2  9 30]  [CDAB  7 14 31]  [BCDA 12 20 32]

    /* Round 3. */
    /* Let [abcd k s t] denote the operation
         a = b + ((a + H(b,c,d) + X[k] + T[i]) <<< s). */
    /* Do the following 16 operations. */
    [ABCD  5  4 33]  [DABC  8 11 34]  [CDAB 11 16 35]  [BCDA 14 23 36]
    [ABCD  1  4 37]  [DABC  4 11 38]  [CDAB  7 16 39]  [BCDA 10 23 40]
    [ABCD 13  4 41]  [DABC  0 11 42]  [CDAB  3 16 43]  [BCDA  6 23 44]
    [ABCD  9  4 45]  [DABC 12 11 46]  [CDAB 15 16 47]  [BCDA  2 23 48]

  /* Round 4. */
    /* Let [abcd k s t] denote the operation
         a = b + ((a + I(b,c,d) + X[k] + T[i]) <<< s). */
    /* Do the following 16 operations. */
    [ABCD  0  6 49]  [DABC  7 10 50]  [CDAB 14 15 51]  [BCDA  5 21 52]
    [ABCD 12  6 53]  [DABC  3 10 54]  [CDAB 10 15 55]  [BCDA  1 21 56]
    [ABCD  8  6 57]  [DABC 15 10 58]  [CDAB  6 15 59]  [BCDA 13 21 60]
    [ABCD  4  6 61]  [DABC 11 10 62]  [CDAB  2 15 63]  [BCDA  9 21 64]

    /* Then perform the following additions. (That is increment each
       of the four registers by the value it had before this block
       was started.) */
    A = A + AA
    B = B + BB
    C = C + CC
    D = D + DD

   end /* of loop on i */
```

Als Ergebnis werden die vier 32-Bit Wörter A, B, C und D ausgegeben. Zum Testen von MD5 werden folgende Werte angegeben:

```
MD5 ("") = d41d8cd98f00b204e9800998ecf8427e
MD5 ("a") = 0cc175b9c0f1b6a831c399e269772661
MD5 ("abc") = 900150983cd24fb0d6963f7d28e17f72
MD5 ("message digest") = f96b697d7cb7938d525a2f31aaf161d0
MD5 ("abcdefghijklmnopqrstuvwxyz") = c3fcd3d76192e4007dfb496cca67e13b
MD5 ("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789") =
d174ab98d277d9f5a5611c2c9f419d9f
MD5 ("12345678901234567890123456789012345678901234567890123456789012345678
78901234567890") = 57edf4a22be3c955ac49da2e2107b67a
```

2.3.2 SHA-1

Die NSA (*National Security Agency*) hat im Auftrag der amerikanischen Normierungsbehörde NIST (*National Institute of Standards and Technology*)[6] den *Secure Hash Algorithm* SHA entwickelt. SHA verarbeitet in jedem Schritt 16 32-Bit Wörter aus der Nachricht in vier Runden mit je 20 Operationen (insgesamt 80 Operationen). Durch die Verwendung einer nichtlinearen Funktion wird das Brechen des Codes zusätzlich erschwert. Das Auffüllen des letzten Nachrichtenblocks wird wie bei MD5 durchgeführt - auch hier wird die Länge der Nachricht mitcodiert.

[6] http://www.itl.nist.gov

2.3.3 RIPEMD-160

Entwickelt von europäischen Kryptographen (u.a. von Dobbertin, Bosselaers, Preneel); der Name leitet sich von “RACE” (*Research and Development in Advanced Communication Technologies*) und *RACE Integrity Privitives Evaluation Message Digest* ab. Die Hash-Werte haben eine Länge von 160 Bit (= 40 Hexadezimalziffern). Bei jedem Berechnungsschritt werden 5 Runden zu je 16 Operationen (insgesamt 80 Operationen) ausgeführt.

2.4 Steganographie

Der Wurzel des Namens “Steganographie” kommt aus dem Altgriechischen und bedeutet “verborgene Schrift”. Tatsächlich sollen Informationen in einem anderen Dokument versteckt werden[7]. So können beispielsweise Daten in die Farbdarstellung eines Bildes codiert werden: Verändert man die Farb-Bytes für die Grundfarben “R”, “G” und “B” um jeweils ein Bit, so bleibt der Gesamteindruck der Farbe unverändert. Daher zerlegt man die Nachricht zunächst in eine Folge von einzelnen Bits und addiert diese Bits zu den FarbBytes eines Bildes. Besteht ein Bild aus 800 $\times$ 600 Bildpunkten, so kann man immerhin rund 180 kB Nachrichten in den Farbwerten verstecken (Multipliziere 100 mit 600 mal 3 ;-).

Kombiniert man steganographische Verfahren mit verschlüsselten Texten, so erhält man eine gute Sicherheit: Die Nachricht wird zuerst verschlüsselt und anschließend in einem Bild versteckt.

2.5 Zero-Knowledge-Protokolle

Jemand muss beweisen, dass er einen Schlüssel k kennt, ohne dass er ihn verrät. Dies ist mit sogenannten *Zero-Knowledge-Proofs* möglich. Das Verfahren kann anhand eines kleinen Labyrinths dargestellt werden: Im Inneren des Labyrinths befindet sich ein Rundgang, der durch eine Tür versperrt ist, die nur mit dem Schlüssel k geöffnet werden kann. Der Rundgang wird von außen nur ber einen verwinkelten Gang erreicht. Will Alice nun beweisen, dass sie den Schlüssel k kennt, so geht sie in den Rundgang bis zur (verschlossenen) Tür. Bob geht nun anschließend bis zum Rundgang und nennt Alice die Seite des Rundganges, auf der sie zu ihm kommen soll. Damit Alice jedesmal den Auftrag von Bob ausführen kann, braucht sie den Schlüssel k für die Tür - nur in 50% aller Fälle steht sie zufällig auf der richtigen Seite, um auf der gewünschten Seite des Rundganges zu Bernd zurückzukehren, *ohne* die Tür aufsperren zu mssen... Die Güte des Tests hängt also nur davon ab, wie oft Bob Alice in den Rundgang schickt.

2.5.1 Aufgaben

1. Ein Chip in einer Uhr enthält ein Geheimnis (Passwort, Geheimzahl oder zufallsbestimmte Zeichenkette), mit dem sich der Besitzer legitimiert. Der Chip kann seine Information grundsätzlich nicht mitteilen (technisch unmöglich), sondern er kann spezielle Fragen über Eigenschaften des Geheimnisses beantworten... (*challenge-response*-Prinzip). Beurteile dieses Verfahren!

2. Das digitale Unterschriftsverfahren DSA (*Digital Signature Algorithm*) wurde durch die amerikanische Normungsbehörde zu Beginn der 90er Jahre veröffentlicht. Recherchiere seine Grundelemente!

[7] Steganographische Verfahren sind somit eigentlich keine Verschlüsselungsverfahren.

3 Ein kurzer Ausflug in die Zahlentheorie

3.1 Restklassen

Die Restklasse einer Zahl a modulo einer Zahl m ist die Menge aller Zahlen, die bei der Division durch m denselben Rest ergeben wie a.

Es sei m einer von 0 verschiedene ganze Zahl und a eine beliebige ganze Zahl. Die Restklasse von a modulo m, geschrieben

$a + mZ$,

ist die Äquivalenzklasse von a bezüglich der Kongruenz modulo m. Sie besteht aus allen ganzen Zahlen b, die sich aus a durch die Addition ganzzahliger Vielfacher von m ergeben:

$a + mZ = \{b|b = a + k \cdot m$ für ein $k \in Z\}$ = $\{b|b \equiv a$ (mod m) $\}$.

Beispiele:

- Die Restklasse von 0 modulo 2 ist die Menge der geraden Zahlen.
- Die Restklasse von 1 modulo 2 ist die Menge der ungeraden Zahlen.
- Die Restklasse von 0 modulo m ist die Menge der Vielfachen von m.
- Die Restklasse von 1 modulo 3 ist die Menge $\{$1, 4, 7, 10, . . . , -2, -5, -8, . . . $\}$.

3.2 Primzahlen

Eine Primzahl ist eine natürliche Zahl mit genau zwei verschiedenen natürlichen Teilern, nämlich 1 und sich selbst. Die Primzahlen sind also 2, 3, 5, 7, 11, . . .

Die fundamentale Bedeutung der Primzahlen für viele Bereiche der Mathematik beruht auf den folgenden drei Konsequenzen aus dieser Definition:

- Primzahlen lassen sich nicht als Produkt zweier natürlicher Zahlen, die beide größer als eins sind, darstellen.
- *Lemma von Euklid:* Ist ein Produkt zweier natürlicher Zahlen durch eine Primzahl teilbar, so ist bereits einer der Faktoren durch sie teilbar.
- *Eindeutigkeit der Primfaktorzerlegung:* Jede natürliche Zahl lässt sich als Produkt von Primzahlen schreiben. Diese Produktdarstellung ist bis auf die Reihenfolge der Faktoren eindeutig.

Jede dieser Eigenschaften könnte auch zur Definition der Primzahlen verwendet werden.

Eine natürliche Zahl größer als 1 heißt **prim**, wenn sie eine Primzahl ist, andernfalls heißt sie **zusammengesetzt**. Die Zahlen 0 und 1 sind weder prim noch zusammengesetzt.

Bereits die antiken Griechen interessierten sich für die Primzahlen und entdeckten einige ihrer Eigenschaften. Obwohl sie über die Jahrhunderte stets einen großen Reiz auf die Menschen ausübten, sind bis heute viele die Primzahlen betreffende Fragen ungeklärt. Über zweitausend Jahre lang wusste man keinen praktischen Nutzen aus dem Wissen über die Primzahlen zu ziehen.

Dies änderte sich erst mit dem Aufkommen elektronischer Rechenmaschinen, wo die Primzahlen beispielsweise in der Kryptographie eine zentrale Rolle spielen. Viele Verschlüsselungssysteme, beispielsweise RSA, basieren darauf, dass man zwar sehr schnell große Primzahlen multiplizieren kann, andererseits aber kein effizientes Faktorisierungsverfahren bekannt ist und allem Anschein

nach auch nicht existiert. So ist es innerhalb von Sekunden problemlos möglich, zwei 500-stellige Primzahlen zu finden und miteinander zu multiplizieren. Mit den heutigen Methoden würde die Rückgewinnung der beiden Primfaktoren aus diesem 1000-stelligen Produkt dagegen Millionen von Jahren benötigen.

Es gilt der **Fundamentalsatz der Arithmetik**: Jede positive ganze Zahl lässt sich bis auf die Reihenfolge eindeutig als Produkt von Primzahlen darstellen (siehe Primfaktorzerlegung). Die in dieser Darstellung auftretenden Primzahlen nennt man die **Primfaktoren** der Zahl. Die Schwierigkeiten bei der Primfaktorzerlegung bezeichnet man als Faktorisierungsprobleme. Man versucht sie mit geeigneten Faktorisierungsverfahren zu minimieren.

Aufgrund dieses Satzes, also dass sich jede natürliche Zahl durch Multiplikation von Primzahlen eindeutig darstellen lässt, nehmen die Primzahlen eine besondere atomare Stellung in der Mathematik ein. Alexander K. Dewdney bezeichnete diese als den Elementen der Chemie weitgehend ähnlich.

Mit Ausnahme der Zahl 2 sind alle Primzahlen *p* ungerade, denn alle größeren geraden Zahlen lassen sich außer durch sich selbst und 1 auch noch (mindestens) durch 2 teilen. Damit hat jede Primzahl außer 2 die Form $2 \cdot k + 1$ mit einer natürlichen Zahl k.

3.2.1 Der kleine Satz von Fermat

Für jede Primzahl p und jede natürliche Zahl a mit $0 < a < p$ gilt:

$a^{p-1} \equiv 1$ mod p bzw. $a^{p-1} - 1 \equiv 0$ mod p

Es gibt auch Zahlen, die keine Primzahlen sind, sich aber dennoch, zu einem Teil der Basen a, wie Primzahlen verhalten und somit den kleinen Satz von Fermat erfüllen. Solche Nichtprimzahlen nennt man fermatsche Pseudoprimzahlen. Pseudoprimzahlen, die pseudoprim zu allen Basen a sind, welche nicht Teiler dieser Pseudoprimzahlen sind, nennt man Carmichael-Zahlen.

Besonders in diesem Zusammenhang zeigt sich die Problematik von Pseudoprimzahlen: sie werden von Algorithmen, die den kleinen Satz von Fermat nutzen, um festzustellen ob eine bestimmte Zahl prim ist, fälschlicherweise für Primzahlen gehalten. Wenn allerdings ein Verschlüsselungsverfahren wie RSA eine zusammengesetzte Zahl statt einer Primzahl verwendet, ist die Verschlüsselung nicht mehr sicher. Deshalb müssen bei solchen Verfahren Primzahltests verwendet werden, die mit einer sehr hohen Wahrscheinlichkeit Primzahlen von zusammengesetzten Zahlen unterscheiden können. Diese Wahrscheinlichkeit ist bei Verwendung des kleinen Satzes von Fermat als Basis allein nicht hoch genug, es gibt aber sicherere Primzahltests.

3.2.2 Der kleine Satz von Fermat - Beweis

Der “kleine Satz von Fermat” ist einer der wichtigsten Sätze der Zahlentheorie.

Für alle Primzahlen p und alle natürlichen Zahlen n, die kein Vielfaches von p sind, gilt:

$n^{p-1} - 1$ ist teilbar durch p.

Anders ausgedrückt: n^{p-1} ergibt bei der Division durch p immer den Rest 1.

Beweis

Zuerst betrachten wir die ersten $p - 1$ Vielfachen der Zahl n etwas genauer. Besser gesagt, wir betrachten ihre Reste bei der Division durch p.

$a \cdot n$ und $b \cdot n$ sind zwei unterschiedliche Vielfache von n, wobei a und b größer als 0, aber kleiner als p sein sollen. Außerdem sei $a < b$.

$a \cdot n$ ergebe bei der Division durch p den Quotienten q_a und den Rest r_a, analog ergebe $b \cdot n$ den Quotienten q_b und den Rest r_b. Wegen $a < b$ ist $q_a \leq q_b$.

Dann kann man schreiben:

$a \cdot n = q_a \cdot p + r_a$ und $b \cdot n = q_b \cdot p + r_b$.

Nun kann man anhand dieser beiden Gleichungen leicht zeigen, dass $r_a \leq r_b$ sein muss.

Wir nehmen dazu an, dass die beiden Reste doch gleich seien und erhalten recht bald einen Widerspruch, wodurch bewiesen ist, dass sie nicht gleich sein können. Dazu subtrahieren wir die erste Gleichung von der zweiten:

$a \cdot n = q_a \cdot p + r_a$

$b \cdot n = q_b \cdot p + r_b$

$(b - a) \cdot n = (q_b - q_a) \cdot p$

Der Ausdruck auf der rechten Seite $(q_b - q_a) \cdot p$ ist offensichtlich ein positives Vielfaches von p oder Null, denn die Differenz in der Klammer ist wegen $q_q \leq q_b$ größer gleich Null.

Da wegen $a \leq b$sowohl $(b - a) > 0$ ist, als auch $n > 0$ gilt, ist das nicht möglich, denn die linke Seite kann nicht Null sein.

Falls rechts ein positives Vielfaches von p stehen sollte, müsste auch der Ausdruck links ein Vielfaches von p sein. Damit wäre p entweder ein Teiler von $(b - a)$ oder von n. Auch dies ist unmöglich, denn a und b sind beide nach der Voraussetzung kleiner als p, aber größer als Null, womit $b - a$ sicher kein Vielfaches von p sein kann. Und auch n ist nach der Voraussetzung kein Vielfaches von p.

Die Annahme, dass für irgendwelche a und b die Reste gleich seien, führte also auf eine falsche Aussage. Aus diesem Widerspruchsbeweis folgt, dass alle Reste der ersten $p-1$ Vielfachen von n verschieden sein müssen.

Als Reste bei der Division durch p können nur Zahlen vorkommen, die kleiner als p sind. Bei der Division von $a \cdot n$ durch p kann auch der Rest 0 nicht auftreten, wenn a kleiner als p ist und außerdem n kein Vielfaches von p.

Aus beiden Feststellungen folgt: Die ersten $p - 1$ Vielfachen von n besitzen bei der Division durch p die Reste 1, 2, 3, 4, ... $p - 1$ in irgendeiner Reihenfolge.

Nun bilden wir das Produkt dieser ersten $p-1$ Vielfachen von n und fassen die Zahlen und die ns zusammen:

$n \cdot 2n \cdot 3n \cdot 4n \cdot \ldots \cdot (p-1)n = 1 \cdot 2 \cdot 3 \cdot 4 \cdot \ldots \cdot (p-1) \cdot n^{p-1}$

Eben hatten wir festgestellt, dass diese Vielfachen von n die Reste 1 bis $p - 1$ erzeugen. Ähnlich wie oben schreiben wir nun für die Vielfachen von n:

$n = q_1 \cdot p + r_1$

$2n = q_2 \cdot p + r_2$

$3n = q_3 \cdot p + r_3$

$4n = q_4 \cdot p + r_4$

$\ldots$

$(p-1)n = q_{p-q} \cdot p + r_{p-1}$

Und bilden wie bei das Produkt der Vielfachen, allerdings multiplizieren wir diesmal die rechten Seiten, die ja ebenfalls diese Vielfachen darstellen:

$(q_1 \cdot p + r_1) \cdot (q_2 \cdot p + r_2) \cdot (q_3 \cdot p + r_3) \cdot \ldots \cdot (q_{p-1} \cdot p + r_{p-1})$

Welchen Rest lässt dieses Produkt bei der Division durch p? In jeder Klammer finden wir ein Vielfaches von p, das auf diesen Rest einflusslos ist.

Übrig bleiben die r_1 bis $_{p-1}$, von denen wir wissen, dass sie die Zahlen 1 bis $p - 1$ darstellen. Der gesuchte Rest ist also der Rest des Produktes

$1 \cdot 2 \cdot 3 \cdot 4 \cdot \ldots \cdot (p-1)$

Dies bedeutet: Das Produkt der ersten $p - 1$ Vielfachen von n hat denselben Rest wie das Produkt der ersten $p - 1$ natürlichen Zahlen.

Mit dieser Erkenntnis gehen wir zurück zur Darstellung des Produktes aus

$n \cdot 2n \cdot 3n \cdot 4n \cdot \ldots \cdot (p-1)n = 1 \cdot 2 \cdot 3 \cdot 4 \cdot \ldots \cdot (p-1) \cdot n^{p-1}$

und denken auch hier über den Rest bei der Division durch p nach.

Links steht das Produkt aus den ersten $p-1$ Vielfachen, rechts ein Produkt aus einerseits dem Produkt aus den ersten $p-1$ natürlichen Zahlen und der Potenz n^{p-1}. Wir wissen, dass die linke Seite denselben Rest hat wie der erste Faktor rechts.

Was bedeutet das für n^{p-1}?

Überlegen wir das anhand einer übersichtlicheren Situation:

$a = b \cdot c$

Auch hier sollen a und b denselben Rest r bei der Division durch p haben. Außerdem seien a, b, $c > 0$ und r, $r_c < p$. Schreiben wir wieder Gleichungen für die Divisionen mit Rest durch p:

$a = q_a \cdot p + r$

$b = q_b \cdot p + r$

$c = q_c \cdot p + r_c$

Wir subtrahieren die ersten beiden Gleichungen, lösen nach a auf und ersetzen a in $a = b \cdot c$ durch diesen Ausdruck:

$a - b = (q_a - q_b) \cdot p \mid +b$

$a = (q_a - q_b) \cdot p + b$

und daher

$(q_a - q_b) \cdot p + b = b \cdot c$

In dieser Gleichung ersetzen wir c gemäß $c = q_c \cdot p + r_c$:

$(q_a - q_b) \cdot p + b = b \cdot q_c \cdot p + r_c \mid : b$ ($b > 0$)

$\frac{(q_a - q_b) \cdot p}{b} + 1 = q_c \cdot p + r_c \mid$ - $q_c \cdot p$

$\frac{(q_a - q_b) \cdot p}{b} - q_c \cdot p + 1 = r_c$

Fassen wir links zusammen (gemeinsamer Nenner!) so erhalten wir:

$\frac{(q_a - q_b - q_c b) p}{b} + 1 = r_c$

Weil im letzten Ausdruck r_c und 1 ganzzahlig sind, muss auch der Bruch ganzzahlig sein. Außerdem kann b kein Teiler der Primzahl p sein. Damit ist der Bruch nicht nur ganzzahlig, sondern ein Vielfaches von p. Der Rest r_c ist also 1 plus irgendein Vielfaches von p.

Daraus folgt, dass c bei der Division durch p den Rest 1 lässt, denn Vielfache von p haben auf den Rest keinen Einfluss.

Gleich ist es geschafft. Denn wenn wir erneut zur Gleichung

$n \cdot 2n \cdot 3n \cdot 4n \cdot \ldots \cdot (p-1)n = 1 \cdot 2 \cdot 3 \cdot 4 \cdot \ldots \cdot (p-1) \cdot n^{p-1}$

zurückkehren, sind wir ein gutes Stück klüger als vorhin, denn wir wissen nun, welchen Rest n^{p-1} haben muss, wenn die Reste der blauen und der grünen Zahl einander gleich sind, und das sind sie!

Wir wissen nun: n^{p-1} hat bei der Division durch p den Rest 1. Und das genau ist der Satz von Fermat.

3.2.3 Der Satz von EULER[8]

Der Satz von Euler verallgemeinert den kleinen Fermatschen Satz und wird deshalb auch Satz von Euler-Fermat genannt.

Zur Erinnerung - der kleine Fermat besagt:

a^{p-1} mod $p = 1$

[8] Quelle: Mag. Walter Wegscheider aus www.austromath.at/medienvielfalt, Jan. 2006

Satz

Sind a und n zwei natürliche teilerfremde Zahlen, dann gilt:
$a^{\phi(n)}$ mod $n = 1$

- $\phi(n)$ ist die Anzahl der zu n teilerfremden natürlichen Zahlen (die Anzahl aller Zahlen $\leq n$, deren größter gemeinsamer Teiler mit n gleich 1 ist).
- Beispiele:
  ϕ(12) = 4, teilerfremde Zahlen sind $\{$1, 5, 7, 11$\}$
  ϕ(13) = 12, alle Zahlen von 1 bis 12 sind teilerfremd, da 13 eine Primzahl ist
  ϕ(14) = 6, teilerfremde Zahlen sind $\{$1, 3, 5, 9, 11, 13$\}$
- Zu einer Primzahl p sind alle Zahlen von 1 bis $(p-1)$ teilerfremd - daraus folgt: $\phi(p) = p - 1$.
- Für prime Moduln p geht der Satz von Euler daher in den kleinen Satz von Fermat über.
- Für das Produkt zweier Primzahlen p und q gilt weiters:
  $\phi(p \cdot q) = (p-1) \cdot (q-1)$
- Somit:
  $a^{\phi(n)\cdot\phi(m)}$ mod $n \cdot m = 1$
  für n und m prim
  $a^{(n-1)(m-1)}$ mod $n \cdot m = 1$ (a teilerfremd zu m und n)

Beispiel:

Was ist die letzte Dezimalstelle von 7^{333}?

Die Frage kann umgedeutet werden zu: 7^{333} mod $10 = x$ (gefragt ist der Rest bei Division durch 10).

Wir wissen: ϕ(10) = 4 und damit 7^4 mod $10 = 1$ und zerlegen daher 333 geschickt:
$333 = 4 \cdot 83 + 1$
$7^{333} = 7^{(4 \cdot 83+1)}$
$((7^4)^8 3 \cdot 7)$ mod $10 = (1^{83} \cdot 7$ mod $10 = 7$
Das ist die Antwort - die letzte Stelle lautet 7.

3.3 Der diskrete Logarithmus

Eine wichtige Erfahrung für uns Mathematiker/innen ist es, Rechenoperationen wieder umkehren zu können - in der Volksschule lernen wir das Addieren und das Subtrahieren: Weil das Subtrahieren etwas unangenehmer ist als das Addieren, führen wir gewöhnlich die Probe durch Addieren aus, das heißt, wir machen von der Umkehrbarkeit der Rechenoperation praktischen Gebrauch. Eine ähnliche Situation stellt sich für uns beim Multiplizieren: Wer erinnert sich nicht an die Mühen des schriftlichen Dividierens? Das Bestimmen des Quotienten und des Restes beim Dividieren gehört zu den anspruchsvollsten Rechenfertigkeiten, mit denen wir in der Volksschule befasst worden sind!

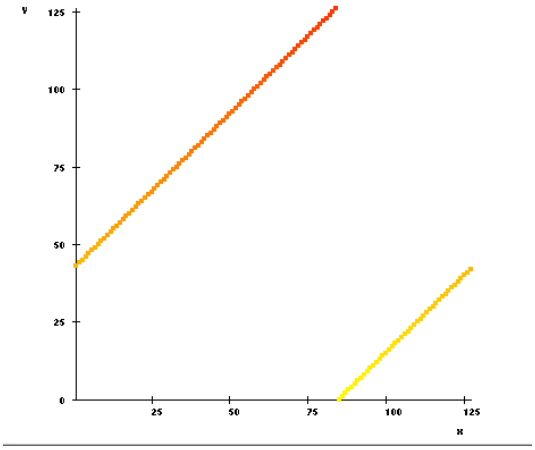
Umkehroperationen spielen später in Form von passenden Äquivalenzumformungen für Gleichungen und Ungleichungen oder als Umkehrfunktionen eine wichtige Rolle: Wir lernen das

Wurzelziehen als Umkehrung zum Potenzieren kennen, also als Aufgabe, die Basis bei bekanntem Exponent und bekannter Potenz zu bestimmen. Da das Quadratwurzelziehen mit einem schriftlichen Algorithmus durchaus mühsam ist, wird es seit wenigstens 30 Jahren nicht mehr unterrichtet und länger schon nicht mehr praktisch angewendet... Das Logarithmieren als Umkehrung zur Exponentiation ist noch schwieriger: Der Exponent ist bei bekannter Basis und Potenz zu bestimmen.

Mit Logarithmenbuch, Rechenschieber, Taschenrechner oder Computer-Algebra-System ist das Durchführen einer Operation und ihrer Umkehrung vergleichsweise einfach. Besonders interessant ist dies jedoch, wenn man diese Rechenoperationen mit Restklassen durchführt:

3.3.1 Addition einer Konstante

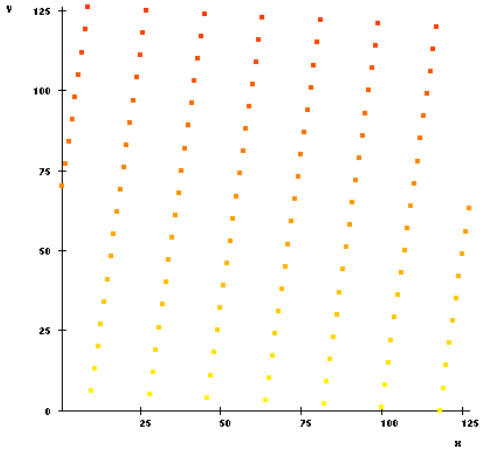
Wir stellen die Funktion $f(x) = x + 42 \; mod \; 127$ grafisch dar: Wir erkennen eine bijektive Zuordnung (“1-1-Funktion”; sie besteht den “horizontal-line-test”).



Die Umkehrung, die Subtraktion, kann offenbar mit der “Modulo-Arithmetik” eindeutig durchgeführt werden.

3.3.2 Die lineare Funktion

Die lineare Funktion $y = k \cdot x + d$ ist ein Paradebeispiel einer biejektiven Funktion. Diese Eigenschaft ändert sich radikal, wenn wir Restklassen verwenden. Um dies zu zeigen, stellen wir die Funktion $f(x) = 7 \cdot x + 63 \; mod \; 127$ grafisch dar:

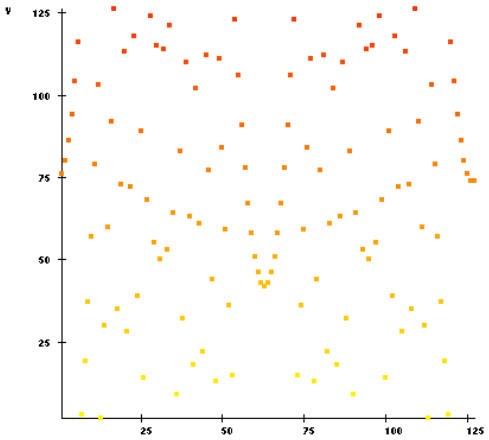


Der Funktionsgraph (er besteht aus einzelnen, “diskreten”, Punkten) zeigt keinesfalls eine 1-zu-1-Zuordnung! Obwohl es nicht möglich ist, zu einem Funktionswert ein eindeutiges zugehöriges Argument zu finden, beobachten wir aber, dass die Funktionswerte regelmäßig auftreten... Aus

“Erfahrung” wissen wir, dass in der Restklassen-Arithmetik keine eindeutige Division besteht - beim wiederholten Addieren (also Multiplizieren) einer Zahl wiederholen sich (nach wenigen Schritten) die Ergebnisse. Aus diesem Grund verläuft ja der Funktionsgraph sägezahnartig.

3.3.3 Die Quadrat-Funktion

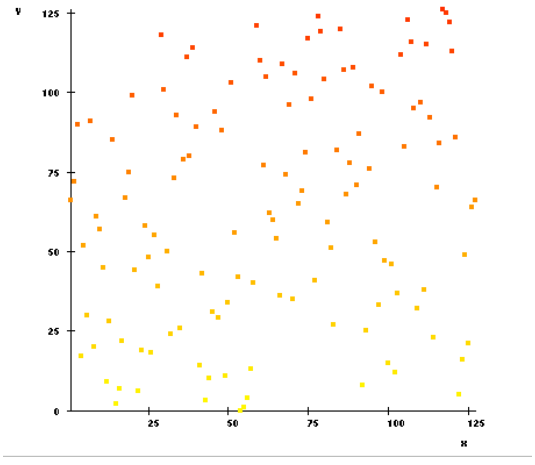
Obwohl nach dem letzten Abschnitt nicht zu erwarten ist, dass für eine Restklasse Quadratwurzeln eindeutig bestimmt werden können, stellen wir die Funktion $f(x) = (x-63)^2 \ mod \ 127$ grafisch dar und philosophieren ber mögliche Umkehroperationen:



Manche Punkte ergeben zusammen ein gewisse Ähnlichkeit zur Parabel, wie wir von der Darstellung der Quadratfunktion für reelle Zahlen her kennen. Die zahlreichen Punkte dazwischen machen die Umkehrung schwer. Der geübte Leser / die geübte Leserin ist eingeladen, die Regelmäßigkeit der Punkte (zum Beispiel in Bezug auf das Argument $x = 63$) zu untersuchen!

3.3.4 Die Exponentialfunktion

Wir stellen die Funktion $f(x) = 3^x + 63 \ mod \ 127$ grafisch dar und diskutieren die Umkehrung, das Auffinden des Logarithmus. Hier kann gezeigt werden, dass dieses Unterfangen für Restklassen mit großem m mit großer Wahrscheinlichkeit scheitert:



Die Punkte, die die diskreten Zahlenpaare $(x/f(x))$ darstellen, scheinen wahllos ber das Koordinatensystem verstreut zu liegen. Die Schwierigkeit in einer Restklasse den Logarithmus zu bestimmen, wird als das “Diskreter-Logarithmus-Problem” (DLP) bezeichnet. Es spielt in der Kryptographie eine wichtige Rolle für moderne Verschlüsselungsverfahren.

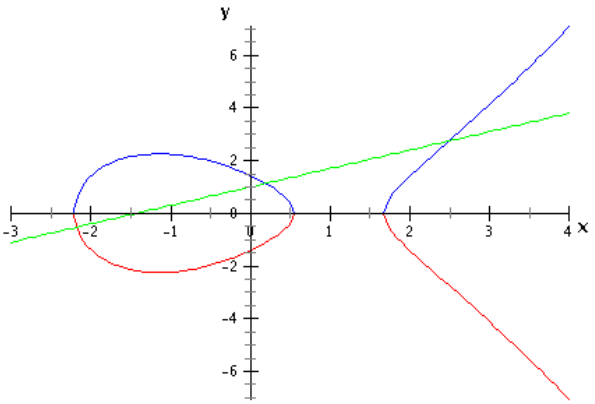
3.4 Elliptische Kurven

3.4.1 Grundlagen

Elliptische Kurven sind nicht etwa Ellipsen, sondern Punktmengen, die die Gleichung

$$y^2 = x^3 + a \cdot x + b$$

erfüllen. Für die Form der Kurve spielen die Variablen a und b die Rolle von Parametern, wir wählen beispielsweise $a = -4$ und $b = 2$. Wie man leicht nachrechnet, liegen dann die Punkte $P_1 = (-2, -\sqrt{2})$ und $P_2 = (0, \sqrt{2})$ auf der Kurve, d.h. sie erfüllen mit ihren Koordinaten die Gleichung $y^2 = x^3 - 4x + 2$.



Interessant ist, dass man für Punkte P_1 und P_2 auf der Kurve eine Addition $P_1 + P_2$ definieren kann, die die folgenden Regeln befolgt: sie ist kommutativ, assoziativ, es gibt ein neutrales Element 0 und zu jedem Punkt $P(x_1, y_1)$ einen negativen Punkt $-P$ (inverses Element), mit der Eigenschaft $-P_1 = (x_1, -y_1)$. Wie man unschwer sieht, erhält man den inversen Punkt durch Spiegeln an der x-Achse.

Die Addition kann somit geometrisch leicht beschrieben werden: Für die Addition zweier Punkte P_1 und P_2 schneidet man die Gerade durch P_1 und P_2 mit der Kurve: Dieser Schnittpunkt Q wird an der x-Achse gespiegelt und ergibt $-Q = P_1 + P_2$.

Interessant ist nun, dass man Punkte auch mit sich selbst addieren kann: $P + P = 2 \cdot P$. Geometrisch bedeutet dies, dass bei der Addition nicht eine Gerade durch zwei verschiedene Punkte P_1 und P_2 der Kurve (also eine Sekante) sondern eine Tangente im Punkt P an die Kurve gelegt werden muss (der Schnittpunkt dieser Tangente mit der Kurve wird an der x-Achse gespiegelt und ergibt den Punkt $2 \cdot P$. Auf diese Weise lässt sich ein Produkt $k \cdot P$ definieren.

Unter bestimmten Voraussetzungen existiert zu jedem Punkt Q auf der Kurve ein Punkt P mit der Eigenschaft $Q = k \cdot P$. Das Problem aus den gegebenen Punkten P und Q dieses k zu berechnen, heißt das **diskrete Logarithmusproblem für elliptische Kurven** (*ECDLP*): k heiß der diskrete Logarithmus von Q zur Basis P. Es ist ähnlich schwierig zu lösen wie das diskrete Logarithmusproblem (*DLP*) und spielt eine wichtige Rolle in verschiedenen Verschlüsselungsverfahren. Dabei werden alle Rechenoperationen in Restklassenarithmetik modulo p ausgeführt: In diesem Fall entstehen keine geschlossenen Kurven mehr, sondern diskrete Punktmengen.

3.4.2 Beispiel: F_{23}

Für die elliptische Kurve $y^2 = x^3 + x$ ($a = 1$, $b = 0$) erhält man mod 23, ausgehend vom Punkt $P(9, 5)$ folgende 23 Punkte der Kurve: $(0, 0)$, $(1, 5)$, $(1, 18)$, $(9, 5)$, $(9, 18)$, $(11, 10)$, $(11, 13)$, $(13, 5)$, $(13, 18)$, $(15, 3)$, $(15, 20)$, $(16, 8)$, $(16, 15)$, $(17, 10)$, $(17, 13)$, $(18, 10)$, $(18, 13)$, $(19, 1)$, $(19, 22)$, $(20, 4)$, $(20, 19)$, $(21, 6)$, $(21, 17)$.

4 Ein Ausschnitt aus der Wahrscheinlichkeitsrechung

4.1 Einleitung

Für die Kryptographie ist es von Bedeutung herauszufinden, wie "wahrscheinlich" es ist, einen bestimmten Rechenschritt zufällig durchzuführen - und damit etwa zufällig einen bestimmten Code zu knacken.

Die Behandlung eines konkreten Problems geht aber weit über die Grundlagen der Wahrscheinlichkeitsrechnung hinaus; an dieser Stelle wird dennoch eine Hinführung zu diesem Problem versucht. Der geneigte Leser / die geneigte Leserin wird ersucht, zur Wahrscheinlichkeitsrechnung in anderen Werken oder Medien nachzulesen, falls dies erforderlich ist (ich wende mich mit dieser Aufforderung ganz an die jungen Mathematiker/innen, die in der Schule noch keine Einführung in die Wahrscheinlichkeitsrechnung hören konnten ;-)).

4.2 Häufigkeiten

Betrachten wir folgende Situation: Für eine Partnerarbeit soll eine zweite Person aus einer Gruppe von 10 Schüler/innen zufällig ausgewählt werden. In dieser Gruppe sind sind 4 Mädchen und 6 Burschen.

4.2.1 Relative Häufigkeit

Im letzten Beispiel können wir auch sagen, "4 von 10 Personen sind weiblich". Wir beschreiben diesen Anteil mit dem Bruch $\frac{4}{10}$ oder mit Mitteln der Prozentrechung durch den Anteil von 40%, und bezeichnen dies als die **relative Häufigkeit**[9]. Der Anteil $\frac{4}{10}$ beschreibt den Anteil der Mädchen in einer beliebig groön Personengruppe; beispielsweise bedeutet der gleiche Anteil bei 200 Personen, dass $200 \cdot \frac{4}{10} = 80$ Mädchen in der Personengruppe sind.

4.2.2 Laplacesche Wahrscheinlichkeitsdefinition

Betrachten wir eine Gruppe von 1000 Personen. 3 von ihnen haben am 2. Mai Geburtstag. Die relative Häufigkeit beträgt dafür $\frac{3}{1000} = 0,3\%$.

4.2.3 Wahrscheinlichkeit

In bestimmten einfachen Situationen kann die relative Häufigkeit mit der Wahrscheinlichkeit gleichgesetzt werden. Wir übertragen das auf eine Personengruppe mit "vielen" Personen. Wie groß ist die Wahrscheinlichkeit, dass eine zufällig befragte Person, an einem ganz bestimmten Tag im Jahr ihren Geburtstag hat? Unter der Voraussetzung, dass jeder Tag im Jahr mit der gleichen Wahrscheinlichkeit als Geburtstag dieser Person in Frage kommt, beträgt die Wahrscheinlichkeit $\frac{1}{365}$[10].

Unter der Voraussetzung dass alle betrachteten Ereignisse mit der gleichen Wahrscheinlichkeit eintreten, wählt man in der Wahrscheinlichkeitsrechnung folgende einfache Berechnungsmethode: Wir erhalten die Wahrscheinlichkeit dafür, dass **ein bestimmtes Ereignis** unter einer riesigen Anzahl von Versuchen eintritt als Bruch:

[9] Im Gegensatz dazu war in der Einleitung des Kapitels die **absolute Häufigkeit**, also "4 Personen" gegeben.

[10] ... heuer ist kein Schaltjahr!

$$P = \frac{g}{m}$$

Dabei bezeichnet P die Wahrscheinlichkeit ("*probability*"), g die "Anzahl der günstigen Fälle" und m die "Anzahl der möglichen Fälle".

Wenden wir dies auf eine ähnliche Frage an: Wie groß ist die Wahrscheinlichkeit, dass in einer Gruppe von vielen Personen, eine beliebig ausgewählte Person im Juli Geburtstag hat? Wir berlegen: Das Jahr hat 365 Tage, also ist $m = 365$. Der Monat Juli hat 31 Tage, damit ist die Anzahl der günstigen Fälle $g = 31$. Die Wahrscheinlichkeit beträgt daher:

$P = \frac{g}{m} = \frac{31}{365} = 0,084931... \approx 8,5\%$

4.2.4 Zero-Knowledge-Problem

Ist die Wahrscheinlichkeit, zufällig die Kenntnis der Nachricht (des Schlüssels) zu kennen, $p = \frac{1}{2}$, so kann ein Betrüger die Nachricht bei 2, 3, 4, ... n Runden nur mehr mit der Wahrscheinlichkeit von $(\frac{1}{2})^2, (\frac{1}{2})^3, (\frac{1}{2})^4 ... (\frac{1}{2})^n$ erraten. Diese Wahrscheinlichkeit, dass ihm das in allen n Runden gelingt, geht somit mit wachsender Zahl n von Runden gegen Null.

4.2.5 Gegenwahrscheinlichkeit

Unter der "Gegenwahrscheinlichkeit" P' für das Eintreten eines bestimmten Ereignisses verstehen wir die Wahrscheinlichkeit dafür, dass dieses Ereignis **nicht** eintritt. Für das letzte Beispiel könten wir die Wahrscheinlichkeit dafür berechnen, dass eine Person **nicht** im Juli Geburtstag hat: $m = 365$, $g = 365 - 31 = 334$

$P' = \frac{g}{m} = \frac{334}{365} = 0,915068... \approx 91,5\%$

Wir können leicht nachrechnen, dass $P + P' = 1$ gilt.

$P + P' = \frac{31}{365} + \frac{334}{365} = \frac{31+334}{365} = \frac{365}{365} = 1.$

Die letzte Beziehung, $P + P' = 1$ wird höfig dazu verwendet, P bei bekannter Gegenwahrscheinlichkeit P' zu berechnen ...

4.3 Das Geburtstagsproblem

4.3.1 Die Aufgabenstellung

Wir betrachten eine Gruppe von n Personen (zB im Rahmen einer - größeren - Geburtstagsparty ;-)). Wie groß ist die Wahrscheinlichkeit, dass von n **zufällig** anwesenden Personen mindestens zwei am selben Tag Geburtstag haben?

Um diese Aufgabe löen zu können, brauchen wir noch ein paar Klarstellungen. "Am selben Tag Geburtstag haben" bedeutet **nicht**, dass die beiden Personen **gleich alt** sind, sondern dass sie am selben Tag im Jahr ihren Geburtstag feiern. Und um die Aufgabe wirklich genau und richtig zu behandeln, löen wir das "Geburtstagsproblem" in zwei Schritten:

1. Wie groß ist die Wahrscheinlichkeit, dass die zwei Personen an einem **ganz bestimmten Tag** gemeinsam Geburtstag haben?

2. Wie groß ist die Wahrscheinlichkeit, dass die zwei Personen an **irgendeinem Tag** gemeinsam Geburtstag haben?

4.3.2 Wir lösen das Geburtstagsproblem

Zunächst zum ersten Schritt: Wir berechnen die Wahrscheinlichkeit dafür, dass zwei Personen an einem ganz bestimmten Tag (zum Beispiel am Geburtstag der Person, die eine - riesige - Geburtstagsparty veranstaltet) Geburtstag haben?

Da es auch vorkommen kann, dass 3, 4 oder noch mehr Personen am gleichen Tag Geburtstag haben, rechnen wir mit der Gegenwahrscheinlichkeit: Wie groß ist die Wahrscheinlichkeit, dass in einer Gruppe von n Personen **keine** zwei Personen an einem bestimmten Tag Geburtstag haben?

Dazu schreiben wir die Geburtstage aller Personen in einem Vektor mit n Einträgen:

$(34, 128, 3, ...249)$

Für jeden Eintrag gibt es 365 verschiedene Möglichkeiten (dabei gehen wir davon aus, dass kein Schaltjahr vorliegt). Wählen wir einen ganz bestimmten Tag (also beispielsweise den Geburtstag des Partygebers) aus, so sind für das Gegenereignis nur 364 Tage für die Geburtstage möglich. Für jede ausgewählte Person ist die Wahrscheinlichkeit für ihren Geburtstag $\frac{364}{365}$.

Für 2 Personen gilt dann: Die erste Person darf nicht an diesem bestimmten Tag Geburtstag haben **und** die zweite Person ebenfalls nicht. Die Wahrscheinlichkeit dafür ist $\frac{364}{365} \cdot \frac{364}{365}$.

Für 3 Personen gilt: Die erste Person darf nicht an diesem bestimmten Tag Geburtstag haben, die zweite Person nicht **und** die zweite Person ebenfalls nicht. Die Wahrscheinlichkeit dafür ist $\frac{364}{365} \cdot \frac{364}{365} \cdot \frac{364}{365}$.

u.s.f....

Für n Personen beträgt die Gegenwahrscheinlichkeit: $P' = \frac{364^n}{365^n} \quad \Rightarrow \quad P = 1 - \frac{364^n}{365^n}$.

In der zweiten Aufgabenstellung legen wir den untersuchten Tag nicht fest. Wir berechnen die Gegenwahrscheinlichkeit für das Ereignis “zwei Personen haben an irgendeinem Tag (im Jahr) gemeinsam Geburtstag”. Dazu bilden wir das Gegenereignis “alle Geburtstage für n Personen sind verschieden”: Für n Personen haben wir wieder 365^n mögliche Geburtstagskombinationen (inklusive aller doppelten, dreifachen, etc...). Sollen - für das Gegenereignis - alle Personen verschiedene Geburtstage haben, so berechnen wir die Anzahl der günstigen Fälle schrittweise:

1. Die $1.$ Person kann an irgendeinem Tag Geburtstag haben: 365 Möglichkeiten
2. Die $2.$ Person kann nur mehr an 364 Tagen Geburtstag haben; 364 Möglichkeiten
3. Für die $3.$ Person erhalten wir 363 Möglichkeiten
4. u.s.f....

Für n Personen (aber jedenfalls höchstens 365 viele!) erhalten wir die Gesamtzahl aller Geburtstagsmöglichkeiten:

$365 \cdot 364 \cdot 363 \cdot ...(365 - n + 1) = \frac{365!}{(365-n)!}$

Damit erhalten wir für die Gegenwahrscheinlichkeit P':

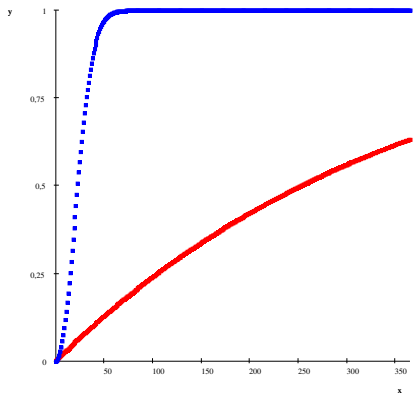
$P' = \frac{\frac{365!}{(365-n)!}}{365^n}$

Die Löung für das Geburtstagsproblem ist dann $P = 1 - P' = 1 - \frac{\frac{365!}{(365-n)!}}{365^n}$.

4.3.3 MuPAD:

Wir stellen beide Wahrscheinlichkeiten für n Personen ($n < 366$) mit MuPAD grafisch dar:

```
>> p1 := n -> 1 - (364^n) / (365^n)

                          n -> 1 - 364^n/365^n
>> p2 := n -> 1 - (365!/(365-n)!)/(365^n)

                n -> 1 - (fact(365)/fact(365 - n))/365^n
>> plot(plot::Pointlist([i,p1(i)] $ i=1..365, Color=RGB::Red), plot::Pointlist([i,p2(i)] $ i=1..365, Color=RGB::Blue));
```



Die Wahrscheinlichkeit, dass von n Personen zwei an einem beliebigen Tag im Jahr gemeinsam Geburtstag haben ist sehr viel größer als dass zwei Personen an einem bestimmten Tag im Jahr gemeinsam Geburtstag haben! Ab etwa 80 Personen ist die Wahrscheinlichkeit, dass zwei an einem beliebigen Tag gemeinsam Geburtstag haben knapp 100%...

4.3.4 Und was hat das mit Kryptographie zu tun?

So genannte **Hashverfahren** liefern zu einer bestimmten Nachricht oder zu einem bestimmten Datenbestand einen Wert, der sich bereits bei geringen Änderungen an der Nachricht oder am Datenbestand, deutlich ändert. Dabei ist es sehr wichtig, dass nicht zwei verschiedene Nachrichten den gleichen Hashwert ergeben - man nennt diese Eigenschaft die **Kollisionsfreiheit** eines Hashverfahrens. Die Löung des Geburtstagsproblem bedeutet zum Leidwesen aller Kryptologen: Während die Wahrscheinlichkeit, dass zu einem bestimmten Hashwert eine zweite Nachricht mit dem gleichen Hashwert gefunden werden kann, relativ klein ist, ist die Wahrscheinlichkeit, zwei beliebige Nachrichten mit dem gleichen Hashwert zu finden, wesentlich größer ...

5 Aufgaben aus der Kryptographie mit MuPAD lösen

Nicht alle kryptographischen Verfahren lassen sich gut mit MuPAD lösen. Der Eigenschaft als Computer-Algebra-Systems folgend werden die Verfahren hier vorgestellt, die mit Matrizenoperationen, mit Hilfe von Primzahlen oder Modulo-Arithmetik durchgefhrt werden.

5.1 Einleitung in MuPAD

MuPAD wurde ursprünglich an der Universität Paderborn (Deutschland) entwickelt. Die Weiterentwicklung hat nunmehr SciFace GmbH (http://www.sciface.com) über. Für Schülerinnen und Schüler steht eine eigene Lizenz zur nicht kommerziellen Nutzung des Programmes zur Verfügung. Zur Zeit kann die Version 2.5 mit einer kostenfreien Lizenz, die Version 3.1 gegen einen geringen Betrag verwendet werden: Während die Version 2.5 den mathematischen Kern und eine einfache Ein- und Ausgabe zur Verfügung stellt, werden mit der Version 3.1 die Erstellung von so genannten“Notebooks” (= mathematische Texte, Aufsätze) und das Zusammenspiel zu moderner Software unterstützt. Die Beispiele in diesem Skriptum beziehen sich vorwiegend auf die Version 2.5[11].

Alle Eingaben in MuPAD werden in einer oder mehreren Zeilen eingegeben. Die [Enter]-Taste schließt eine Eingabe ab und veranlasst die Berechnung des letzten Ausdrucks. Ein Zeilenumbruch zwischen mehreren Eingaben wird mit Hilfe der Tastenkombination [Shift] + [Enter] eingegeben[12]. Ein Doppelpunkt (:) am Ende einer Zeile unterdrückt die Ausgabe des (Zwischen)Ergebnisses. Auf diese Weise eignet sich der Doppelpunkt zum Aneinanderfügen mehrerer Rechenausdrücke. Ein Strichpunkt (;) veranlasst die anschließnde Ausgabe. In den meisten Anwendungen wird es jedoch bersichtlicher sein, jede Anweisung in eine eigene Zeile zu stellen.

Die (zahllosen) Methoden eines Moduls werden werden mit Hilfe eines doppelten Doppelpunktes (::) referenziert. Links steht der Name des Moduls, rechts der Name der gewünschten Methode, beispielsweise:

```
>> numlib::toAscii("Informatik ist schoen");
...
>> plot(plot::Pointlist([i,i+42 mod 127] $ i = 1 .. 127));
```

5.1.1 Restklassen

Das Rechnen mit Restklassen spielt bei vielen Verschlüsselungsverfahren eine wichtige Rolle. Wir bestimmen den ganzzahligen Rest bei der Division einer Zahl durch einen bestimmten “Modulus” m:

```
>> 11 mod 3;

                                  2
>> _mod(11,3);

                                  2
>>
```

Der **Operator** `mod` gibt den ganzzahligen Rest bei der Division $11 : 3 = 3 \cdot 3 + 2$, also 2 aus. Zusätzlich zur Operatorenschreibweise kann auch die Funktionenschreibweise `_mod(11,3)` verwendet werden.

Diesen ganzzahligen Rest kann man natürlich auch für große Zahlen, etwa für Potenzen berechnen. Wir besprechen dies im nächsten Abschnitt.

[11] Alle Arbeiten, die Texterfassung sowie die Rechnungen mit MuPAD, wurden auf einem PC unter SuSE Linux 9.0 durchgeführt.

[12] Beachten Sie bitte diesen Unterschied zu anderen, weit verbreiteten Computer-Algebra-Systemen!

5.1.2 Modulares Potenzieren

Wir betrachten folgende Potenzen von 2 mod 3:
$2^0 = 1$
$2^1 = 2$
$2^2 = 4 \equiv 1$ mod 3
$2^3 = 8 \equiv 2$ mod 3
$2^4 = 16 \equiv 1$ mod 3 usw.

Diese Rechnungen lassen sich mit MuPAD entweder mit Hilfe des Operators `mod` nach dem Potenzieren oder mit Hilfe der Funktion `powermod()` ausführen:

```
>> 8^2 mod 7;

                                  1
>> powermod(8,2,7);

                                  1
>>
```

Die MuPAD-Funktion `powermod(Basis, Hochzahl, Modul)` berechnet den ganzzahligen Rest der angegebenen Potenz in Bezug auf den festgelegten Modul.

5.1.3 Beispiel - Punkte plotten

Die in vorangegangenen Abschnitt dargestellten Funktionsgraphen wurden mit folgenden MuPAD-Zeilen erzeugt:

```
>> plot(plot::Pointlist([i,i+42 mod 127] $ i = 1 .. 127));

>> plot(plot::Pointlist([i,7*i + 63 mod 127] $ i = 1 .. 127));

>> plot(plot::Pointlist([i,((i-63)^2 + 42) mod 127] $ i = 1 .. 127));

>> plot(plot::Pointlist([i,(3^i + 63) mod 127] $ i = 1 .. 127));
```

Beachten Sie bitte, dass die Methode `Pointlist()` von Version 2.5 nach 3.1 abgeändert wurde.

5.1.4 Zahlentheorie - numlib

Die *library for number theory* stellt zahlreiche Funktionen zur Zahlentheorie zur Verfügung.

`numlib::toAscii(s)` - gibt von allen Zeichen der angegebenen Zeichenkette s den zugehörigen Ascii-Code aus. Das Ergebnis ist eine Liste der Ascii-Indizes.

```
>> numlib::toAscii("Informatik ist schoen");

[73, 110, 102, 111, 114, 109, 97, 116, 105, 107, 32, 105, 115, 116, 32, 115, 99, 104, 111, 101, 110]
```

`numlib::fromAscii(cl)` - gibt einen String aller Zeichen aus, die durch die angegebene Indexliste cl beschrieben werden.

```
>> numlib::fromAscii([73, 110, 102, 111, 114, 109, 97, 116, 105, 107, 32, 105, 115, 116, 32, 115, 99, 104, 111, 101, 110])

                          "Informatik ist schoen"
```

5.1.5 Programmieren mit MuPAD

Unter einem Programm verstehen wir die Zusammenfassung mehrerer Befehle zur Programmsteuerung oder Rechenbefehle. Jedes Programm beginnt mit dem reservierten Wort `proc` und einer entsprechenden Parameterliste. Nach der allfälligen Deklaration lokaler Variable stehen die eigentlichen Befehle zwischen den reservierten Worten`begin` und `end_proc`. Wird die letzte Zeile mit einem Doppelpunkt beendet, so erhält man eine Mitteilung über die (syntaktisch richtige) Zuweisung der Befehle - alternativ dazu könnte man die letzte Zeile auch mit einem Doppelpunkt abschließen.

```
>> y := proc(x)
   begin
     return (x*x);
   end_proc;

                              proc y(x) ... end
>> y(2);

                                      4
>> y(-3)

                                      9
```

Das obige Beispiel kann als einfache Funktionsdeklaration auch so realisiert werden:

```
>> y := x -> x*x;

                                  x -> x*x
>> y(-5);

                                    25
>> y(1.2);

                                   1.44
```

Umfangreichere Programme werden in den folgenden Abschnitten verwendet. Dabei werden auch Verzweigungen und Schleifen vorgestellt.

5.1.6 Text in Zahlen umwandeln und umgekehrt

Für kryptographische Anwendungen muss der Klartext üblicherweise zunächst in eine Zahlenfolge transformiert werden. Auf diese Zahlenfolge werden dann beim Verschlüsseln die entsprechenden Algorithmen angewendet. Das Ergebnis beim Entschlüsseln liegt in Zahlen vor - diese müssen vor der Ausgabe wieder in den entsprechenden Text umgewandelt werden.

Wir verwenden dazu zwei kurze MuPAD-Programme, `text2zahl(text)` und `zahl2text(zahl)`:

```
>> text2zahl:=proc(text)
    local atext, laenge, zahl;
    begin
      atext := numlib::toAscii(text);
      laenge:=nops(atext);
      m:=0;
      (m := m + atext[laenge - i] * 10^(i*3)) $ i=0..laenge-1;
      return (m);
    end_proc:

>> zahl2text:=proc(zahl)
   local atext, laenge;
    begin
      atext:=[];
      repeat
        atext :=[modp(zahl,1000)].atext;
        zahl:=floor(zahl/1000);
        until zahl/1000 < 1 end_repeat;
      atext:=[zahl].atext;
```

```
      return(numlib::fromAscii(atext));
    end_proc:

>> text2zahl("Kryptographie ist ein Thema aus den Fachbereichen Mathematik, Informatik und Physik");

751141211121161111031140971121041051010321051151160321011051100320841041 01\
109097032097117115032100101110032070097099104098101114101105099104101110 03\
207709711610410110909711610510704403207311010211111410909711610510703211 71\
101000320801041211151051 07

>> zahl2text(751141211121161111031140971121041051010321051151160321011051100320841041 01\
109097032097117115032100101110032070097099104098101114101105099104101110 03\
207709711610410110909711610510704403207311010211111410909711610510703211 71\
101000320801041211151051 07);

"Kryptographie ist ein Thema aus den Fachbereichen Mathematik, Informatik \
und Physik"
```

Beachten Sie, wie der Text in dreistellige ASCII-Zahlen und diese Dreiergruppen in eine (riesige) Ganzzahl umgewandelt wird! Die Zahl wird deshalb ebenfalls durch die Division durch 1000 in Dreiergruppen zusammengefasst in die einzelnen Buchstaben transformiert...

5.1.7 Primzahlen

Primzahlen haben genau 2 Teiler, die Zahl 1 und sich selbst. MuPAD testet beliebige Zahlen mit der Funktion `isprime()`, ob eine Primzahl vorliegt, alternativ kann man auch die Faktorisierung der Zahl ausgeben:

```
>> isprime(97);

                                TRUE
>> factor(97);

                                 97
>> isprime(1040);

                                FALSE
>> factor(1040);

                                 4
                               2  5 13
```

`ifactor()` liefert die Faktorisierung von ganzen Zahlen, `factor()` zerlegt Polynome in Primfaktoren.

Noch wichtiger ist die Möglichkeit, mit MuPAD Primzahlen zu bestimmen:

```
>> nextprime(1000000);

                              1000003
>> numlib::prevprime(1000000);

                              999983
>> ithprime(1000);

                               7919
>> ithprime(100000);

                              1299709
>> ithprime(5);

                                11
```

`nextprime()` liefert die zur angegebenen Zahl nächst größere Primzahl. Die Funktion `ithprime()` liefert die so und so vielte Primzahl.

5.1.8 Große Primzahlen ermitteln

Große Primzahlen sind die Voraussetzung für sichere Verschlüsselungsverfahren. MuPAD stellt sie mit Hilfe der Funktion `nextprime()` zur Verfügung, zB:

```
>> nextprime(2^500);

3273390607896141870013189696827599152216642046043064789483291368096133796 4\
0467455488327009232590415715088668412756007100921725654588539305332852758 9\
431
>> isprime(%);

                                  TRUE
>>
```

Ob eine Primzahl vorliegt kann mit der Funktion `isprime()` (oder durch Faktorisieren) überprüft werden.

Braucht man zwei ungefähr gleich große, aber verschiedene Primzahlen, so kann man eine relativ kleine Zahl zum Argument von `nextprime()` addieren:

```
>> a := nextprime(2^500);

3273390607896141870013189696827599152216642046043064789483291368096133796 4\
0467455488327009232590415715088668412756007100921725654588539305332852758 9\
431
>> b := nextprime(2^500 + 2^16);

3273390607896141870013189696827599152216642046043064789483291368096133796 4\
0467455488327009232590415715088668412756007100921725654588539305332852765 5\
079
>> b - a;

                                  65648
>> factor(b-a);

                                 4
                                2  11 373
>>
```

Beachte, dass die Differenz zwischen beiden Primzahlen ungefähr der addierten Zahl (2^16) entspricht. Die Differenz der beiden Primzahlen ist keine Primzahl...

5.1.9 Zufallszahlen

Zufallszahlen stellen Zahlen (aus einem bestimmten Wertebereich) dar, die alle **mit der gleichen Wahrscheinlichkeit** auftreten, wenn man sie zufällig aus der Wertemenge auswählt. Die Funktion `random(a..b)` stellt Zufallszahlen zwischen a und b zur Verfügung:

```
>> zz := random(0..1);

                          proc random() ... end
>> zz();

                                    0
>> zz() $ i = 1..10;

                      1, 0, 1, 1, 1, 0, 0, 0, 1, 1
>> random(0..1)();

                                    1
>>
```

Wie Sie sehen, liegen die möglichen Werte zwischen 0 und 1. Gibt man eine Liste von zB 10 Zufallszahlen aus, treten “etwa 5 Mal” die Zahlen 0 und 1 auf (im Beispiel: 4 mal 0 und 6 mal 1). Beachten Sie bitte, wie sich mit MuPAD Listen erzeugen lassen.

Benötigt man eine Zufallszahl, die zu einer bestimmten Zahl N teilerfremd ist, verwendet man eine kurze Programmschleife:

```
>> N:=nextprime(10^200);

100000000000000000000000000000000000000000000000000000000000000000000000000000\
000000000000000000000000000000000000000000000000000000000000000000000000000000\
000000000000000000000000000000000000000000000000000357
>> while (igcd(zz,N) > 1) do
     zz := random(1..N)()
   end_while:
   zz;

600993746759829337668454735094736764707883422813387791917924959003937512 09\
539300628363443011313746086538005862664913074813656220643842443844131905 75\
456567207535839113553710879599163815547445261008998 1
>> gcd(nextprim(10^200),%);

                                    1
>>
```

Die Funktion `igcd(a,b)` liefert den größten gemeinsamen Teiler zweier ganzer Zahlen a und b, die Funktion `gcd(a,b)` liefert den größten gemeinsamen Teiler der beiden Polynome a und b; gilt `gcd(a,b) = 1` heißen a und b **teilerfremd**.

Alternativ kann man zu einer gegebenen Zahl N teilerfremde Zufallszahlen mit einer kurzen Repeat-Schleife suchen:

```
>> N:=nextprime(10^200);

100000000000000000000000000000000000000000000000000000000000000000000000000000\
000000000000000000000000000000000000000000000000000000000000000000000000000000\
000000000000000000000000000000000000000000000000000357
>> repeat
      zz := random(1..N)()
   until gcd(zz,N)=1 end_repeat:
   zz;

672313605025423083821990536755928252407886139918985672771168817937493408 07\
728335795394301261629479870548736450984003401594705923178314906195914825 13\
697328131486228945410074523776903441005708070070045950 4
>>
```

5.1.10 Aufgaben

1. Lesen Sie in den MuPAD-Hilfe-Texten über `gcd()`, `igcd()`, `isprime()`, `ithprime()`, `nextprime()`, etc. nach und führen Sie die angegebenen Beispiele aus!
2. Beschreiben Sie den Unterschied der MuPAD-Funktionen `factor()` und `isprime()` und testen Sie deren Effizienz bei großen Zahlen!
3. Recherchieren Sie, was *Mersennesche Primzahlen* sind und testen Sie solche mit Hilfe der MuPAD-Funktionen!,
4. Verfassen Sie kurze MuPAD-Programme: Ausgabe von Primzahlen, Ausgabe von Primzahlpotenzen, ...

In den folgenden Abschnitten behandeln wir verschiedene kryptographische Verfahren. Ihre Algorithmen werden kurz vorgestellt und anschließend mit MuPAD realisiert, was vor allem bei den Verfahren eine bedeutende Rolle spielt, die auf modularen Primzahlpotenzen beruhen.

5.2 Der Caesar-Code

Der schon im Altertum verwendete Code zum Verschlüsseln von Texten besteht darin, alle Buchstaben eines Klartextes durch einen anderen Buchstaben zu ersetzen. Das dabei verwendete

Bildungsgesetz ist denkbar einfach: Jeder Buchstabe wird durch einen Buchstaben ersetzt, der eine gleichbleibende Anzahl von Schritten im Alphabet weiter hinten steht. Ist das Ende des Alphabets erreicht, so zählt man zyklisch über “Z” und “A” wieder von vorne durch ...

Der Algorithmus ist durch eine einfache Addition zum Modul 26 (wenn beispielsweise nur Großbuchstaben im Klartext vorkommen) gegeben:

$y \equiv x + N \; mod \; m$

Dabei gilt $m = 26$. Bei dieser so genannten “Parallelverschiebung” findet man den Schlüssel N sehr leicht durch Probieren: Im Durchschnitt sind lediglich 13 Versuche notwendig, bis der Schlüssel gefunden wurde.

5.2.1 Verschlüsseln von Zeichenketten mit Caesar

Im folgenden kleinen MuPAD-Programm wird eine Zeichenkette `text`, die aus Großbuchstaben besteht, durch zyklisches Verschieben der Buchstaben um `s` Stellen im Alphabet “verschlüsselt”. Zunächst liefert die Funktion `numlib::toAscii` eine Liste der entsprechenden ASCII-Zahlen; sie liegen fr Großbuchstaben zwischen 65 und 90. Diese Liste wird in der Variablen `atext` (“Ascii-Text”) gespeichert. Die Liste `actext` enthöht die ASCII-Indizes der Verschlüsselung (“Ascii-Chiffre-Text”) - sie und die Variable `interim` werden mit Hilfe des Null-Operators initialisiert.

Die Funktion `nops()` liefert die Anzahl der Operanden einer Liste. Mit ihr kann die for-Schleife zur Verschiebung aller ASCII-Indizes realisiert werden: Liegt der Index zwischen 64 und 91 (wurde also ein Großbuchstabe eingegeben), so wird zunächst der ASCII-Index um 65 verringert, der Schlüssel `s` addiert und nach der Modulo-Operation mit 26 wieder mit 65 addiert. Das Ergebnis liefert den Eintrag für `actext[i]`. Alle Einträge zusammen liefern eine Liste, die mit Hilfe der eckigen Klammern als Array an die Funktion `numlib::fromAscii([actext])` übergeben werden.

```
>> caesar := proc(text, s)
     local atext, actext, ctext, interim;
   begin
     atext := numlib::toAscii(text);
     actext := null();
     interim := null();
     for i from 1 to nops(atext) do
       actext := (actext, 0);
     end_for;
     for i from 1 to nops(atext) do
       interim := atext[i];
       if interim > 64 then
         if interim < 91 then
           interim := 65 + ((interim - 65 + s) mod 26);
         end_if;
       end_if;
       actext[i] := interim;
     end_for;
     numlib::fromAscii([actext]);
   end_proc;

                        proc caesar(text, s) ... end
>> caesar("HALLO",2);

                                    "JCNNQ"
>> caesar("HALLO",1);

                                    "IBMMP"
>> caesar("IBMMP", -1);

                                    "HALLO"
```

5.2.2 Entschlüsseln von Zeichenketten nach dem Caesar-Verfahren

Wie man sich leicht überlegen kann, läuft die Entschlsselung darauf hinaus, den richtigen Schlüssel `s` mit entgegengesetztem Vorzeichen einzugeben. Schreiben Sie eine kleine Prozedur, die auf

einen eingegebenen Schlüsseltext alle Schlüssel von 1 bis 26 anwendet und das Dechiffrat ausgibt!

Lösung:

```
>> caesar_ex:=proc(Buchstabe, Schluessel)
     local interim;
   begin
     interim:=Buchstabe:
     if interim>64 then
       if interim < 91 then
         interim:=65+((interim-65+Schluessel) mod 26):
       end_if:
     end_if:
     if interim > 96 then
       if interim < 123 then
         interim := 97 + ((interim - 97 + Schluessel) mod 26):
       end_if:
     end_if:
     interim:
   end_proc:
>> attack := proc(chiffre)
     local interim, Schluessel;
   begin
     for Schluessel from 1 to 25 do
       interim := numlib::fromAscii(map(numlib::toAscii(chiffre),caesar_ex,-Schluessel)):
       print (Unquoted, "Mit Schlssel " . Schluessel . ": " . interim):
     end_for:
   end_proc:
>> attack("Zewfidrkzb zjk jtyfve leu Cvyivi clvxve eztyk");

     Mit Schlssel 1: Ydvehcqjya yij isxeud kdt Buxhuh bkuwud dysxj

     Mit Schlssel 2: Xcudgbpixz xhi hrwdtc jcs Atwgtg ajtvtc cxrwi

     Mit Schlssel 3: Wbtcfaohwy wgh gqvcsb ibr Zsvfsf zisusb bwqvh

     Mit Schlssel 4: Vasbezngvx vfg fpubra haq Yruere yhrtra avpug

     Mit Schlssel 5: Uzradymfuw uef eotaqz gzp Xqtdqd xgqsqz zuotf

     Mit Schlssel 6: Tyqzcxletv tde dnszpy fyo Wpscpc wfprpy ytnse

     Mit Schlssel 7: Sxpybwkdsu scd cmryox exn Vorbob veoqox xsmrd

     Mit Schlssel 8: Rwoxavjcrt rbc blqxnw dwm Unqana udnpnw wrlqc

     Mit Schlssel 9: Qvnwzuibqs qab akpwmv cvl Tmpzmz tcmomv vqkpb

     Mit Schlssel 10: Pumvythapr pza zjovlu buk Sloyly sblnlu upjoa

     Mit Schlssel 11: Otluxsgzoq oyz yinukt atj Rknxkx rakmkt toinz

     Mit Schlssel 12: Nsktwrfynp nxy xhmtjs zsi Qjmwjw qzjljs snhmy

     Mit Schlssel 13: Mrjsvqexmo mwx wglsir yrh Pilviv pyikir rmglx

     Mit Schlssel 14: Lqirupdwln lvw vfkrhq xqg Ohkuhu oxhjhq qlfkw

     Mit Schlssel 15: Kphqtocvkm kuv uejqgp wpf Ngjtgt nwgigp pkejv

     Mit Schlssel 16: Jogpsnbujl jtu tdipfo voe Mfisfs mvfhfo ojdiu

     Mit Schlssel 17: Informatik ist schoen und Lehrer luegen nicht

     Mit Schlssel 18: Hmenqlzshj hrs rbgndm tmc Kdgqdq ktdfdm mhbgs

     Mit Schlssel 19: Gldmpkyrgi gqr qafmcl slb Jcfpcp jscecl lgafr

     Mit Schlssel 20: Fkclojxqfh fpq pzelbk rka Ibeobo irbdbk kfzeq

     Mit Schlssel 21: Ejbkniwpeg eop oydkaj qjz Hadnan hqacaj jeydp
```

```
     Mit Schlssel 22: Diajmhvodf dno nxcjzi piy Gzcmzm gpzbzi idxco

     Mit Schlssel 23: Chzilgunce cmn mwbiyh ohx Fyblyl foyayh hcwbn

     Mit Schlssel 24: Bgyhkftmbd blm lvahxg ngw Exakxk enxzxg gbvam

     Mit Schlssel 25: Afxgjeslac akl kuzgwf mfv Dwzjwj dmwywf fauzl

>>
```

Nach wenigen Probeläufen hat man den “geheimen” Schlüssel geknackt; wie man leicht nachprüfen kann, sind bei 26 verschiedenen Schlüsseln im Durchschnitt 13 Suchläufe notwendig, um den Schlüssel zu knacken. Damit hat die Caesar-Chiffre jedenfalls keinerlei Bedeutung für die Verschlüsselung von Nachrichten ...

5.3 Der Vigenère-Code

Die Vigenère -Verschlüsselung (nach Blaise de Vigenère) galt lange als sicherer Chiffrieralgorithmus. Ein Schlüsselwort bestimmt, wie viele Alphabete genutzt werden. Die Alphabete leiten sich aus der Caesar-Substitution ab.

Dem britischen Mathematiker Charles Babbage gelang um das Jahr 1854 erstmals die Entschlüsselung einer Vigenère-Chiffre. Diese Entdeckung wurde jedoch damals nicht öffentlich bekannt gemacht. Der preußische Offizier Friedrich Kasiski veröffentlichte im Jahr 1863 seine Lösung und ging damit in die Geschichte ein.

Polyalphabetische Ersetzungschiffren bezeichnen in der Kryptographie Formen der Textverschlüsselung, bei der einem Buchstaben/Zeichen jeweils ein anderer Buchstabe/Zeichen zugeordnet wird. Im Gegensatz zur monoalphabetischen Substitution werden für die Zeichen des Klartextes mehrere Geheimtextalphabete verwendet.

5.3.1 Die Vigenère - Verschlüsselung an einem Beispiel

Text: GEHEIMNIS
Schlüssel: AKEYAKEYA
Cyphertext: HPMDJXSHT

- Das Schlüsselwort sei *AKEY*, der Text *GEHEIMNIS*.
- Vier Caesar-Substitutionen verschlüsseln den Text.
- Die erste Substitution ist eine Caesar-Verschlüsselung mit dem Schlüssel *A*. *A* ist der erste Buchstabe im Alphabet. Er verschiebt den ersten Buchstaben des zu verschlüsselnden Textes, *G* um 1 Stelle, es wird zum *H*.
- Der zweite Buchstabe des Schlüssels, das *K*, ist der 11. Buchstabe im Alphabet, er verschiebt das zweite Zeichen des Textes, das *E* um 11 Zeichen. Aus *E* wird ein *P*.
- Das dritte Zeichen des Schlüssels *E* verschiebt um 5, *Y* um 25 Stellen.
- Die Verschiebung des nächsten Buchstabens des Textes beginnt wieder bei *A*, dem ersten Buchstaben des Schlüssels

Verwende die folgende Tabelle zum “Handverschlüsseln” von Nachrichten nach Vigenère:

													T	e	x	t										
		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	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
	B	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
	C	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	B
	D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
	E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
	F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
S	G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
c	H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
h	I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
l	J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
ü	K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
s	L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
s	M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
e	N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
l	O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
	P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
	Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
	R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
	S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
	T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
	U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
	V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
	W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
	X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
	Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
	Z	Z	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

Die ***Autokey***-**Vigenère**-Verschlüsselung vermeidet die Periodizität des Schlüsselwortes. Stattdessen wird an den Schlüssel der Klartext angehängt:

Text: GEHEIMNIS
Schlüsselwort: AKEY
Schlüssel: AKEYGEHEI
Cyphertext: HPMDPRVNB

Der Spezialfall, dass der Schlüssel genauso lang ist wie der zu verschlüsselnde Text, heißt ***Vernam-Verschlüsselung*** oder **One-Time-Pad**-Verschlüsselung.

Bei der Vigenère-Verschlüsselung bestimmt das Schlüsselwort die Zahl und Auswahl der Chiffrier-Alphabete. Gleiches leisten Walzen oder Räder, auf die die Buchstaben des Alphabets eingraviert sind. Richtig zueinander orientiert, liest man an ihnen unmittelbar den chiffrierten Text ab. Kommt man überein, bei jedem Buchstaben die Stellung der Walzen zueinander zu verändern, lässt sich die Zahl der zur Verfügung stehenden Alphabete um ein Vielfaches erhöhen. Die **Enigma** war eine elektro-mechanische Verschlüsselungsmaschine, die im Zweiten Weltkrieg im Funkverkehr des deutschen Militärs verwendet wurde. Das Wort *Enigma* kommt aus dem Griechischen und bedeutet Rätsel.

5.3.2 MuPAD

Die Methode `ver(Buchstabe, Schluessel)` transformiert analog zum Caesar-Verfahren des ASCII-Wert eines Klartextbuchstaben in den ASCII-Wert des Geheimtextes. Die Methode `krypt(text,swort)` wendet die Methode `ver` Buchstabe für Buchstabe entsprechend dem Vigenère-Verfahren an: Bei jeder Transformation wird (zyklisch) der nächste Buchstabe des Schlüsselwortes zum Verschlüsseln verwendet:

```
>> ver:=proc(Buchstabe, Schluessel)
   local interim;
   begin
   interim:=Buchstabe:
   if interim > 64 then
   if interim < 91 then
   interim := 65 + ((interim - 65 + Schluessel) mod 26):
   end_if:
   end_if:
```

```
   if interim > 96 then
   if interim < 123 then
   interim := 97 + ((interim - 97 + Schluessel) mod 26):
   end_if:
   end_if:
   return (interim):
   end_proc:
>> krypt:=proc(text,swort)
   local pos,interim, swortlength, atext, actext;
   begin
     swortlength:=length(swort):
     atext:=numlib::toAscii(text);
     interim:=null();
     actext:=null();
     pos:=0;
     for i from 1 to nops(atext) do
       actext := (actext, 0);
     end_for;
     for i from 1 to nops(atext) do
       interim := atext[i];
       interim:=ver(interim,numlib::toAscii(swort)[pos+1]-64):
       pos:=(pos + 1) mod swortlength:
       actext[i]:=interim:
     end_for:
     numlib::fromAscii([actext]);
  end_proc;

                      proc krypt(text, swort) ... end
>> krypt("GEHEIMNIS","AKEY");

                                "HPMDJXSHT"
```

5.3.3 Aufgaben

1. Verfasse eine passende MuPAD-Routine, mit der eine Vigenère-Chiffre (bei bekanntem Schlüsselwort) entschlüsselt werden kann!

2. Recherchiere zu Enigma (Tipp: Ein ausführlicher Bericht findet sich beispielsweise bei WikiPedia)!

3. Beschreibe Gemeinsamkeiten und Unterschiede zwischen dem Caesar- und dem Vigenère - Verfahren!

4. Beurteile, warum das Autokey-Vigenère - Verfahren sicherer ist als das Vigenère - Verfahren!

5. Was macht deiner Meinung nach das One-Time-Pad - Verfahren so sicher (Tipp: Wikipedia)?

5.4 Das Hill-Verfahren

Beim Hill-Verfahren versucht man dem Nachteil der Vigenère-Verschlüsselung zu entgegnen. Im Gegensatz zum Vigenère-Verfahren, bei dem einzelne Zeichen ersetzt werden, wendet man beim Hill-Verfahren eine Transposition gleich auf mehrere Zeichen (auf einen ganzen Block) an. Die Länge dieses Blocks, m, muss kleiner oder gleich der gesamten Textlänge, n sein.

Die Verschlüsselung entspricht folgender Matrizenoperation:

$E_m(p, s) \ = \ (A \cdot p \ + \ b) \ mod \ k \qquad s = (A, b)$

Der Schlüssel s besteht also aus der Angabe einer Matrix A und aus einem m-dimensionalen Vektor b. Soll ein Text in Blöcken mit der Blocklänge $m = 5$ verschlüsselt werden, muss der Vektor b m Komponenten aufweisen; und die Matrix A muss eine (5×5) Matrix sein.

5.4.1 Einen Block mit dem Hill-Verfahren verschlüsseln

Die folgende Methode verschlüsselt einen 5 Zeichen langen Text:

```
>> hill:=proc(text)
     local atext, actext, ctext, A, B, P;
   begin
     A:=matrix([[3,5,1,0,8],[2,7,18,1,2],[13,24,6,11,20],[21,25,17,13,19],[9,3,11,4,5]]);
     B:=matrix([[0],[1],[3],[2],[1]]);
     atext:=numlib::toAscii(text);
     P:=matrix([[atext[i]-64] $ i = 1..5]);
     actext:=(A*P+B);
     for i from 1 to nops(actext) do
       actext[i] := actext[i] mod 26;
     end_for;
     ctext:=numlib::fromAscii([actext[i]+64 $ i = 1..5]);
     ctext;
   end_proc;

                           proc hill(text) ... end
>> hill("KRYPT");

                                  "VEDFU"
```

5.4.2 Entschlüsseln eines Blockes nach dem Hill-Verfahren

Die folgende Methode entschlüsselt einen 5 Zeichen langen Text:

```
>> dehill:=proc(ctext)
     local text, atext, actext, A, B, P;
    begin
      A:=matrix([[3,5,1,0,8],[2,7,18,1,2],[13,24,6,11,20],[21,25,17,13,19],[9,3,11,4,5]]);
      B:=matrix([[0],[1],[3],[2],[1]]);
      actext:=numlib::toAscii(ctext);
      P:=matrix([[actext[i]-64] $ i=1..5]);
      atext:=1/A*(P-B);
      text:=numlib::fromAscii([(atext[i] mod 26)+64 $ i = 1..5]);
      text;
   end_proc;

                         proc dehill(ctext) ... end
>> dehill("VEDFU");

                                  "KRYPT"
```

5.4.3 Aufgaben

1. Beschreibe das Hill-Verfahren für beliebig lange Texte!
2. Formuliere MuPAD-Methoden für das Verschlüsseln und Entschlüsseln beliebig langer Texte nach dem Hill-Verfahren!

5.4.4 Lösung: Hill-Verfahren für beliebig lange Texte

Wir unterteilen die Nachricht in Blöcke zu je 5 Zeichen und füllen den letzten, unvollständigen Block mit Leerzeichen auf. Auf jeden Block wenden wir dann das Hill-Verfahren an und verketten die erhaltenenen Chiffretexte:

```
>> hill:=proc(text)
       local atext, actext, ctexti, ctext, A, B, P, textlaenge;
     begin
       ctext:="";
       A:=matrix([[3,5,1,0,8],[2,7,18,1,2],[13,24,6,11,20],[21,25,17,13,19],[9,3,11,4,5]]);
```

```
      B:=matrix([[0],[1],[3],[2],[1]]);
      textlaenge:=length(text);
      if textlaenge mod 5 > 0 then
text:=_concat(text," " $ i = 0..(5-(textlaenge mod 5)));
      end_if;
      textlaenge:=length(text);
      atext:=numlib::toAscii(text);
      for j from 1 to ((textlaenge div 5)) do
P:=matrix([[atext[i+(j-1)*5]-64] $ i = 1..5]);
actext:=(A*P+B);
for i from 1 to nops(actext) do
  actext[i] := actext[i] mod 27;
end_for;
ctexti:=numlib::fromAscii([actext[i]+64 $ i = 1..5]);
ctext:=_concat(ctext,ctexti);
      end_for;
      ctext;
    end_proc;
```

proc hill(text) ... end

```
>> dehill:=proc(ctext)
      local text, atext, actext, A, B, P, itext, textlaenge;
    begin
      text:="";
      textlaenge:=length(ctext);
      A:=matrix([[3,5,1,0,8],[2,7,18,1,2],[13,24,6,11,20],[21,25,17,13,19],[9,3,11,4,5]]);
      B:=matrix([[0],[1],[3],[2],[1]]);
      actext:=numlib::toAscii(ctext);
      for j from 1 to (textlaenge div 5) do
P:=matrix([[actext[i +(j-1)*5]-64] $ i=1..5]);
atext:=1/A*(P-B);
texti:=numlib::fromAscii([(atext[i] mod 27)+64 $ i = 1..5]);
text:=_concat(text,texti);
      end_for;
      text;
    end_proc;
```

proc dehill(ctext) ... end

```
>> hill("TALENTEZENTRUM");
```

"@@SJAGJJXEWH@GG"

```
>> dehill(%);
```

"TALENTEZENTRUMV"

```
>> hill("ALFRED");
```

```
                                "AGQYTWDTMC"

>> dehill(%);

                                "ALFREDVVVV"
```

5.5 Diffie-Hellman

Ein besonderes Anliegen in der Kryptographie ist das Übertragen eines Schlüssels von einem Absender zu einem Empfänger. Dabei findet die Kommunikation zwischen zwei unbekannten Kommunikationspartnern A und B statt. Mit dem Algorithmus von Diffie-Hellman wird ein Schlüssel auf sicherem Weg generiert - d.h. Angreifer können zwar alle Nachrichten, auch die zur Generierung des Schlssels notwendigen Informationen, abhören, aber dennoch den Schlüssel mit großer Wahrscheinlichkeit dabei nicht berechnen (“knacken”).

Um den Vorgang der Nachrichtenbermittelung anschaulich beschreiben zu können, hat sich in der Literatur zur Kryptographie eingebürgert, den Absender einer Nachricht mit “A” (“Alice”) und den Empfänger mit “B” (“Bob”) zu bezeichnen. Eine mögliche Angreiferin trägt den Namen “Eve” ...

5.5.1 Grundlagen

Die Grundlage bilden eine (große) Primzahl p und ein Modul g, der eine Restklasse mit möglichst großer Zykluslänge beschreibt. Weiters ermitteln die beiden Kommunikationstner A (Alice) und B (Bob) jeweils Zufallszahlen a und b, die kleiner als $p-1$ sind. Die Zahl a steht nur Partner A zur Verfgung (und ist beispielsweise auf dem Rechnersystem von Alice gespeichert), b ist nur Partner Bob zur Verfügung.

Als erstes bestimmt Alice den Ausdruck $\alpha \equiv g^a \, mod \, p$ und übermittelt α an Partner Bob.

Partner Bob bestimmt den Ausdruck $\beta \equiv g^b \, mod \, p$ und übermittelt β an Partner Alice.

Mit den bekannten Parametern p und g können nun beide Partner folgende Ausdrücke berechnen: Alice berechnet mit β den Ausdruck β^a (Alice ist ja im Besitz seiner “Geheimzahl” a); Bob berechnet mit α den Ausdruck α^b (Bob besitzt ja b). Beide erhalten wegen der Kommutativitätï¿½ der (diskreten) Exponentiation den gleichen Wert, den sie als Schlüssel verwenden: $\beta^a \equiv (g^b)^a \, mod \, p$ bzw. $\alpha^b \equiv (g^a)^b \, mod \, p$.

Die Frage ist nur: Ist der auf diese Weise berechnete Schlüssel sicher - d.h. kann ihn ein Angreifer mit den bekannten Informationen (p, g, α, β) berechnen? Bei einem genügend großen “Spielraum” für die Zufallszahlen a und b (also bei einer großen Primzahl p) ist es für Angreifer de facto nicht möglich, a oder b mit Hilfe des diskreten Logarithmus zu bestimmen.

5.5.2 MuPAD - Beispiel

Grundsätzlich müssen wir eine Primzahl p und eine Basis g von Z_p finden. Diese Gruppe hat die Ordung $p-1$. Hat ein Element die Ordnung $p-1$, dann erzeugt dieses Element die Gruppe Z_p. Das Problem besteht nur darin, zu einer (großen) Primzahl p dieses Element g zu finden.

Beispiel Z_7:

Die Restklasse Z_7 besteht aus den Elementen $Z_7 = \{0, 1, 2, 3, 4, 5, 6\}$. Was sind erzeugende Elemente von Z_7? Wir versuchen es der Reihe nach mit den Zahlen 2 .. 6:

$2^1 = 2, 2^2 = 4, 2^3 = 8 \equiv 1, 2^4 = 16 \equiv 2, 2^5 = 32 \equiv 4, 2^6 = 64 \equiv 1, ...$

2 ist sicher kein erzeugendes Element.

$3^1 = 3, 3^2 = 9 \equiv 2, 3^3 = 27 \equiv 6, 3^4 = 3^2 \cdot 3^2 \equiv 2 \cdot 2 = 4, 3^5 = 3^3 \cdot 3^2 \equiv 6 \cdot 2 = 12 \equiv 4 = 5, 3^6 = 3^3 \cdot 3^3 \equiv 6 \cdot 6 = 36 \equiv 1$

3 ist also ein erzeugendes Element von Z_7.

Eine Restklasse kann natürlich auch mehrere erzeugende Elemente haben; die Frage ist: Wie findet man erzeugende Elemente am besten? Dazu betrachten wir das letzte Beispiel: Um zu testen ob irgendein x aus Z_7 ein erzeugendes Element der Gruppe ist, faktorisiert man $p-1$, also in unserem Fall die Zahl $6 = 2 \cdot 3$. Sind alle Potenzen $x^{\frac{p-1}{q}}$ ungleich 1 (für jeden Primfaktor q von $p-1$, zB 2 und 3), dann ist x ein erzeugendes Element.

```
>> powermod(2,i,7) $ i = 1 ..6;
                           2, 4, 1, 2, 4, 1
>> powermod(3,i,7) $ i = 1 ..6;
                           3, 2, 6, 4, 5, 1
>> powermod(4,i,7) $ i = 1 ..6;
                           4, 2, 1, 4, 2, 1
>> powermod(5,i,7) $ i = 1 ..6;
                           5, 4, 6, 2, 3, 1
>> powermod(6,i,7) $ i = 1 ..6;
                           6, 1, 6, 1, 6, 1
```

Direktes Durchrechnen liefert somit die erzeugenden Elemente für Z_7: 3 und 5. Um erzeugende Elemente zu finden sucht man aber allgemein zuerst die Primfaktoren von $p-1$, in unserem Fall also die beiden Zahlen 2 und 3. Dann testet man beliebige Zahlen aus Z_7:

```
>> powermod(2,6/2,7);
                                  1
>> powermod(2,6/3,7);
                                  4
>> powermod(3,6/2,7);
                                  6
>> powermod(3,6/3,7);
                                  2
>>
```

2 ist also kein erzeugendes Element, 3 hingegen schon.

MuPAD stellt dazu die Funktion `numlib::primroot` zur Verfgung:

```
>> numlib::primroot(7);
                                  3
```

Bedauerlicherweise arbeitet diese Funktion für große Primzahlen nicht mehr. In diesem Fall geht man folgendermaßen vor:

Das Programm `kleinePrim(`n`)` ermittelt das Produkt aller Prinzahlen bis zu einem bestimmten Index n. Das Programm `entferneFaktoren(`p`,`n`)` dividiert die Zahl $p-1$ durch das Produkt der kleinen Primzahlen - so lange die Zahlen `grFaktor` und `kleinePrim(`n`)` nicht teilerfremd sind, wird die Zahl `grFaktor` durch die gemeinsamen Teiler dividiert. Am Ende bleiben das Produkt der Primzahlen, durch die dividiert werden konnte und das erzeugende Element `ProdFaktoren`.

```
>> kleinePrim := proc(n)
   begin
     _mult(ithprime(i) $ i=1..n):
   end_proc:
>> entferneFaktoren := proc(p,n)
     local grFaktor, gemFaktor, ProdFaktoren;
   begin
     grFaktor := p-1:
     gemFaktor := igcd(kleinePrim(n), grFaktor):
     while (gemFaktor > 1) do
       grFaktor := grFaktor / gemFaktor:
       gemFaktor := igcd(kleinePrim(n), grFaktor):
     end_while:
     ProdFaktoren := (p-1) / grFaktor:
     return(grFaktor, ProdFaktoren):
   end_proc
```

Sobald die Primzahl p und die Basis g bestimmt wurde, können wir den Algorithmus von Diffie-Hellman anwenden:

```
>> p:=nextprime(random(2^500..2^600)());

237393209906714456688634703968559146744115366748431067452900259702663406 46\
583892894647918203579585316015873186363698527838209187146246601817406327 03\
4645513233123652828546596901886 73
>> Zahlen := entferneFaktoren(p,300);

618211484132068897626652874918122777979467100907372571491927759642352621 00\
478887746478953655155170093791336422822131582912003091526683858899495643 31\
938935737792617924100676276533

   , 384
>> x:=random(1..p)();

177941280757791884450975314571192877052399951727462917984557293125511094 74\
086251111022026275356611921035677465270245044795760683227263344743964274 78\
329176009366885457828819667754 8
>> g:=powermod(x,Zahlen[2],p);

106256600267563049773150217360632804854769673263764949142069240593867238 13\
535328801834306431000432556482620438791591823116768444270433161269897064 59\
007767601096050450779921114632916
>> powermod(g,Zahlen[1],p);
                                  1
```

Zu beachten ist, dass x eine Zufallszahl ist, die mit dem Produkt der entfernten Primzahlen potenziert wird. Sie hat eine Ordnung, die den großen Faktor teilt. Daher muss für die Potenz $g^{grFaktor} \equiv 1 \; mod \; p$ gelten.

Für den Algorithmus von Diffie-Hellman sind außer dem Paar (p,g) zwei Zufallszahlen a und b für Alice und Bob erforderlich. Diese Zufallszahlen sind geheim (haben Alice und Bob sie auf ihrem System gespeichert, brauchen sie sie auch nicht zu kennen). Alice und Bob bestimmen jetzt jeweils ihren öffentlich bekannten Teil des Schlüssels, `Alpha` und `Beta`.

```
>> a := random(2..p-2)();

565043770767103196688881671340569890487059925752176060716120475462379589 63\
341786626967382432353747839584597769522934600666137439353628583711509506 57\
836354419500565996237528539605 25
>> b := random(2..p-2)();

515428544309096035453876100308311226573659157451893295709098216343858987 92\
395750941883746059056488359846678927375779995264122658926158747442597332 34\
547245454130893005296301394758 58
>> Alpha := powermod(g,a,p);

224778134700506563747472371302152044647392930866144964186381035165392013 04\
320710710859686899825908508680706493066439450216348218313203237676113757 41\
170124559792813976583739049949796
>> Beta := powermod(g,b,p);

228675102652823543165909203671221900124958164138772840922202307565406934 86\
228671943796640505076090973162848012612405879991625295213188912619138769 21\
230790804370891903686600342854401
>> KAlice := powermod(Beta,a,p);

201565931508216784122235226666702649899246202510585027769372858951364351 06\
694411006430766540955600798835902518769577778767023191519260445503531140 77\
901923870695494083620046961972364
>> KBob := powermod(Alpha,b,p);

201565931508216784122235226666702649899246202510585027769372858951364351 06\
694411006430766540955600798835902518769577778767023191519260445503531140 77\
901923870695494083620046961972364
>> is(KAlice=KBob);

                                TRUE
```

Dabei bezeichnet `KAlice` den Schlüssel, den Alice mit Hilfe der öffentlich bekannten Informationen p und `Beta` und mit ihrer geheimen Zufallszahl a berechnet. `KBob` ist der Schlüssel, den Bob mit Hilfe der öffentlich bekannten Informationen p und `Alpha` und mit seiner geheimen Zufallszahl b berechnet. In der letzten Zeile wird berprüft, ob die beiden Schlüssel, die Alice und Bob bestimmt haben, identisch sind.

5.5.3 Texte verschlüsseln

Nachdem wir uns überlegt haben, wie man Schlüsselpaare nach Diffie-Hellman erzeugt, sollen nun beliebige Texte nach diesem Verfahren verschlüsselt werden.

```
>> text2zahl:=proc(text)
    local atext, laenge, zahl;
    begin
      atext := numlib::toAscii(text);
      laenge:=nops(atext);
      m:=0;
      (m := m + atext[laenge - i] * 10^(i*3)) $ i=0..laenge-1;
      return (m);
    end_proc;

                        proc text2zahl(text) ... end
>> zahl2text:=proc(zahl)
   local atext, laenge;
    begin
      atext:=[];
      repeat
        atext :=[modp(zahl,1000)].atext;
        zahl:=floor(zahl/1000);
        until zahl/1000 < 1 end_repeat;
      atext:=[zahl].atext;
      return(numlib::fromAscii(atext));
    end_proc;

                        proc zahl2text(zahl) ... end
>> p := 29: g := 27: // oeffentlich bekannt
>> a := random (2 .. p-2)(): anton := powermod(g,a,p);

                                     8
>> b := random (2..p-2)(): berta := powermod(g,b,p);

                                    13
>> ka := powermod(berta,a,p); // Schluessel fuer Anton

                                    22
>> kb := powermod(anton, b, p); // Schluessel fuer Berta

                                    22
>> // Die beiden Schluessen muessen gleich sein, ehklar
>> mText := "Informatik in Seitenstetten is cool";

                   "Informatik in Seitenstetten is cool"
>> text := "Informatik in Seitenstetten is cool":
>> ztext := text2zahl(text);

731101021111141090971161051070321051100320831011051161011101151161011161\16\
10111003210511503209911111110 8
>> f := (kk,x)->(kk*x):f(kk,x);

                                  kk x
>> ctext := f(ka, ztext);

160842224644451040013655431235470631242070582822431255422442253255422455\45\
54224420706312530706180444444376
>> f_inv:=(kk,x)->(x/kk):f_inv(kk,x);

                                    x
                                   --
                                   kk
>> m_z:=f_inv(kb, ctext);
```

```
731101021111141090971161051070321051100320831011051161011101151161011161161\
1011100321051150320991111111108
>> klar := zahl2text(m_z);

                  "Informatik in Seitenstetten is cool"
>>
```

5.6 Elgamal

Mit dem ElGamal-Algorithmus können Nachrichten verschlüsselt und wieder entschlüesselt, bzw. Nachrichten signiert werden. Der Algorithmus arbeitet mit Public-Key-Verschlüsselung.

5.6.1 Mathematische Grundlagen

Das Verfahren von Elgamal hat viele Gemeinsamkeiten mit dem Algorithmus von Diffie-Hellman. Wieder kommt es darauf an, ein Z_p mit einer großen Primzahl und eine Basis g zu finden. Diese Zahlen bestimmen wir auf die gleiche Weise wie beim Diffie-Hellman-Verfahren.

Sind für Alice und Bob die geheimen Schlüssel a und b und - zum Paar (p, g) passende öffentliche Schlüssel α = `Alpha` und β = `Beta` bestimmt, wird die Nachricht N verschlüsselt. Bob verfährt beim Verschlüsseln folgendermaßen:

$V \ = \ N \cdot \alpha^b \ mod \ p$

Alice kann das Chiffrat mit den - nur ihr - bekannten Schlüsseln so entschlüsseln:

$E \ = \ V \cdot \beta^{p-1-a} \ mod \ p$

Wir müssen jetzt zeigen, dass $E \equiv N \ mod \ p$ gilt; alle Gleichungen in der folgenden Rechnung werden modulo p behandelt:

$E \ = \ V \cdot \beta^{p-1-a}$

Wir setzen fr V ein und erhalten:

$E \ = \ (N \cdot \alpha^b) \cdot \beta^{p-1-a}$

Auf der Basis von g und p konnten die öffentlichen Schlüssel $\alpha \ = \ g^a$ und $\beta \ = \ g^b$ bestimmt werden. Damit erhalten wir:

$E \ = \ (N \cdot (g^a)^b) \cdot (g^b)^{p-1-a}$

Wir formen um und vereinfachen (alles modulo p!):

$E \ = \ N \cdot (g^a)^b) \cdot (g^b)^{-a} \cdot g^{b \cdot (p-1)}$

Damit erhalten wir:

$E \ = \ N \cdot (g^{p-1})^b$

Mit Hilfe des **Kleinen Fermat**, $x^{p-1} \equiv 1 \ mod \ p$, erhalten wir für die letzte Gleichung:

$E \ = \ N \cdot 1^b$

Somit haben wir das gewünschte Ergebnis:

$E \ = \ N$

5.6.2 MuPAD - Beispiel

```
>> // Al Gamal - Verfahren zum Verschlsseln
// von (langen) Texten
// (vgl. Diffie-Hellman)
//
kleinePrim := proc(n)
```

```
begin
  _mult(ithprime(i) $ i=1..n):
end_proc:
//
entferneFaktoren := proc(p,n)
  local grFaktor, gemFaktor, ProdFaktoren;
begin
  grFaktor := p-1:
  gemFaktor := igcd(kleinePrim(n), grFaktor):
  while (gemFaktor > 1) do
    grFaktor := grFaktor / gemFaktor:
    gemFaktor := igcd(kleinePrim(n), grFaktor):
  end_while:
  ProdFaktoren := (p-1) / grFaktor:
  return(grFaktor, ProdFaktoren):
end_proc:
//
// Suche nach einer Basis g ...
Basis := proc(p, n)
  local Ergebnis, x, g, test;
begin
Ergebnis:=entferneFaktoren(p, n):
x := random(1..p)():
g := powermod(x, Ergebnis[2], p):
test := powermod(g, Ergebnis[1], p):
while (test > 1) do
  x := random(1..p)():
  g := powermod(x, Ergebnis[2], p):
  test := powermod(g, Ergebnis[1], p):
end_while:
return(g):
end_proc:
```

Damit haben wir sicher eine große Primzahl p und eine Basis g für Z_p zur Verfügung. Wir wenden jetzt das ElGamal-Verfahren schrittweise an und verschlüsseln einen Text. Das Chiffrat soll anschließend wieder entschlüsselt werden...

```
>> p:=nextprime(random(2^300..2^400)());

489407833960544633640031733776877716277941615887674781141409075409121253 76\
899601177444254089578985222555938660493574678 1
>> g:=Basis(p,200);

112096830000404416898353630576743240218599786320142402320317459182626361 94\
042663672436741139330041123514783871396079444 8
>> a := random(1..p-2)();

146486307198155590763466429392673709525428510973272600608981219760099374 67\
598293376684547350947367647078834228133877919 2
>> Alpha := powermod(g,a,p);

309002577062377668276094946329690698226099251064077154440306738657011027 76\
858849557813069835963015990408138624596051686 7
>> b := random(1..p-2)();

303088066433206575899268894586565295035804470650331081523503999404534966 87\
484643206968936485877581984979900453060136201 6
>> Beta := powermod(g,b,p);

320746126532506091559851733800116961798735290593739645150583223869267438 85\
487745655140867877501149770207022701957833997 6
```

Damit können wir einen Text verschlüsseln. Zunächst weisen wir eine oder mehrere Zeichenketten der Variablen `text` zu. Mit der Methode `toAscii()` erhalten wir aus diesem Text ein Array von ganzen Zahlen. Auf jede dieser Zahl wird nun der Reihe nach die oben beschriebene Verschlüsselung durch Berechnen der Formel angewendet:

```
>> text := "Informatik ist schoen und Lehrer luegen nicht" . "(sagt zumindest ein Lehrer)":
atext := numlib::toAscii(text):
actext := [(atext[i]*powermod(Alpha, b, p) mod p) $ i=1..nops(atext)]:
```

Der verschlüsselte Text besteht aus einer Reihe (großer) Zahlen - sie werden hier aus Platzgründen nicht angegeben. Wir testen hier, ob das Entschlüsselungsverfahren, angewendet auf die Zahlen des Arrays `actext` wieder den ursprünglichen Klartext ergibt:

```
>> adetext := [(actext[i]*powermod(Beta, p-1-a, p) mod p) $ i = 1..nops(actext)];
[73, 110, 102, 111, 114, 109, 97, 116, 105, 107, 32, 105, 115, 116, 32,
   115, 99, 104, 111, 101, 110, 32, 117, 110, 100, 32, 76, 101, 104, 114,
   101, 114, 32, 108, 117, 101, 103, 101, 110, 32, 110, 105, 99, 104, 116,
   40, 115, 97, 103, 116, 32, 122, 117, 109, 105, 110, 100, 101, 115, 116,
   32, 101, 105, 110, 32, 76, 101, 104, 114, 101, 114, 41]
>> detext := numlib::fromAscii(adetext);
"Informatik ist schoen und Lehrer luegen nicht(sagt zumindest ein Lehrer)"
>>
```

5.7 Digitale Unterschrift

Neben der Verschlüsselung von Nachrichten (Vertraulichkeit) ist es in der Praxis sehr wichtig, dass auch nachgewiesen werden kann, dass eine Nachricht von einer ganz bestimmten Person stammt (Authentizifizierung). Dies wird durch die so genannte digitale Unterschrift möglich.

5.7.1 Mathematische Grundlagen

Eine Nachricht soll mit dem im letzten Abschnitt beschriebenen ElGamal-Verfahren signiert werden. Dazu sind die Primzahl p, die Basis g, die geheimen und öffentlichen Schlssel a, b, α und β bekannt. Eine wesentliche Voraussetzung für den Signatur-Algorithmus ist nun das Bestimmen der so genannten Modulo-Inversen. Dies ist für eine Zahl r möglich, die zu $p-1$ teilerfremd ist.

Hat man eine zu $p-1$ teilerfremde Zufallszahl r (zwischen 1 und $p-1$) gefunden, bestimmt man $k = g^r \; mod \; p$. Mit Hilfe der Inversen r^{-1} bestimmen wir für eine Nachricht N den folgenden Ausdruck:

$$s = (N - a \cdot k) \cdot r^{-1} \; mod \; p-1$$

Die digitale Unterschrift ist nun durch das Paar (k, s) gegeben.

Sendet Alice (k, s) an Bob, so muss Bob überprüfen, ob $g^N = \alpha^k \cdot k^s \; mod \; p$ gilt (Bob kennt jedenfalls den öffentlichen Schlüssel α von Alice und das Paar (k, s)). Wir rechnen ($mod \; p$) nach:

$$\alpha^k \cdot k^s = (g^a)^k \cdot (g^r)^s = g^{a \cdot k} \cdot g^{r \cdot s} = g^{a \cdot k + r \cdot s}$$

Aus $s = (N - a \cdot k) \cdot r^{-1} \; mod \; p-1$ erhalten wir $s \cdot r = N - a \cdot k \; mod \; p-1$ und daraus $a \cdot k + r \cdot s = N \; mod \; p-1$. Diese Modulo-Gleichung kann in der folgenden Gleichung mit Hilfe der Unbekannten x so ausgedrückt werden:

$$a \cdot k + r \cdot s = x \cdot (p-1) + N$$

Damit können wir obige Umformung so fortsetzen:

$$g^{a \cdot k + r \cdot s} = g^{x \cdot (p-1) + N} = g^{x \cdot (p-1)} \cdot g^N$$

Wieder wenden wir den Kleinen Fermat an: $g^{x \cdot (p-1)} = 1 \; mod \; p$, sodass wir das gewünschte Resultat erhalten:

$$\alpha^k \cdot k^s = g^N$$

5.7.2 Durchführung mit MuPAD

Die Signatur einer Nachricht nach dem ElGamal-Verfahren setzt in MuPAD die gleichen Prozeduren wie im letzten Abschnitt voraus. Anschließend bestimmen wir das Paar (k, s) und wenden das Signieren auf eine Nachricht an.

```
>> // Al Gamal - Verfahren zum Verschluesseln
// von (langen) Texten
// (vgl. Diffie-Hellman)
//
kleinePrim := proc(n)
begin
  _mult(ithprime(i) $ i=1..n):
end_proc:
>> entferneFaktoren := proc(p,n)
  local grFaktor, gemFaktor, ProdFaktoren;
begin
  grFaktor := p-1:
  gemFaktor := igcd(kleinePrim(n), grFaktor):
  while (gemFaktor > 1) do
    grFaktor := grFaktor / gemFaktor:
    gemFaktor := igcd(kleinePrim(n), grFaktor):
  end_while:
  ProdFaktoren := (p-1) / grFaktor:
  return(grFaktor, ProdFaktoren):
end_proc:
>> // Suche nach einer Basis g ...
Basis := proc(p, n)
  local Ergebnis, x, g, test;
begin
Ergebnis:=entferneFaktoren(p, n):
x := random(1..p)():
g := powermod(x, Ergebnis[2], p):
test := powermod(g, Ergebnis[1], p):
while (test > 1) do
  x := random(1..p)():
  g := powermod(x, Ergebnis[2], p):
  test := powermod(g, Ergebnis[1], p):
end_while:
return(g):
end_proc:
```

Anschließend bestimmen wir eine (große) Primzahl p, die Basis g usf...

```
>> p:=nextprime(random(2^300..2^400)()):
>> g:=Basis(p,200):
>> a := random(1..p-2)():
>> b := random(1..p-2)():
>> Alpha := powermod(g,a,p):
>> Beta := powermod(g,b,p):
>> // Der Elgamal - Algorithmus eignet sich fuer digitale Unterschriften ...
// Wir suchen eine zu (p-1) teilerfremde Zahl r
r := random(1..p-1)():
while (igcd(r, p-1) > 1) do
  r := random(1..p-1)():
end_while:
r;

433599229456898724880795485531802023255050614524952922474293642065329619 15\
491266802685606943845068140764150696291779107 1
>> k := powermod(g,r,p);

799405365412121537241712806762147074178355723899723486132938726357086326 03\
136653242664278923066225404210795452625556250
>> invr := powermod(r,-1,p-1);

309428654429925845482196816259158124832465619558887056104499460731187214 59\
102173665028168657758456898374211215495590077 1
>> text := "informatiklehrer luegen nicht";

                      "informatiklehrer luegen nicht"
>> atext := numlib::toAscii(nachricht);

[105, 110, 102, 111, 114, 109, 97, 116, 105, 107, 108, 101, 104, 114, 101,

   114, 32, 108, 117, 101, 103, 101, 110, 32, 110, 105, 99, 104, 116]
>> s := [0 $ nops(atext)]:
for i from 1 to nops(atext) do
  s[i] :=(atext[i] - a*k) * invr mod p-1
end_for:
```

```
s:
>> ErgA := [powermod(g, atext[i],p) $ i = 1..nops(atext)]:
>> ErgB := [(powermod(Alpha,k,p) * powermod(k, s[i],p) mod p) $ i = 1..nops(s)]:
>> is(ErgA = ErgB);

                                    TRUE
```

Zu beachten ist, dass im Ausdruck `ErgB` die signierte Nachricht `s[i]` verarbeitet wird. Stimmen die Ausdrcke `ErgA` und `ErbB` berein, ist nachgewiesen, dass sie vom angegebenen Absender stammt.

5.8 RSA

RSA wurde 1977 von Ron **R**ivest, Adi **S**hamir und Leonard **A**dleman entwickelt. Das Verfahren verwendet große Primzahlen. Seine Sicherheit basiert auf der Schwierigkeit, große Zahlen zu faktorisieren. Das Verfahren ist einfach zu verstehen und ebenso leicht in Applikationen zu implementieren. Es hat sich als asymmetrisches Verschlüsselungsverfahren für zahlreiche Anwendungen etabliert.

5.8.1 Mathematische Grundlagen

Das Verfahren wurde 1977 entwickelt und basiert auf der Idee, dass die Faktorisierung einer großen Zahl, also ihre Zerlegung in (mindestens zwei) Faktoren, eine sehr aufwändige Angelegenheit ist, während das Erzeugen einer Zahl durch Multiplikation zweier Primzahlen trivial ist. Wenn nun eine Nachricht einem Empfänger verschlüsselt zugeleitet werden soll, generiert dieser einen öffentlichen Schlüssel. Der Nachrichtenabsender verwendet diesen öffentlich bekannt gemachten Schlüssel, indem er damit seine Botschaft verschlüsselt. Nur der Empfänger kann diese dekodieren, da nur er die Zusammensetzung des von ihm erzeugten (öffentlichen) Schlüssels kennt.

Funktionen wie die Multiplikation/Faktorisierung, bei denen eine Richtung leicht, die andere schwierig zu berechnen ist, bezeichnet man als ***Einwegfunktionen*** (engl. *one way function*). Um allerdings die Entschlüsselung tatsächlich möglich zu machen, muss es sich um ***Falltürfunktionen*** (engl. *trap door function*) handeln, die mit Hilfe einer Zusatzinformation auch rückwärts leicht zu berechnen sind.

5.8.2 Algorithmus

1. Wähle zufällig und stochastisch unabhängig zwei Primzahlen $p \neq q$, die etwa gleich lang sein sollten und berechne deren Produkt $N = p \cdot q$. In der Praxis werden diese Primzahlen durch Raten einer Zahl und darauf folgendes Anwenden eines Primzahltests bestimmt.

2. Berechne $\phi(N) = (p-1) \cdot (q-1)$, wobei ϕ für die Eulersche ϕ-Funktion steht.

3. Wähle eine Zahl $1 < e < \phi(N)$, die teilerfremd zu $\phi(N)$ ist.

4. Berechne die Zahl d so, dass das Produkt $e \cdot d$ kongruent 1 bezüglich des Modulus $\phi(N)$ ist, dass also $e \cdot d \equiv 1 (\mathrm{mod} \phi(N)$ gilt. Der **öffentliche Schlüssel** public key besteht dann aus

   - *N*, dem *Primzahlprodukt* sowie
   - *e*, dem *öffentlichen Exponenten*.

   Der **private Schlüssel** private key besteht aus

- – *d*, dem *privaten Exponenten* sowie

– *N*, welches allerdings bereits durch den öffentlichen Schlüssel bekannt ist.

p, *q* und $\phi(N)$ werden nicht mehr benötigt und sollten nach der Schlüsselerstellung auf sichere Weise gelöscht werden.

Beispiel

Die durchzuführenden Schritte werden mit einem Minimum an mathematischen Formeln beschrieben. Für das Beispiel werden bewusst kleine und einfache Zahlen verwendet, die natürlich für die Realität keineswegs hinreichend sind. Ein Beispiel mit größeren Zahlen wird mit dem Computer-Algebra-System MuPAD durchgeführt.

1. Schritt

Wir benötigen zwei sehr große Primzahlen; für die reale Verschlüsselung sollten die Primzahlen 1024 bit groß sein, also 100 oder mehr Ziffern haben. Wir begnügen uns mit $p = 3$ und $q = 5$.

2. Schritt

Nun sollen wir eine Zahl e finden, die folgende Bedingungen erfüllen muss:

1. e muss kleiner als der Wert sein, der sich ergibt, wenn man p und q miteinander multipliziert (Produkt aus p und q)
2. e darf keine gemeinsamen Faktoren mit dem Produkt $(p-1)(q-1)$ haben
3. e braucht keine Primzahl zu sein, muss aber ungerade sein

Das Produkt $(p-1)(q-1)$ ist übrigens zwangsläufig gerade. Das Produkt pq nennen wir N, den **Modulus**. Er wird so in einigen Veröffentlichungen genannt. N sollte etwa doppelt so viele Ziffern wie die Primfaktoren p und q haben.

Wir berechnen nun weiter:

$pq = 15$, $(p-1)(q-1) = 8$

Eine Primfaktorenzerlegung zeigt für 8 die Faktoren $2 \cdot 2 \cdot 2$. e darf also keinen Faktor 2 als Teiler haben, was ja ohnehin gegeben ist, da e ungeradesein soll. Der für unser Beispiel einfachste Wert wäre zB 3; also $e = 3$.

3. Schritt

Der Algorithmus schreibt nun vor, dass wir noch eine Zahl d bestimmen. Diese muss die Bedingung erfüllen, dass das um Eins verminderte Produkt aus d und e, $de - 1$, ohne Rest durch $(p-1)(q-1)$ teilbar ist oder mathematischer beschrieben:

$de = 1$ (mod $(p-1)(q-1)$)

und nicht gleich dem Wert von e .

Anmerkung: Die Zeile ist wie folgt zu lesen: de wird durch $(p-1)(q-1)$ so geteilt, dass Rest 1 bleibt.

Wie macht man das? Nun, eigentlich ganz einfach: man muss nur eine ganzzahlige Zahl x finden, die dafür sorgt, dass die folgende Gleichung ein ganzzahliges Ergebnis liefert:

$d = \frac{x(p-1)(q-1)+1}{e}$

Eine derartige Gleichung nennt man eine **diophantische Gleichung**, die in unserem vorliegenden Fall exakt lösbar ist.

In unserem einfachen Fall kann $x = 4$ gewählt werden, dann wird d zu

$d = (4 \cdot 2 \cdot 4 + 1)/3 = 33/3 = 11$

Damit gilt $de = 33$. Die Probe ist $33/8 = 4$ Rest 1.

Fassen wir mal unsere einzelnen Zahlen zusammen:

$p = 3$

$q = 5$

$N = 15$
$d = 11$
$e = 3$

Die Zahlen N (oder pq) und e bilden den **Public Key**. Die Zahlen N (oder pq) und d bilden den **Secret Key**. Dies erklärt auch die Zusatzforderung, dass e und d nicht den gleichen Wert haben sollten, da sonst Public und Secret Key gleich wären. Sind diese Zahlen einmal berechnet, so müssen alle Hilfsgrößen wie p, q, $(p-1)(q-1)$ und x gelöscht werden.

Verschlüsselung

Die Rechenvorschrift für die Verschlüsselung lautet:
$C = K^e$ mod N

C steht für Ciphertext
K steht für Klartext
e und N sind der Public Key des Empfängers.

Die Zeile sagt lediglich: Potenziere den Klartext mit e, teile das durch N und gib den Rest als Ciphertext aus. Nehmen wir mal an, wir wollen folgende Zahlenreihe verschlüsseln:

12130508

Die einzelnen zu verschlüsselnden Elemente müssen in Stücke zerlegt werden, die kleiner als N sind:

12 13 05 08

Die Ciphertexte wären demnach:

12^3= 1728	1728/15 = 115 Rest **03**
13^3= 2197	2197/15 = 146 Rest **07**
05^3= 125	125/15 = 8 Rest **05**
08^3= 512	512/15 = 34 Rest **02**

Das Ergebnis ist dann: **03 07 05 02**

Entschlüsselung

Die Rechenvorschrift für die Entschlüsselung lautet:
$K = C^d$ mod N

C steht für Ciphertext
K steht für Klartext
d und N sind der Secret Key des Empfängers

Die Mathematik ist exakt die gleiche wie bei der Verschlüsselung. Rechnen wir mal den Ciphertext zurück:

03 07 05 02

03^{11}= 177147	177147/15= 11809 Rest **12**
07^{11}= 1977326743	1977326743/15= 131821782 Rest **13**
05^{11}= 48828125	48828125/15= 3255208 Rest **05**
02^{11}=2048	2048/15=136 Rest **08**

Beachte: $K = C^d = (K^e)^d = K^{de} = K^1$ mod N

5.8.3 Ein Rechenbeispiel . . .

Wir wählen $p = 5$, $q = 7$.

Damit erhalten wir für $N = p \cdot q = 5 \cdot 7 = 35$.

Für $\Phi(N)$ = gilt $\Phi(N) = (p-1)(q-1)$. Beachte, dass p und q beide Primzahlen sind gilt!

Somit ist $\Phi(N) = 4 \cdot 6 = 24 = 2^3 \cdot 3$.

e darf also keine Faktoren 2 oder 3 enthalten; so kommen beispielsweise nur ungerade Zahlen für e in Frage.

Wir versuchen $e = 5$ und suchen die Modulo-Inverse d. Da $e \cdot d \equiv 1$ mod $\Phi(N)$ gelten muss, probieren wir der Reihe nach $d = \frac{x \cdot 24+1}{5}$ - für welches x ist d durch 5 teilbar?

$x = 1 \Rightarrow \frac{1 \cdot 24+1}{5} = \frac{26}{5}$
$x = 2 \Rightarrow \frac{2 \cdot 24+1}{5} = \frac{49}{5}$
$x = 3 \Rightarrow \frac{3 \cdot 24+1}{5} = \frac{73}{5}$
$x = 4 \Rightarrow \frac{4 \cdot 24+1}{5} = \frac{97}{5}$
$x = 5 \Rightarrow \frac{5 \cdot 24+1}{5} = \frac{121}{5}$
$x = 6 \Rightarrow \frac{6 \cdot 24+1}{5} = \frac{145}{5} = 29$ **BINGO!** $\Rightarrow d = 29$

(Wir testen: $e \cdot d = 5 \cdot 29 = 145 \equiv 1$ mod 24)

Mit (N, e) und (N, d) stehen uns der öffentliche Schlüssel und der private Schlüssel für das RSA-Verfahren zur Verfügung.

Wir verschlüsseln den Klartext $K = 10$ mit dem öffentlichen Schlüssel $e = 5$:

$C = K^e = 10^5 = 10^2 \cdot 10^2 \cdot 10 \equiv 30 \cdot 30 \cdot 10 = 90 \cdot 10 \cdot 10 \equiv 20 \cdot 30 = 60 \cdot 10 \equiv 25 \cdot 10 = 250 \equiv 5$ mod 35

Zum Entschlüsseln brauchen wir den privaten Schlüessel $p = 29$:

$K = C^d = 5^{29}$

Wie bilden wir diese hohe Potenz? Gehen wir schrittweise vor:

$5^3 = 125 \equiv 20$ mod 35
$5^4 = 5^3 \cdot 5 \equiv 20 \cdot 5 = 100 \equiv 30$ mod 35
$5^5 = 5^4 \cdot 5 \equiv 30 \cdot 5 = 150 \equiv 10$ mod 35

Damit zerlegen wir $5^29 = 5^5 \cdot 5^5 \cdot 5^5 \cdot 5^5 \cdot 5^5 \cdot 5^4 \equiv 10 \cdot 10 \cdot 10 \cdot 10 \cdot 10 \cdot 30 \equiv 5 \cdot 30 = 150 \equiv 10$ mod 35, also $K = 10$.

5.8.4 MuPAD:

```
>> // Wir suchen zwei grosse Primzahlen p und q
p := nextprime(1000);

                                    1009
>> q := nextprime(2000);

                                    2003
>> // Alice berechnet N und Phi(N) ... N = p.q, Phi-Funktion
Phi_N := (p-1)*(q-1);

                                   2018016
>> N:=p*q;

                                   2021027
>> // Alice berechnet ihren oeffentlichen Schlssel
e := random(2..Phi_N-2);

                            proc random() ... end
```

```
>> e := random(2..Phi_N-2);

                           proc random() ... end
>> >> e := random(2..Phi_N-2)();
Error: Unexpected '>' [line 1, col 2]
>> e := random(2..Phi_N-2)();

                                  479657
>>
igcd(e,Phi_N);

                                    1
>> d:=powermod(e,-1,Phi_N);

                                  1407929
>> (d*e) mod Phi_N;

                                    1
>> // Wir haben den oeffentlichen Schluessel von Alice (N,e):
N,e

                              2021027, 479657
>> // Alice hat ihren privaten Schluessel (N,d):
N,d

                             2021027, 1407929
>> // Bob sendet Alice eine Nachricht ...
// Die Nachricht muss eine ganze Zahl sein!
x:=17;

                                   17
>> y:=powermod(x,e,N);

                                  688446
>> // Alice entschluesselt die verschl. Nachricht von Bob ...
powermod(y,d,N);
```

Die Sicherheit des Verfahrens bezieht sich u.A. darauf, die Zahl N richtig zu faktorisieren. Diese Aufgabe wird bei großen N de facto unlösbar...

5.8.5 Beliebig lange Texte mit RSA verschlüsseln

Im letzten Beispiel haben wir lediglich für eine ganze Zahl die Verschlüsselung mit dem RSA-Verfahren diskutiert. In der Praxis treten jedoch Zeichenketten auf ...

```
>> // Beliebige Texte mit RSA-Verfahren ver- und entschluesseln
// Wir wandeln die Zahlenliste (!!!) der dem Text entsprechenden
// Ascii-Codes in eine 128er - Zahl um
Ascii2Int:=proc(Liste)
  local Zahl, i;
begin
  Zahl := 0;
  for i from 1 to nops(Liste) do
    Zahl := 128 * Zahl + Liste[nops(Liste) + 1 - i]
  end_for:
  return(Zahl):
end_proc:
>> //Text2Int wandelt einen Text in eine ganze Zahl um ...
Text2Int := proc(Text, n)
  local Liste, Quotient, Rest, Block, Zahl, i;
begin
  Liste := numlib::toAscii(Text):
  if nops(Liste) > n-1 then
    Quotient := _div(nops(Liste), n-1):
    Rest := modp(nops(Liste), n-1):
    Block := null():     // initialisiert das Objekt "Block"
    Zahl := null():
    for i from 1 to Quotient do
      Block := op(Liste, (i-1)*n - i + 2..i*(n-1)):
      Zahl := (Zahl, Ascii2Int([Block])):
    end_for:
```

```
    if Rest = 0 then
      return([Zahl]):
    else
      Block := op(Liste, Quotient*n - (Quotient+1)+2..nops(Liste)):
      Zahl := (Zahl, Ascii2Int([Block])):
      return([Zahl]):
    end_if:
  else
    return([Ascii2Int(Liste)]):
  end_if:
end_proc:
>> // Wir verschluesseln nun einen Text - dazu eine kurze
// Prozedur verschluesseln...
verschluesseln := proc(Modul, oeffentlich, Nachricht)
  local n, itext, Codierung;
begin
  n := floor(log(128,Modul)) + 1:
  itext := Text2Int(Nachricht, n):
  Codierung := [powermod(itext[i], oeffentlich, Modul) $ i = 1 .. nops(itext)]:
  return(Codierung)
end_proc:
>> // Zum Entschluesseln arbeiten wir umgekehrt ...
Int2Ascii:=proc(Zahl,n)
  local Liste, Quotient, Rest, i;
begin
  Quotient := Zahl:
  Liste := null():
  for i from 1 to n-1 do
    Rest := modp(Quotient, 128):
    Quotient := _div(Quotient, 128):
    Liste := (Liste, Rest):
  end_for:
  return(Liste):
end_proc:
>> //
Int2Text:=proc(Zahlliste, n)
  local Liste, i, Text;
begin
  Liste := null():
  for i from 1 to nops(Zahlliste) do
    Liste := (Liste, Int2Ascii(Zahlliste[i], n)):
  end_for:
  Text := numlib::fromAscii([Liste]):
  return(Text):
end_proc:
>> //
entschluesseln := proc(Modul, geheim, GeheimeNachricht)
  local n, text, Decodierung;
begin
  n:=floor(log(128, Modul)) + 1:
  Decodierung := [powermod(GeheimeNachricht[i], geheim, Modul) $ i = 1 .. nops(GeheimeNachricht)]:
  text := Int2Text(Decodierung, n):
end_proc:
>> // huch !!!
// jetzt anwenden ....
p := nextprime(2^64);

                          18446744073709551629
>> q := nextprime(p+2^16);

                          18446744073709617247
>> N := p*q;

                 340282366920939674381442582021575345363
>> Schluessel:=proc(p,q)
  local Phi_n;
begin
  Phi_n:=(p-1)*(q-1):
  for i from 1 to Phi_n do
    e := random(2 .. Phi_n -1)():
    if igcd(e, Phi_n) = 1 then
      print ("oeffentlich: " . e):
      break:
    end_if:
  end_for:
```

```
  d := e^(-1) mod Phi_n:
  print ("geheim: " . d)
end_proc:
>> Schluessel(p,q);

           "oeffentlich: 20107169056314830892607910405588105153"

             "geheim: 336306151939690575333302230623497162561"

>> text := "Informatik ist schoen und Lehrer luegen nicht (sagt ein Lehrer)":
>> GeheimeMitteilung := verschluesseln(N, e, text):
>> entschluesseln(N, d, GeheimeMitteilung);

     "Informatik ist schoen und Lehrer luegen nicht (sagt ein Lehrer)"
>>
```

5.9 Fiat-Shamir, Nullwissenprotokoll

Eine wichtige Frage stellt sich bei der Authentisierung einer Person durch ein Passwort. Sobald ein Passwort einmal genannt wurde, ist es nicht mehr geheim. Beispielsweise kann das Verfahren zur Authentisierung ein Passwort nur einmal schützen. Wird dieses Passwort jemanden mitgeteilt, so kann dieser es in Zukunft selbst verwenden und somit missbrauchen. Mögliche Vorkehrungen sind die Verwendung von Einmalpasswörtern (z.B. Transaktionsnummern im Bankverkehr) oder Methoden, die die Kenntnis eines Passworts belegen ohne dem Frager dieses offen zu legen (d.h. dem Frager keine Information über das Passwort übermitteln, daher auch *Zero-Knowledge-Protokolle*.

5.9.1 Der Algorithmus

Das Fiat-Shamir-Protokoll ist ein solches Zero Knowledge Protokoll, welches von Amos Fiat und Adi Shamir im Jahre 1986 vorgestellt wurde. Es erlaubt es, seinem Gegenüber zu beweisen, dass man eine geheime Zahl s kennt, ohne diese Zahl selbst preiszugeben. Das Verfahren arbeitet interaktiv, d. h. es finden mehrere Runden zwischen Geheimnisträger und dem Prüfer statt.

In jeder Runde kann die Kenntnis der Zahl zu 50 % bewiesen werden. Nach 2 Runden bleibt eine Restunsicherheit von 25 %, nach der 3. Runde nur noch bei 12,5 % usw. Die Sicherheit des Fiat-Shamir Protokolls stützt sich auf die Berechnung der Quadratwurzel im Restklassenring Z_n^*[13].

Veröffentlicht wird lediglich das Quadrat dieser Zahl gerechnet modulo einer Zahl m (und der Modul m selbst). Es ist ein schwieriges, rechenzeitaufwendiges Problem zu einer Zahl die Quadratwurzel (mod m) zu finden, wenn $m = p \cdot q$ das Produkt zweier großer Primzahlen ist[14].

Nehmen wir an, Alice möchte Bob von der Kenntnis der geheimen Zahl s überzeugen, ohnde diese Zahl offen zu legen. Dazu geht sie so vor:

Alice wählt ein Modul $m = p \cdot q$ und veröffentlicht m und $v = s^2$ (mod m). Das Paar (m, v) bildet den öffentlichen Schlüssel des Verfahrens. Um Bob von der Kenntnis von s zu überzeugen, geht Alice nun folgendermaßen vor:

1. Alice wählt eine zufällige Zahl r
2. Alice sendet deren Quadrat $x = r^2$ (mod m) an Bob.

[13] vgl. WikiPedia, Artikel “Fiat-Shamir-Protokoll”, Jan. 2006

[14] Die Tatsache, dass der Modul $m = p \cdot q$ und das Quadrat des “Geheimnisses s, $v = s^2$ mod m veröffentlicht wird, zeigt, dass es sich um eine so genannte **Public-Key-Verschlüsselung** handelt.

3. Bob darf jetzt genau eine der Zahlen r oder $y = r \cdot s$ mod m von Alice anfordern (um dies zu entscheiden, wählt er ein Zufallsbit, also 0 oder 1).
   Fordert Bob r, so kann Bob prüfen, ob r eine Quadratwurzel von $x = r^2$ mod m ist.
   Fordert Bob y, so kann Bob prüfen, ob $y^2 = r^2 \cdot s^2 = x \cdot v$ gilt (x und v sind ja bekannt).

4. Dieser Vorgang wird solange wiederholt, bis Bob mit ausreichender Wahrscheinlichkeit sicher ist, dass Alice das Geheimnis s kennt.

Das Verfahren ist korrekt, weil Alice das von ihr gewählte Geheimnis s kennt - also kann sie $y = r \cdot s^b$ berechnen. Bob akzeptiert dies, weil $y^2 \equiv (r \cdot s^b)^2 \equiv r^2 \cdot s^{2b} \equiv r^2 \cdot v^b \equiv x \cdot v^b$ mod m gilt.

Angenommen, ein Betrüger behauptet Alice zu sein (und deshalb das Geheimnis s zu kennen). Er möchte Bob davon überzeugen: Falls Bob zufällig das Bit $b = 0$ wählt, gilt $y = r \cdot s^0 = r$ und der Betrug gelingt. Ist aber $b = 1$, dann erfordert die Berechnung von y mit $y^2 \equiv x \cdot v$ mod m die Kenntnis des Geheimnisses s (vorausgesetzt, dass das Berechnen der Quadratwurzel modulo m für große m sehr schwer ist)... Allerdings darf Alice nicht gleichzeitig y und r senden, da sonst aus $y = r \cdot s$ einfach s berechnet werden könnte. Da nur eine der beiden Zahlen r oder y gefälscht werden kann, kann Bob sich durch zufällige Auswahl von der Richtigkeit einer dieser beiden Zahlen überzeugen. Wegen der zufälligen Wahl ist Bob jedoch bei jedem Durchgang nur zu 50% überzeugt, dass Alice nicht betrügt. Wird diese Überprüfung jedoch mehrfach wiederholt, so kann nach n Schritten davon ausgegangen werden, dass Alice nur mit einer Wahrscheinlichkeit von $0,5^n$ betrügt, also nach 10 Schritten zu 0,1%, nach 20 Schritten zu 0,0001% usw. Bob kann also selbst bestimmen, mit welcher Sicherheit er Alices Legitimität nachweisen will.

Sind zu $m = p \cdot q$ die Primfaktoren p und q bekannt, so kann zu einer beliebigen Zahl v mit großer Wahrscheinlichkeit und relativ effizient die Quadratwurzel s berechnet werden. Wird v beispielsweise aus einer Identifikation wie Name, Kartennummer usw. hergeleitet, so lässt sich (u.U. mit freier Zusatzzahl) ein quadratischer Rest v bestimmen, zu dem es eine Quadratwurzel gibt. Auf diese Weise lassen sich Chipkarten vor Fälschungen schützen. Die Sicherheit des Verfahrens beruht also darauf, dass große Zahlen nur schwierig zu faktorisieren sind, also zu m die Primfaktoren p und q zu finden.

5.9.2 Das Fiat-Shamir-Verfahren mit MuPAD

```
>> // Fiat-Shamir
>> // Alice kennt das Geheimnis s und moechte Bob davon ueberzeugen ...
>> // Alice waehlt zwei grosse Primzahlen p und q
>> p := 11; q := 17;

                                    11

                                    17
>> //Alice berechnet den Modul m
>> m := p * q;

                                   187
>> //Alice waehlt nun eine Zufallszahl s (also das Geheimnis):
>> s := random(1..m);

                             proc random() ... end
>> s := random (1..m)();

                                   109
>> // Alice behaelt dieses Geheimnis fuer sich, veroeffntlicht aber den Modul m und v = s^2 mod m:
>> v := powermod(s,2,m);

                                   100
>> // Nun berechnet Alice x mit Hilfe einer zu m teilerfremden Zufallszahl r:
>> r := random (1..m)(): while (igcd(r,m)>1) do r:=random(1..m)(): end_while: r;

                                   137
>> x := powermod(r,2,m);
```

```
                                    69
>> // Bob waehlt jetzt zufaellig 0 oder 1 und fordert entweder r oder y = r . s mod m
>> b := random(0..1)();

                                     1
>> // Damit berechnet Alice:
>> y := (r * powermod(s,b,m)) mod m;

                                    160
>> test:=powermod(y,2,m);
   if b=0 then if test=x then erg:="OK": else erg:="Kas": end_if: end_if:
   if b=1 then if test=modp(x*v,m) then erg:="OK": else erg:="Kas": end_if: end_if:
erg;

                                    168

                                   "OK"
>> // Wir testen fuer "viele" Runden:

>> teste:=proc()
   local r, x, test, b, y, erg;
   begin
     repeat
        r:= random(2..m)():
     until gcd(m,r)=1 end_repeat:r;
     x:=powermod(r,2,m);
     b:=random(0..1)();
     y:=(r * powermod(s,b,m)) mod m;
     test := powermod(y,2,m);
     if b=0 then
        if test = x then erg:="OK": else erg:="Kas": end_if:
     end_if:
     if b=1 then
        if test = modp(x*v,m) then erg:="OK": else erg:="Kas": end_if:
     end_if:
     return([b, test, erg]);
   end_proc:
>> teste() $ i=1..10;

[0, 36, "OK"], [1, 60, "OK"], [0, 25, "OK"], [1, 49, "OK"], [1, 49, "OK"],

   [0, 89, "OK"], [1, 60, "OK"], [0, 1, "OK"], [1, 16, "OK"],

   [1, 137, "OK"]
>>
```

In diesem Fall konnte in 10 Runden die Gültigkeit (dass Alice tatsächlich das Geheimnis s kennt) jeweils mit einer Wahrscheinlichkeit von 50 % überprüft werden. Dieses Ereignis tritt mit der Wahrscheinlichkeit von lediglich $0,5^{10} = 0,00097...$ zufällig ein ...

Hinweis: Um das Verfahren de facto sicher zu gestalten, ist es notwendig, große Primzahlen p und q zu bestimmen:

```
>> p:=nextprime(2^400);

258224987808690858965591917200301187432970579282922351283065935654064762 20\
16841194629645353280137831435903171972747493557
>> q := nextprime(2^450);

290735489718242756219729523155201813741456544274927224112596079672255715 24\
53591693304764202855054262243050086425064711734138406514458837
>> m := p*q;

750751682880470022997115769550925686131175959354950353667789939076263156 26\
19231707947410198580331380848554019184705463145413927056322182038695930329\
08478474285970870173841044846940518128022293686301941402371293009360791949\
69989294140732851419213719992132 09
>> s:=random(1..m)():while (igcd(s,m)>1) do s:=random(1..m)():end_while:s;

176245812092419973031722841422623380599065120053252067939997191326639628 26\
84799896739257703235257830006485992683914045957695479709880440230315990131\
64297454347942454882342208522440192040038249500580991105311136652237860005\
```

```
8468091257468888479432752251694338
>> v := powermod(s,2,m);

191025035027748602924034846387778276319281630656078187849878247311390751 99\
868759051395025205067291240187491624962255075370667792367223684429453742 69\
129623264448478835148187787657552349127968215992607481235811225437409745 76\
586394422965345045811427444852554 2
>> r := random(1..m)(): while(igcd(r,m)>1) do r:=random(1..m)():end_while:r;

722262328759153478246922501144147665346788932548441707156147291471702653 50\
075519727000018350423682557075973449158035859055517691212663851982043252 75\
782553545791378742956013355649957315200549400047573150873458552951402363 66\
817069918258338326712841284555779 4
>> x:=powermod(r,2,m);

379173316577605782992173111217836637703384251341235692397210192541381973 61\
458781392970948805605119992200286111327590152432541275046209032213464301 67\
457868745130517288407314025316486263459041495298960708318060992656729328 59\
062009943745769665600777152571474 6
>> b:=random(0..1)();

                                    0
>> y := (r*powermod(s,b,m)) mod m;

722262328759153478246922501144147665346788932548441707156147291471702653 50\
075519727000018350423682557075973449158035859055517691212663851982043252 75\
782553545791378742956013355649957315200549400047573150873458552951402363 66\
817069918258338326712841284555779 4
>> erg1 := powermod(y,2,m):
erg2 := (x*powermod(v,b,m)) mod m:
is (erg1=erg2);

                                  TRUE
>>
```

5.10 Hash-Funktionen

5.10.1 Grundlagen

Hash-Funktionen berechnen aus einer Nachricht oder einem bestimmten Datenbestand ein - möglichst - eindeutiges Ergebnis, und zwar mit folgenden Eigenschaften:

1. Die Hashwerte mssen eindeutig sein, dh. wenn eine Hashfunktion auf die gleiche Datenmenge angewendet wird, muss der Hashwert der gleiche sein.
2. Die Hashwerte sollen sich bei geringer Änderung der Eingangsdaten dramatisch ändern (beispielsweise soll die Änderung eines Bits der Nachricht dazu führen, dass etwa die Hälfte aller Bits im im Hashwert geändert werden)
3. Die Hashwerte dürfen bei verschiedenen Eingabedaten nicht gleich sein (“Kollisionsfreiheit”)
4. Es darf nicht möglich sein, die Eingangsdaten aus dem Hashwert berechnen zu können (“Unumkehrbarkeit”).
5. u.a.m.

Ist das Konstruieren einer neuen Nachricht mit dem gleichen Hashwert mindestens so schwierig wie den diskreten Logarithmus modulo p zu bestimmen[15], dann liegt eine so genannte **starke Kollisionsresistenz** vor.

[15] DLP - *Disrete Logarithmus Problem*

Ein Hash-Funktion ist somit eine **Einweg-Funktion**, die eine Nachricht komprimiert. Hashwerte werden beispielsweise dazu verwendet, aus Passwörtern so genannte “Shadow-Werte” zu berechnen, die schließlich auf dem Datenträger gespeichert werden[16].

5.10.2 Die Ziffernsumme

```
zs := proc(n)
  local summe;
begin
  summe:=0;
  while (n>=1) do
    summe := summe + (n mod 10);
    n := n div 10;
  end_while;
  return(summe);
end_proc:

zs(12025);

                                    10

zs(807);

                                    15

zs(10238239);

                                    28

zs(20138239);

                                    28
```

5.10.3 ISBN-Nummer

Die ISBN-Nummer (*International Standard Book Number*) ordnet jedem Buchtitel eine - weltweit - eindeutige Nummer zu. Die 10-stellige Zahl besteht aus

1. Gruppennummer
2. Verlagsnummer
3. Titelnummer
4. Prüfziffer

Während die Prüfziffer immer einstellig ist (0..9,X), ist die Stellenanzahl von Gruppennummer, Verlagsnummer und Titelnummer beliebig - insgesamt bilden sie zusammen genau 9 Ziffern. Die Prüfziffer dient dazu, Übertragungsfehler dieser 9 Ziffern zu detektieren. Sie wird so berechnet, dass man den Ziffern der Reihe nach mit 1, 2, 3, ... 9 multipliziert und das Ergebnis modulo 11 bildet.

```
>> isbn:=proc(isbn)
     local summe,i;
   begin
     summe:=0:
     for i from 9 downto 1 do
       summe:=summe+modp((modp(isbn,10)*i),11):
```

[16] Dass Passwörter im Klartext auf einem Datenträger gespeichert werden, womöglich noch auf dem System, an dem man sich mit diesem Passwort anmelden möchte, gehört zu den gröberen Dummheiten in der Geschichte der EDV.

```
      isbn:=_div(isbn,10):
    end_for:
    return(summe mod 11):
  end_proc:
>> isbn(917285036);
```

1

5.10.4 ISBN-13

Für die ISBN-Nummer mit 13 Stellen werden die Ziffern mit geradem Index mit 3 multipliziert und alle Ziffern modulo 10 addiert. Das Ergebnis, das auch 10 lauten kann, wird nocheinmal mod 10 vereinfacht. Damit kann die Prüfziffer die Werte 0, 1, ... 9 annehmen.

```
>> isbn13:=proc(isbn)
     local summe,i;
   begin
     summe:=0:
     for i from 12 downto 1 do
       summe:=summe+modp((modp(isbn,10)*3^modp(i+1,2)),10):
       isbn:=_div(isbn,10):
     end_for:
     return(summe mod 10):
   end_proc;
```

```
proc isbn13(isbn) ... end
```

```
>> isbn13(978376572781);
```

2

Beachte: Die Prüfziffer lässt falsch eingegebene Ziffern sowie “Zifferndreher” sofort erkennen ...

5.10.5 Eine passable Hash-Funktion

Bevor wir die folgende Hash-Funktion mit MuPAD erstellen, überlegen wir uns einige Eigenschaften von Primzahlen. Eine besondere Rolle spielen dabei Primzahlen der Form $p = 2 \cdot q + 1$ (p und q werden “Sophie Germain Primzahlen” genannt).

In der zyklischen Gruppe Z_p^* haben dann erzeugende Elemente g (“Primitivwurzeln”) die Eigenschaft, dass $g^2 \neq 1$ mod p, $g^q \neq 1$ mod p und $g^{p-1} = 1$ mod p gilt.

Wir testen für $p = 7$, $q = 3$ ($p = 2 \cdot q + 1$):

2: $2^2 = 4$

$2^3 = 8 \equiv 1$

$\Rightarrow g \neq 2$

3: $3^2 = 9 \equiv 2$

$3^3 = 27 \equiv 6$

$3^4 = 81 \equiv 4$

$3^5 = 3^3 \cdot 3^2 \equiv 6 \cdot 2 = 12 \equiv 5$

$3^6 = 3^3 \cdot 3^3 \equiv 6 \cdot 6 = 36 \equiv 1$

$\Rightarrow g = 3$

4: $4^2 = 16 \equiv 2$

$4^3 = 4^2 \cdot 4 \equiv 2 \cdot 4 = 8 \equiv 1$

$\Rightarrow g \neq 4$

5: $5^2 = 25 \equiv 4$

$5^3 = 25 \cdot 5 \equiv 4 \cdot 5 = 20 \equiv 6$

$5^4 = 25 \cdot 25 \equiv 4 \cdot 4 = 16 \equiv 2$

$5^5 = 5^3 \cdot 5^2 \equiv 6 \cdot 4 = 24 \equiv 3$

$5^6 = 5^3 \cdot 5^3 \equiv 6 \cdot 6 = 36 \equiv 1$

$\Rightarrow g = 5$

6: $6^6 = 36 \equiv 1$

$\Rightarrow g \neq 6$

Wir haben also zwei Primitivwurzeln für Z_7^* gefunden. Primitivwurzeln werden im folgenden Algorithmus verwendet:

```
// Eine passable Hash-Funktion ...
h:=(x,y) -> modp(powermod(g,x,p) * powermod(z,y,p),p);

            (x, y) -> powermod(g, x, p)*powermod(z, y, p) mod p

suche_pq := proc(start)
  local saat, p, q;
begin
  repeat
    saat := random(start div 2 .. start)();
    q := numlib::prevprime(saat);
    p := 2*q+1;
  until isprime(p) end_repeat;
  return (p,q);
end_proc:
// Primitivwurzel zu p,q suchen ...
pw := proc(p,q)
  local a;
begin
  repeat
    a := random(2..p-1)();
  until (modp(a*a,p)>1) and (powermod(a,q,p)>1) end_repeat;
  return(a);
end_proc;

                          proc pw(p, q) ... end

// p und q suchen
pq := suche_pq(10^18);

                  1526932858784724983, 763466429392362491

p := op(pq,1): q := op(pq,2):
// Eine Primitivwurzel g von p bestimmen:
g := pw(p,q):
// z zufi¿½lig aus zstern(p) wi¿½len:
z := random(p)():
//
// Und jetzt hashen!
//
h(1234567890,1234567890);
```

```
                            723906937863658139

h(1234567899,1234567899);

                             71939681467958300

h(2134567890,2134567890);

                            1424968478824388713
```

5.10.6 Hashfunktionen in der Praxis

In der Praxis werden - derzeit - beispielsweise folgende Hashfunktionen eingesetzt:

MD5 - liegt den meisten Linux-Distributionen in Form von `md5sum` bei. Wird häufig zur Integritätsprüfung von Daten (etwa nach erfolgter Üertragung) verwendet, etwa von PGP. Sind auf Download-Seiten die originalen MD5-Summen angegeben, kann man nach erfolgtem Download “per Hand” überprüfen, ob die Daten komplett übertragen wurden: Der am System impelementierte MD5 muss den gleichen Hashwert wie auf der Download-Seite ergeben.

SHA-1 (*Secure Hash Algorithm*),

RIPEMD-160

Eingangsbit	Ausgangsbit															
Bit 1 - 16	58	50	42	34	26	18	10	2	60	52	44	36	28	20	12	4
Bit 17 - 32	62	54	46	38	30	22	14	6	64	56	48	40	32	24	16	8
Bit 33 - 48	57	49	41	33	25	17	9	1	59	51	43	35	27	19	11	3
Bit 49 - 64	61	53	45	37	29	21	13	5	63	55	47	39	31	23	15	7

Figure 1: Die Eingangspermutation IP

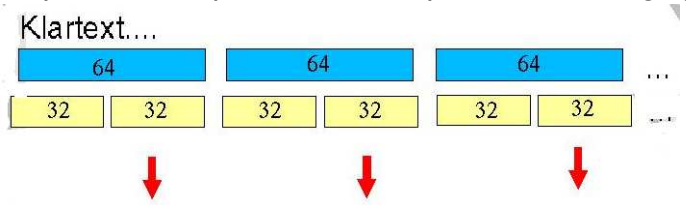
6 DES, Data-Encryption-Standard

Der DES wurde im Wesentlichen von IBM entwickelt und ist 1977 standardisiert worden. Er wurde von Anfang an öffentlich bekanntgegeben, diskutiert und hat gerade deshalb eine sehr große Verbreitung gefunden. Unzählige Meldungen, nach denen es gelungen sei ihn zu knacken, haben sich nie bestätigt.

Dieser Algorithmus ist ein so genannter symmetrischer Algorithmus, das heißt, dass zum Verschlëssseln und zum Entschlëssseln derselbe Schlüssel verwendet werden muss.

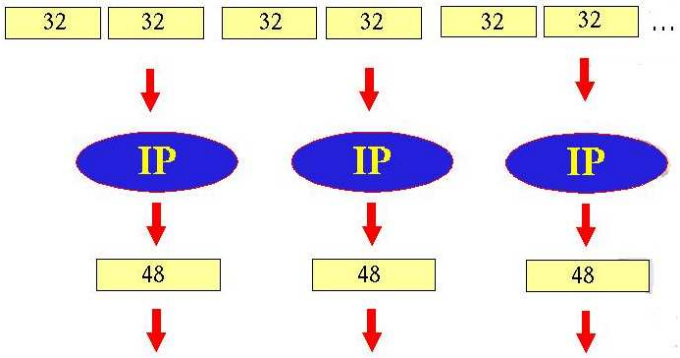
6.1 Beschreibung der Verschlüsselung

Der DES arbeitet immer in 64-Bit-Blöcken. Jeder Block wird zuerst einer Eingangspermutation IP (*initial permutation*) unterworfen. (siehe Abbildung 1)



Danach wird jeder Block in zwei 32-Bit lange Blöcke unterteilt auf die 16 Mal die folgenden Rechenschritte angewendet werden:

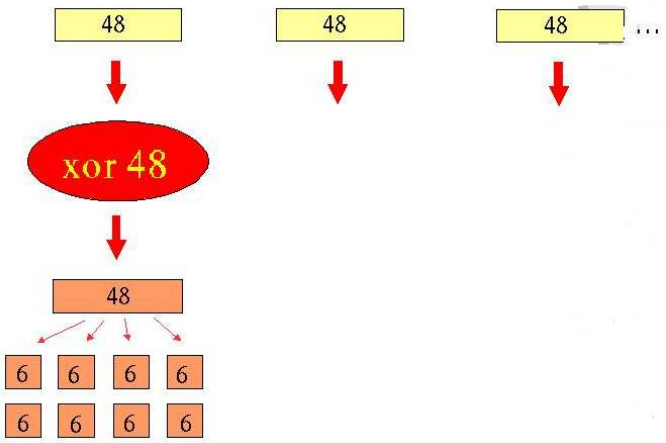
1. Zuerst wird der rechte Block mithilfe der Expansionsfunktion E (siehe Abbildung 2) auf 48 Bit erweitert.



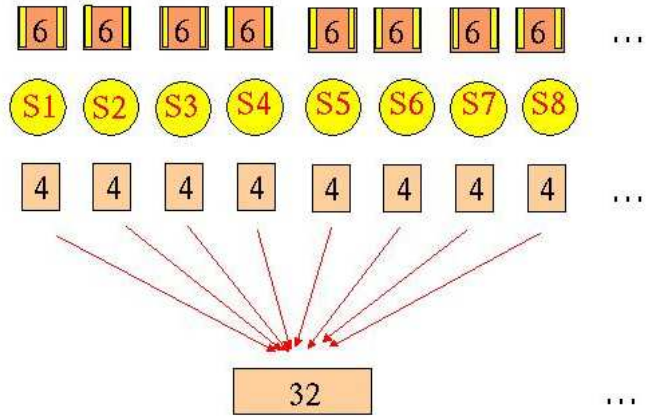
2. Der rechte Block wird dann mit einem 48-Bit langen Teilschlëssel, der bei jedem Teilschritt aus dem eigentlichen Schlüssel neu generiert wird, “exklusiv-oder” verknüpft.

Eingangsbit	Ausgangsbit											
Bit 1 - 12	32	1	2	3	4	5	4	5	6	7	8	9
Bit 13 - 24	8	9	10	11	12	13	12	13	14	15	16	17
Bit 15 - 36	16	17	18	19	20	21	20	21	22	23	24	25
Bit 37 - 48	24	25	26	27	28	29	28	29	30	31	32	1

Figure 2: Die Expansion E



3. Das so erhaltene 48-Bit lange Zwischenergebnis wird in 8 Blöcke zu je 6 Bit unterteilt.



4. Auf jeden 6-Bit langen Block wird dann in einer S-Box ein unterschiedlicher Rechenschritt angewendet. Dabei wird jeweils ein 4-Bit langes Ergebnis zurückgeliefert (siehe Abb. 3 und Abb. 4). In jeder S-Box wird ein 6-Bit-Wert, nach einer für jede S-Box verschiedenen Tabelle, verändert. Das erste und das letzte Bit, geben die Reihe an, die Bits 2 bis 5 geben hexadezimal gelesen die Spalte an.

5. Dann werden alle acht 4-Bit-Ergebnisse wieder zu einem 32-Bit-Block zusammengesetzt, auf den dann die Permutation P (siehe Abb. 5) angewendet wird.

Die S1-Box

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

Die S2-Box

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

Die S3-Box

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

Die S4-Box

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

Figure 3: Die S-Boxen (1-4)

Die S5-Box

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

Die S6-Box

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

Die S7-Box

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

Die S8-Box

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

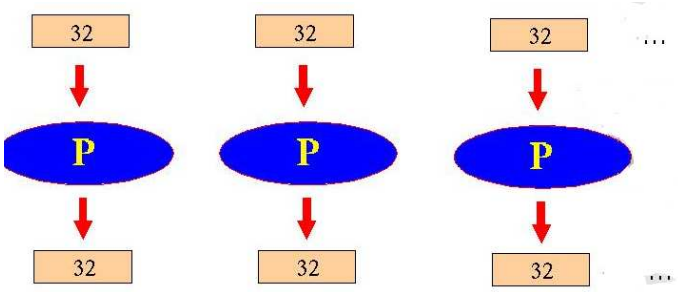
Figure 4: Die S-Boxen (4-8)

Eingangsbit	Ausgangsbit							
Bit 1 - 8	16	7	20	21	29	12	28	17
Bit 9 - 16	1	15	23	26	5	18	31	10
Bit 17 - 24	2	8	24	14	32	27	3	9
Bit 25 - 32	19	13	30	6	22	11	4	25

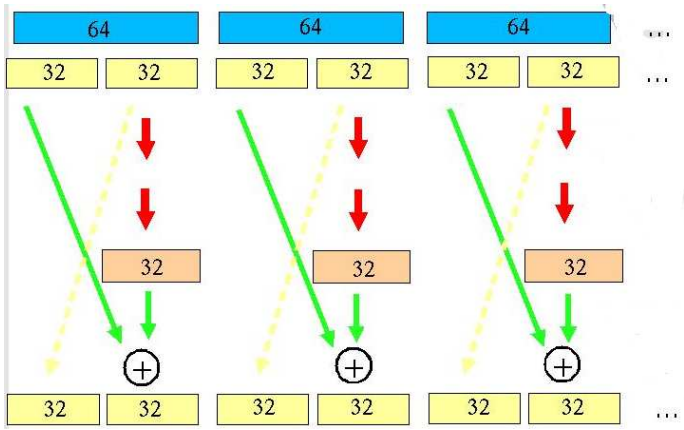
Figure 5: Die Permutation P

Eingangsbit	Ausgangsbit															
Bit 1 - 16	40	8	48	16	56	24	64	32	39	7	47	15	55	23	63	31
Bit 17 - 32	38	6	46	14	54	22	62	30	37	5	45	13	53	21	61	29
Bit 33 - 48	36	4	44	12	52	20	60	28	35	3	43	11	51	19	59	27
Bit 49 - 64	34	2	42	10	50	18	58	26	33	1	41	9	49	17	57	25

Figure 6: Die Ausgangspermutation IP^{-1}



6. Das Ergebnis wird dann mit dem linken Teilblock “exklusiv-oder” verknüpft und im nächsten Teilschritt als neuer rechter Teilblock verwendet.



7. Der rechte Teilblock dieses Schrittes wird im nächsten Teilschritt als linker Teilblock verwendet

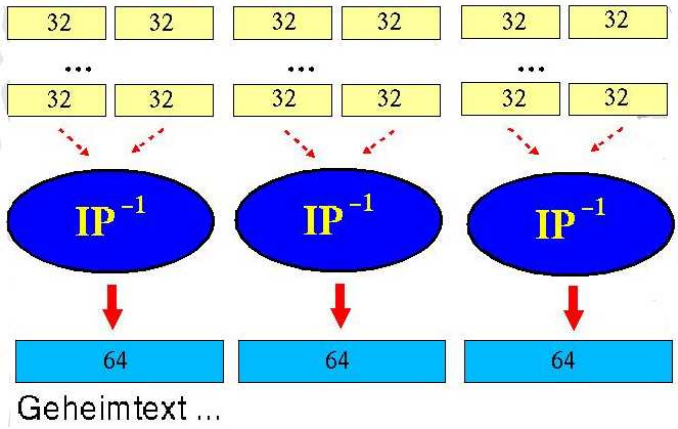
Nach 16 Teilschritten wird das 64-Bit-Ergebnis aus den beiden 32-Bit-Blöcken des letzten Teilschrittes zusammengesetzt, auf das dann eine Ausgangspermutation IP^{-1} (*inverse inititial permutation*) angewendet wird (siehe Abb. 6).

Eingangsbit	Ausgangsbit						
Bit 1 - 7	57	49	41	33	25	17	9
Bit 9 - 15	1	58	50	42	34	26	18
Bit 17 - 23	10	2	59	51	43	35	27
Bit 25 - 31	19	11	3	60	52	44	36
Bit 33 - 39	63	55	47	39	31	23	15
Bit 41 - 47	7	62	54	46	38	30	22
Bit 49 - 55	14	6	61	53	45	37	29
Bit 57 - 63	21	13	5	28	20	12	4

Figure 7: Die Schlüsselpermutation PC^{-1}

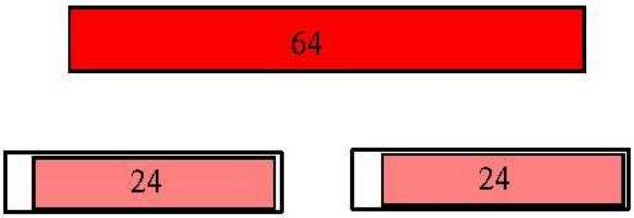
Schritt	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Linksschiebungen	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

Figure 8: Anzahl der Linksschiebungen pro Verschlëesselungsschritt



Schlüsselerzeugung:

Der Schlüssel für ein DES Verfahren muss 64 Bit lang sein. Der Schlüssel wird dabei als Kette von 8-Bit Blöcken betrachtet. Das achte Bit ist ein Kontrollbit das so gesetzt werden muss, dass die Parität des 8-Bit-Blocks ungerade ist.



Ausgangsbit	Eingangsbit											
Bit 1 - 12	14	17	11	24	1	5	3	28	15	6	21	10
Bit 13 - 24	13	19	12	4	26	8	16	7	27	20	13	2
Bit 25 - 36	41	52	31	37	47	55	30	40	51	45	33	48
Bit 37 - 48	44	49	39	56	34	53	46	42	50	36	29	32

Figure 9: Die Schlüsselauswahlfunktion PC2

Für jeden der 16 Teilschritte muss ein neuer 48-Bit Schlüssel aus dem 56 Bit langen Schlüssel erzeugt werden. Dies geschieht folgendermaßen:

1. Zuerst wird der eigentliche Schlüssel von den 64 Schlüsselbits getrennt und nach der in PC (*permutated choice*, siehe Abb. 7) festgelegten Ordung permutiert.

2. Dann wird der Schlüssel in zwei Teile aufgespaltet und in zwei verschiedenen Registern C und D zu je 28 Bit gespeichert.

3. Bei jedem Teilschritt werden die Bits in den beiden Registern zyklisch nach links verschoben. Dies geschieht entweder ein oder zweimal pro Schritt (siehe Abb. 8). Da insgesamt in allen 16 Schritten die Bits genau 28 mal nach links verschoben werden, befinden sich beide Register nach den 16 Teilschritten wieder in ihrer Ausgangsbelegung, das heißt, dass sie nicht neu geladen werden müssen, solange sich der externe Schlüssel nicht ändert.

4. In jedem Teilschritt wird aus der Zusammensetzung der beiden Register mithilfe der Schlüsselauswah PC2 (siehe Abb. 9) ein 48-Bit langer Teilschlüssel ausgewählt. Dieser 48-Bit lange Teilschlëssel wird dann “exclusiv-oder” mit dem auf 48 Bit erweiterten, rechten Block verknüpft. Das Ergebnis wird dann in den 8 S-Boxen weiterverwendet.

6.2 Die Sicherheit von DES

Die einzige Methode den DES zu knacken ist eine so genannte “brute-force-attack”, das heißt man probiert einfach alle möglichen Schlüssel durch und hofft den richtigen möglichst schnell zu finden.

Der DES arbeitet daher mit 56 signifikanten Schlüsselbits. Es gibt daher 2^{56} (mehr als 70 000 000 000 000 000) mï¿½liche Schlüssel. Als dieses Verfahren 1977 entwickelt wurde war es noch unvorstellbar, dass ein Computer jemals in der Lage sein wrde alle diese Mï¿½lichkeiten auszuprobieren.

Allerdings wird ein Verfahren sicherer, je mehr Rechenleistung für den Verschlëesselungsvorgang aufgewendet werden muss, da eine “brute-force-attack” dann mehr Zeit benï¿½igt. Das heiï¿½ aber nicht, dass ein langsamer Algorithmus sicherer als ein schneller ist. Da der DES eigentlich nur für Hardwareimplementierungen entwickelt wurde, suchten die Entwickler damals Methoden die aufgrund ihrer Struktur leicht und schnell in Hardware zu implementiern waren, bei einer Softwareimplementierung aber viel mehr Rechenleistung benï¿½igen. Zu diesen Methoden gehören die Eingangspermutation IP und die Ausgangspermutation IP^{-1}, die nicht direkt zu einer hï¿½eren Sicherheit beitragen, weil die Permutationsschablonen allen Anwendern bekannt sein mssen.

Im Januar 1999 veranstaltete die Firma RSA-Security Inc. eine DES-Challenge, dabei wurde eine mit DES verschlëesselte Nachricht in nur 22 Stunden und 15 Minuten geknackt. Auch wenn an der Challenge fast 100000 Computer weltweit beteiligt waren, zeigte diese Aktion die Notwendigkeit einer Erweiterung des DES.

6.3 Triple DES

Es wurde daher ein weiterer Standard in den DES aufgenommen: Triple DES. Beim Triple DES wird die Sicherheit dadurch erhöht, dass man dreimal hintereinander mit entweder 2 oder 3 verschiedenen Schlüsseln ver- oder entschlëesselt. Wenn nur 2 verschiedene Schlüssel verwendet werden wird für das erste und dritte Mal derselbe Schlüssel verwendet.

Es gibt 3 verschiedene Modi, mit denen Triple DES arbeitet (siehe Abb. 10).

EDE3	Encrypt, Decrypt, Encrypt mit drei Schlüsseln
EEE2	Encrypt, Encrypt, Encrypt mit zwei Schlüsseln
EEE3	Encrypt, Encrypt, Encrypt mit drei Schlüsseln

Figure 10: Mögliche Triple DES

7 Quantenkryptographie

17

Die Quantenkryptographie ermöglicht die sichere Übermittlung von Schlüsseln in öffentlichen Datennetzen. Sie macht sich dabei die Eigenschaften von Photonen zunutze.

Photonen werden dazu auf der Senderseite polarisiert: Mit einem Polarisationsfilter werden Photonen definierter Schwingungsrichtung ausgestrahlt. Auf der Empfängerseite misst eine Apparatur die Polarisation ankommender Photonen; dazu eignet sich beispielsweise ein Kalkspatkristall: Durchdringt ein Photon einen Kalkspatkristall ungebrochen, so hat es beim Austritt eine zu dessen optischer Achse senkrechte Polarisationsebene; wird es abgelenkt, dann ist es parallel zur optischen Achse des Kalkspatkristalls polarisiert. Trifft also ein Photon mit einer solchen “passenden” Schwingungsrichtung (sagen wir 0 ° oder 90 °) auf, so behält es diese bei und wird entweder abgelenkt oder auch nicht. Liegt die Polarisationsebene dazwischen, so wird es mit einer bestimmten Wahrscheinlichkeit ungebrochen durch den Kristall treten oder abgelenkt werden - dabei verliert es aber seine ursprüngliche Polarisation und nimmt die Schwingungsrichtung an, die dem Weg durch den Kalkspatkristall entspricht[18]. Am wenigsten vorhersagbar ist das Verhalten eines Photons, dessen Schwingungsrichtung bei 45 ° und 135 ° liegt.

Angenommen, Bob erfährt vorab, dass das nächste Photon entweder unter 0 ° oder 90 ° eintreffen wird: solche Photonen werden wir “gerade” Photonen nennen. Bob kann diese Photonen dann leicht mit seinem Detektor untersuchen und feststellen, unter welcher dieser beiden Polarisationsebenen das Photon bei ihm angekommen ist.

Möchte Bob aber “schräge” Photonen detektieren, so muss er sein Nachweisgerät um 45 ° drehen; in diesem Fall kann er aber keine Informationen über gerade Photonen erhalten. Gerade und schräge Polarisierung bilden somit ein Paar zueinander komplementärer Eigenschaften; die Messung der einen Eigenschaft zerstört die Informationen über die andere.

Darauf beruht das Verfahren, das Benett und Brassart 1984 vorgeschlagen und daher **BB84-Protokoll** genannt haben: Alice und Bob kommunizieren über einen Quantenkanal (LWL) und über einen öffentlichen Kanal (zB Telefonleitung). Lauschangriffe werden sofort entdeckt, da jede Messung an den übertragenen Photonen unweigerlich die Informationen über ihre Schwingungsrichtung vernichtet. Doch dazu später . . .

Alice und Bob vergleichen die durch den Quantenkanal gesendeten Daten mit Hilfe des öffentlichen Datenkanals: Zunächst erzeugt Alice eine Serie von Photonen, deren Polarisationsebene in zufälliger Folge entweder 0 °, 45 °, 90 ° oder 135 ° betragen und sendet sie an Bob. Beim Empfang jedes Photons entscheidet Bob zufällig, ob er dessen gerade oder schräge Polarisation misst. Als nächstes gibt Bob über den öffentlichen Kanal für jedes Photon bekannt, **welche Art von Messung er ausgeführt hat** aber **nicht das Messergebnis** (0 ° / 90 ° oder 45 ° / 135 °) selbst. Alice teilt ihm dann ebenfalls über den öffentlichen Kanal für jedes Photon mit, ob er

[17] Nach einem Artikel von Charles Benett, Gilles Brassard und Artur K. Ekert, Spektrum der Wissenschaften 2001, und auf der Basis der Diplomarbeit von Heidemarie Knobloch, Universität Wien 2009

[18] Ein Prinzip in der Quantenphysik besagt, dass jede Messung den physikalischen Zustand beeinflusst . . .

die richtige Arb von Messung gewählt hat. Alice und Bob verwerfen dann alle Fälle, in denen Bob (zufällig) die falsche Messung durchgeführt hat oder in denen sein Empfänger kein Photon nachgewiesen hat.

Wenn niemand den Quantenkanal abgehört hat, kennen Alice und Bob die verbleibenden Polarisationen und niemand sonst.

Wie überprüfen Alice und Bob nun, ob sie abgehört wurden? Dazu verwenden sie einen beliebigen Teil ihrer Photonen-Polarisationsdaten und vergleichen sie öffentlich. Ergibt sich nun, dass jemand gelauscht hat (weil ein Polarisationspaar nicht übereinstimmt), dann verwerfen sie das gesamte Experiment und wiederholen es. Hat niemand gelauscht, können sie den noch nicht verwendeten Teil ihrer Polarisationen als ihr gemeinsames Geheimnis als Schlüssel verwenden: Alle 0 ° und 135 ° - Polarisationen werden als 0 und alle 45 ° und 90 ° - Polarisationen als 1 verwendet.

Verbessert kann dieses Verfahren dadurch werden, dass nicht alle Photonen einer Folge verglichen werden, sondern lediglich überprüft wird, ob die Anzahl der “1” in einer zufällig bestimmten und öffentlich übermittelten Anzahl von Photonen gerade oder ungerade ist. Alice teilt Bob beispielsweise mit: “Ich habe das 1., 3., 4., 9., . . . 966. und 999. meiner 1000 Datenbits angeschaut; unter ihnen ist eine gerade Zahl von 1”. Bob zählt daraufhin ebenfalls ab - findet er eine ungerade Anzahl von “1”, so steht fest, dass ein Lauschangriff stattgefunden hat.

Eine gerade Anzahl beweist allerdings nicht, dass die Daten sicher sind, sondern legt dies nur mit der Wahrscheinlichkeit von 50 % fest. Wiederholt man diesen Test jedoch 10 mal oder 20 mal mit verschiedenen Teilmengen der Photonen-Polarisationen, so geht die Wahrscheinlichkeit einer unentdeckt gebliebenen Abweichung rasch gegen Null.

8 Kryptographische Verfahren in praktischen Beispielen

8.1 Passwörter

Passwörter (Kennwörter) werden zur Authentifizierung von Benutzern verwendet. Sie werden keinesfalls im Klartext gespeichert, sondern ihr Hashwert wird in einer so genannten “shadow”-Datei gespeichert. Bei der Authentifizierung wird daher zunächst der Hashwert des eingegebenen Passwortes berechnet; dieser Wert wird schließlich mit dem gespeicherten Hashwert verglichen.

Hier ein Beispiel einer Zeile aus der Datei `/etc/shadow`:

```
nus:$2a$10$h8dvSn968z5Ulhl36E8.d.HC8N05WPOc/rV.6tWZi.Y3VEHyTqUK2:13162:0:99999:7:::
```

Natürlich hat nur der Systemadministrator `root` Leserechte für die `shadow` - Datei...

8.1.1 Schwachpunkte

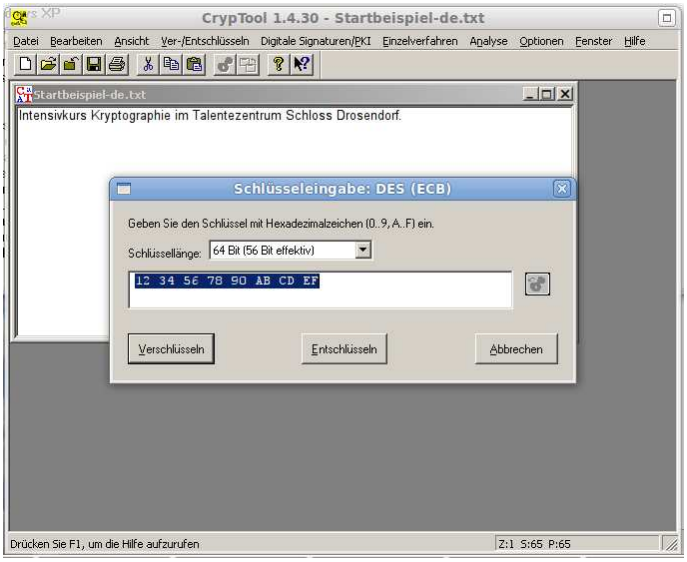
- Wird das Passwort - etwa über eine Netzwerkverbindung - unverschlüsselt übertragen, so kann dieses abgehört werden. Damit kann es der Angreifer gemeinsam mit dem unverschlüsselt übertragenen Benutzernamen jederzeit verwenden.
- Brute-Force-Attacke: Ein Skript probiert zahlreiche “gängige” Kombinationen aus Benutzername und Passwort durch, indem es beispielsweise Wörter aus einem Wörterbuch der Reihe nach als Passwort verwendet. Hat ein Systemadministrator etwa zum raschen Ausprobieren einen Benutzer “test” mit dem Passwort “test” angelegt und nicht wieder gelöscht, wird eine solche Attacke mit Sicherheit erfolgreich sein. Auch Passwörter, die aus sinnvollen

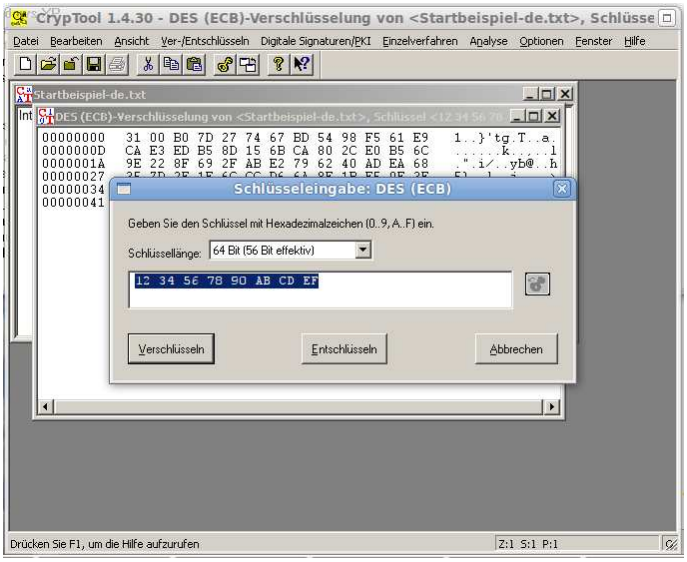
Silben bestehen sind potenziell gefährdet. Als Regel für gute Passwörter gilt daher, dass sie aus einer zufälligen Folge von Buchstaben und Ziffern bestehen sollen. Sind dann noch zusätzliche Zeichen wie Punkte, Unterstriche, etc. enthalten, ist das Passwort gut gegen Brute-Force-Attacken geschützt.

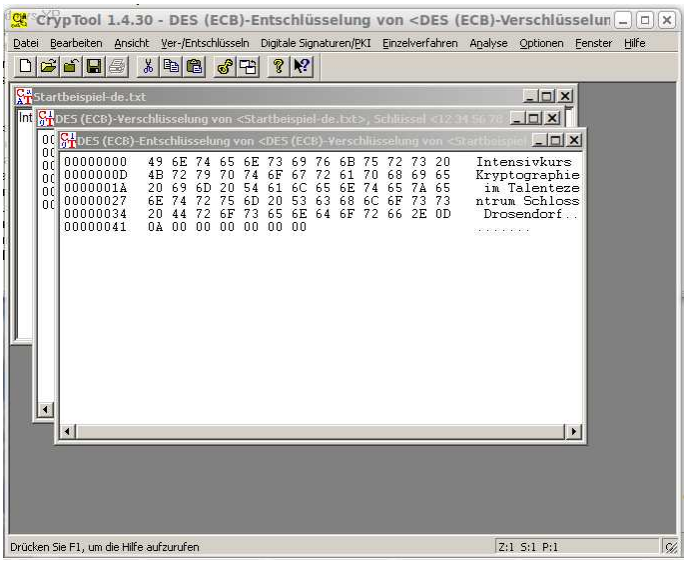
8.2 CrypTool

CrypTool steht als Lernsoftware für kryptographische Verfahren zur Verfügung (OSS):

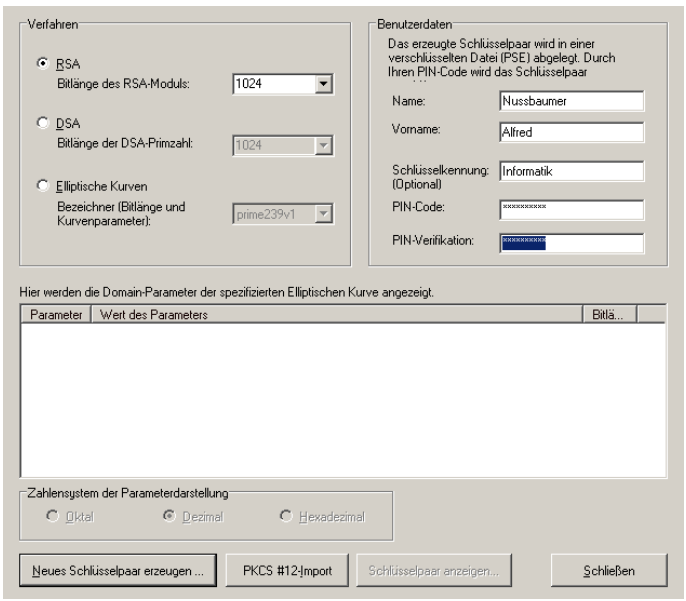
In einem Texteingabefeld werden zu bearbeitende Nachrichten eingegeben, die Verschlüsselungsverfa
aus dem Menü ausgewählt und angewendet. Die Schlüssel für symmetrische Verfahren werden als Hex-Zahlen eingegeben; die Primzahlen und Basen für asymmetrische Verfahren werden von CrypTool ermittelt.



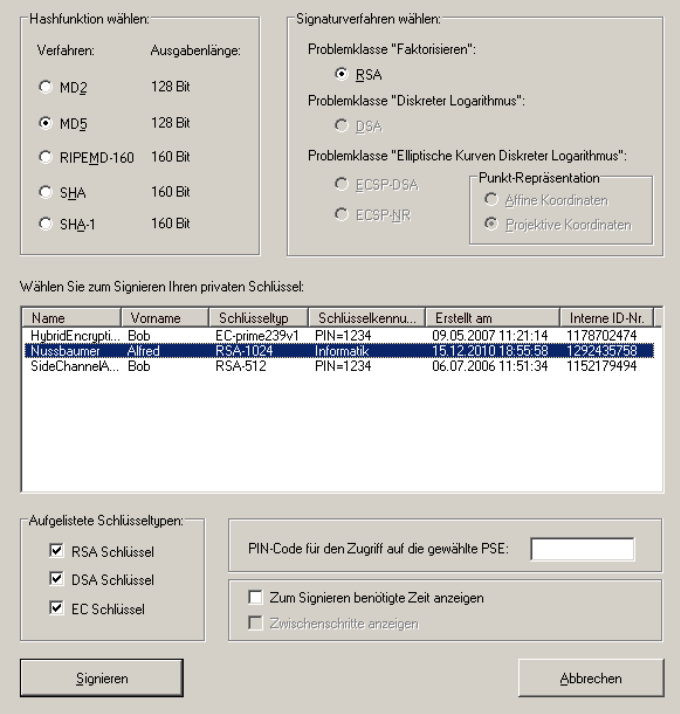




Für Signaturverfahren muss zunächst ein persönlicher Schlüssel erzeugt werden:



Der persönliche Schlüssel ist durch eine PIN gegen unbefugte Zugriffe geschützt:



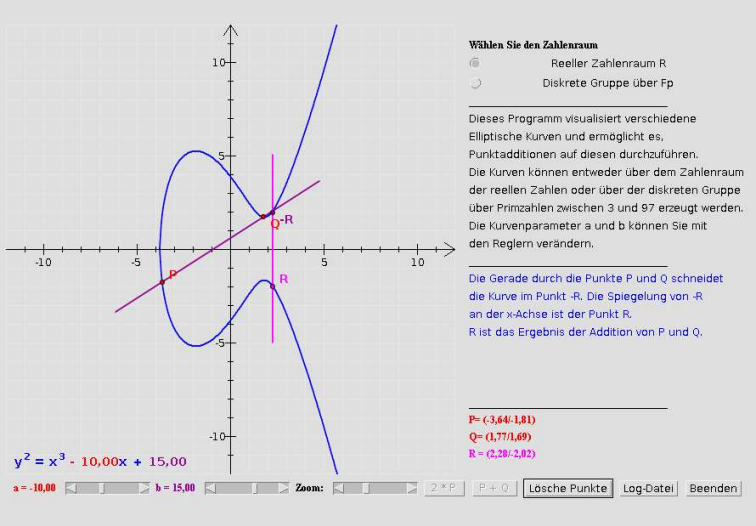
Signieren Sie eine Nachricht und überprüfen Sie die Signatur! Was geschieht, wenn Sie die signierte Nachricht geringfügig verändern?

8.2.1 Aufgaben:

- Zählen Sie systematisch Verschlüsselungsverfahren auf!
- Untersuchen Sie das Signieren einer Nachricht und den Test der Signatur!
- Untersuchen Sie die Hash-Verfahren!
- Beobachten Sie die Demonstrationen zur Signatur und zum RSA-Verfahren!
- Machen Sie sich mit den Analyse-Werkzeugen und -methoden vertraut!
- Untersuchen Sie mögliche Angriffe (zB “Known Plaintext”, “Brute Force” oder “Faktorisieren einer Zahl”)!

8.2.2 Elliptische Kurven

CrypTool stellt eine Visualisierung von verschiedenen elliptischen Kurven und der Punkt-Addition zur Verfügung:



8.3 TrueCrypt

TrueCrypt ist OpenSource-Software, die für Windows 2000/XP/Vista[19], MacOS und Linux **Datenträger**, **Verzeichnisse** oder **einzelne Dateien** sicher verschlüsselt. Dies erhält vor allem für USB-Sticks, externe Festplatten und Notebook-Rechner eine enorme Bedeutung: Im Fall eines Verlust der Datenträger (zB durch Diebstahl) gelangen verschlüsselte Daten nicht ungewollt in die Hände Dritter...

TrueCrypt steht kostenlos zur Verfügung und ist sehr einfach zu bedienen.

8.3.1 Download, Installation

TrueCrypt ist kostenlos verfügbar und kann von der Adresse `http://www.truecrypt.org` bezogen werden. Das ungefährt 1,5 MByte große ZIP-Archiv entpackt man in ein eigenes Verzeichnis und installiert das Programm. Dabei wird üblicherweise ein “Desktop-Icon” erzeugt, mit dem TrueCrypt gestartet werden kann.

8.3.2 Daten verschlüsseln

Um Daten verschlüsselt speichern zu können sind drei Schritte nötig:

1. Container anlegen (“Create Volume”). Der Vorgang läuft über einen “Wizzard” gesteuert ab. Dabei besteht die Möglichkeit, entweder einen “Standard-Container” oder einen “verborgenen Container” (“standard TrueCrypt volume”, “hidden TrueCrypt volume”) zu erstellen. Mit “Select File” wird der Container mit einem bestimmten Namen erzeugt. **Interessant:** Man legt den Verschlüsselungs-Algorithmus (defaultmäßig AES, alternativ DES, Blowfish, Twofish, CAST5, Serpent, Triple DES) und den Hash-Algorithmus (standardmäßig RIPEMD-160 fest. Die Größe des Containers hängt vom Dateisystem ab (Informationen werden vom Wizzard angezeigt). Anschließend legt man das Passwort für den Zugriff (Schreiben und Lesen) fest und wählt das Dateisystem für den Container aus (oft NTFS). Zuletzt wird der Container angelegt (“Volume created”). Ihm wurde ein - freier - Laufwerksbuchstabe (Windows!) zugeordnet.

[19] seit wenigen Tagen ist die Version 4.3 auch in der Lage unter Windows Vista Daten zu verschlüsseln

2. Das Volume wird - etwa im “Arbeitsplatz” mit seinem Laufwerksbuchstaben bezeichnet. Bevor man Daten dorthin schreiben oder von dort lesen kann, muss dieses Volume gemountet werden (Laufwerksbuchstabe wählen, anschließend Schaltfläche Mount). Natürlich muss dazu das Passwort für den Container angegeben werden...

3. Das gemountete Volume steht - zB im Dateimanager, Arbeitsplatz ... - wie ein Laufwerk zur Verfügung - Schreib- und Lesezugriffe sind unmerklich langsamer. Der “Benchmark-Test” gibt Auskunft über die Geschwindigkeit des ausgewählten Verfahrens.

Anmerkung: Anstelle einer Datei kann eine gesamte Partition oder das ganze Laufwerk verschlüsselt werden. Verschlüsselt man eine ganze Partition oder ein Laufwerk, so werden alle bestehenden Daten gelöscht.

8.3.3 Praktische Vorgangsweisen

- Bevor Daten verschlüsselt werden sollen, legt man einen Container, eine Partition oder ein ganzes Laufwerk fest und erzeugt damit ein TrueCrypt-Volume.

- Bevor Daten in ein TrueCrypt-Volume gespeichert oder Daten daraus gelesen werden können muss das Volume gemountet werden (vgl. oben). Dazu muss entweder das Passwort eingegeben werden, mit dem das Volume erstellt wurde, oder das Keyfile muss geladen werden, das auch beim Verschlüsseln verwendet wurde. **Hinweis:** Wird das Keyfile verloren, so können die damit verschlüsselten Daten nicht wieder hergestellt werden :-(

- Für USB-Sticks oder CD-ROMs kann eine so genannte **TrueCrypt-Traveller-Version** erstellt werden. Dazu wählt man einfach im Menüpunkt “Tools” das Erstellen der Traveller-Version aus - nur das Laufwerk, in dem diese Daten gespeichert werden sollen, muss angegeben werden (auf diesem wird ein Verzeichnis mit vier Dateien erstellt: `TrueCrypt.exe`, `TruCrypt Format.exe`, `truecrypt.sys` und `truecrypt-x64.sys`[20]). Wichtig: Zum Mounten des Truecrypt-Containers ist das Passwort bzw. das Keyfile erforderlich!

- Alternativ kann auch ein “dynamischer Container” angelegt werden - Vorsicht: Die Verarbeitung der Daten wird dann deutlich langsamer...

- Language-Pack: Man lädt das Package von `http://www.truecrypt.org/localizations.php` herunter, entpackt die gewählte Sprachdatei und kopiert sie in das TrueCrypt-Verzeichnis.

- Möchte man eine Festplatte oder eine Partition wieder “normal” nutzen, so kann die Partition oder die Festplatte nur gelöscht und neu formatiert werden - alle Daten im TrueCrypt-Container gehen dabei verloren.

8.3.4 Hidden Volume

TrueCrypt erlaubt das Erstellen von “Hidden Volumes“: Damit kann man innerhalb eines *Outer Volume* ein verstecktes Volume erstellen. Beide Volumes erhalten **verschiedene** Passwörter - wird man gewzungen, das Passwort für den TrueCrypt-Container herauszugeben, so kann der Angreifer dennoch das “hidden volumne” nicht erkennen. Beim Mounten muss man beide Volumes mounten.

[20] Diese 4 Dateien können auch händisch aus dem Verzeichnis der installierten vollen Version von TrueCrypt auf den Datenträger kopiert werden

8.4 GPG - GNU Privacy Guard

8.4.1 Grundlagen

GPG wurde als OpenSource-Ersatz für PGP entwickelt und steht für Linux, MacOS und Windows zur Verfügung. Es dient als freies Kryptographiesystem zum Signieren und Verschlüsseln von Nachrichten. Mit Hilfe verschiedener Programme kann GPG in zahlreichen Anwendungen eingesetzt werden (zB Evolution, Enigmail, etc.). Im Rahmen des Kurses soll gpg auf der Konsole Schritt für Schritt verwendet werden.

```
nus@ice:~/>gpg --version
gpg (GnuPG) 1.4.0
Copyright (C) 2004 Free Software Foundation, Inc.
This program comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it
under certain conditions. See the file COPYING for details.

Home: ~/.gnupg
Unterstuetzte Verfahren:
Oeff.Schluessel: RSA, RSA-E, RSA-S, ELG-E, DSA
VerschlÃ¼.: 3DES, CAST5, BLOWFISH, AES, AES192, AES256, TWOFISH
Hash: MD5, SHA1, RIPEMD160, SHA256, SHA384, SHA512
Komprimierung: nicht komprimiert, ZIP, ZLIB, BZIP2
```

8.4.2 GPG verwenden

1. `gpg --gen-key` - erzeugt ein Paar aus dem privaten und öffentlichen Schlüssel einer Person. Da der Schlüssel mit Hilfe von Zufallszahlen erzeugt wird, soll die Person während der Schlüsselerzeugung verschiedene Aktivitäten (Tastatur, Maus, Anwenderprogramme) setzen. Weiters gibt man den Namen, die Mail-Adresse und eine (Berufs?)Bezeichnung des Schlüsselbesitzers an. Verschiedene Verschlüsselungsverfahren werden angeboten - Elgamal wird vorgeschlagen. Die verfügbaren Schlüssellängen sind derzeit 1024, 2048 und 4096 Bit... Sobald der Schlüssel erzeugt wurde, stehen im Verzeichnis `.gnupg` entsprechende Dateien:

   ```
   nus@ice:~/.gnupg> ls
   agent.info  gpg.conf           pubring.gpg   pubring.kbx  secring.gpg
   agent.pid   private-keys-v1.d  pubring.gpg~  random_seed  trustdb.gpg
   ```

2. `gpg --list-keys` - gibt eine Liste der zur Verfügung stehenden Schlüssel aus

   ```
   nus@ice:~/.gnupg> gpg --list-keys
   /home/nus/.gnupg/pubring.gpg
   ----------------------------
   pub   1024D/354B56C0 2006-01-22 [expires: 2006-01-23]
   uid                  Alfred Nussbaumer (Lehrer am Stiftsgymnasium Melk) <alfred.nussbaumer@schule.at>
   sub   4096g/9D4FF124 2006-01-22 [expires: 2006-01-23]
   ```

3. `gpg -a -export <USERID>` - den öffentlichen Schlüssel exportieren. Die <USERID> ist ein Teil der eingegebenen Informationen zu Name, E-Mail oder Bezeichnung des Benutzers. Der Parameter “a” (armor) gibt den Schlüssel in einem ASCII-File (“ASCII-Hülle”) aus.

4. `cat oeff_schluessel.txt | gpg --import` - der in der (ASCII-)Datei “oeff_schluessel.txt” enthaltene öffentliche Schlüssel wird in das individuelle Schlüsselsystem importiert (= “zum Schlüsselbund hinzugefügt”).

5. `gpg -e -r <USERID> <datei>` - eine Datei mit dem privaten Schlüssel der Person verschlüsseln, deren USERID (oder Teile des Namens, der Mail-Adresse oder Bezeichnung) angegeben wurde.

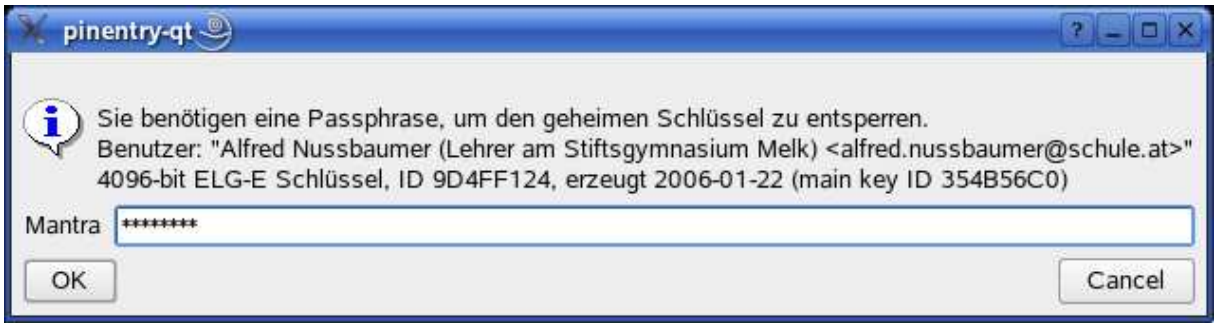
```
nus@ice:~/.gnupg> gpg -e -r Alfred nachricht.txt
```

   Anschließend besteht die neue (Binär-) Datei `nachricht.txt.gpg`, die den Inhalt der Klartextdatei `nachricht.txt` verschlüsselt enthält.

6. `gpg -d <datei>` - Chiffre entschlüsseln

```
nus@ice:~/.gnupg> gpg -d nachricht.txt.gpg

You need a passphrase to unlock the secret key for
user: "Alfred Nussbaumer (Lehrer am Stiftsgymnasium Melk) <alfred.nussbaumer@schule.at>"
4096-bit ELG-E key, ID 9D4FF124, created 2006-01-22 (main key ID 354B56C0)

gpg: encrypted with 4096-bit ELG-E key, ID 9D4FF124, created 2006-01-22
      "Alfred Nussbaumer (Lehrer am Stiftsgymnasium Melk) <alfred.nussbaumer@schule.at>"
Informatik ist schoen und Lehrer luegen nicht, sagt ein Informatiklehrer
```

   Da ja mit dem geheimen Schlüssel des Anwenders entschlüsselt wird, muss sich der Besitzer des privaten Schlüssels mit seiner “Passphrase” legitimieren:



7. `gpg -s <datei>` - Daten verschlüsseln und signieren. Da zum Signieren der private Schlüssel verwendet wird, muss auch hier der Besitzer sein Kennwort eingeben. GPG erzeugt in diesem Fall eine Signatur, die den Namen der Datei mit angehängter Bezeichnung `.gpg` trägt.

8. `gpg --verify <datei>` - prüft die Signatur einer Nachricht

```
nus@ice:~/.gnupg> gpg --verify nachricht.txt.gpg
gpg: Signature made So 22 Jan 2006 23:36:40 CET using DSA key ID 354B56C0
gpg: Good signature from "Alfred Nussbaumer (Lehrer am Stiftsgymnasium Melk) <alfred.nussbaumer@schule.at>"
```

9. `gpg -d <datei>` - prüft die Sigantur einer Nachricht und entschlüsselt sie

```
nus@ice:~/.gnupg> gpg -d nachricht.txt.gpg
Informatik ist schoen und Lehrer luegen nicht, sagt ein Informatiklehrer
gpg: Signature made So 22 Jan 2006 23:36:40 CET using DSA key ID 354B56C0
gpg: Good signature from "Alfred Nussbaumer (Lehrer am Stiftsgymnasium Melk) <alfred.nussbaumer@schule.at>"
```

10. `gpg --clearsign <datei>` - Klartextdaten werden verschlüsselt. In diesem Fall können die Daten gelesen werden; die gesamte signierte Nachricht wird durch Linien geklammert in der gleichnamigen Datei mit der Endung `.asc` ausgegeben:

```
nus@ice:~/.gnupg> gpg --fingerprint
/home/nus/.gnupg/pubring.gpg
----------------------------
pub   1024D/354B56C0 2006-01-22 [expires: 2006-01-23]
      Key fingerprint = 0569 937B 4B81 723C 1EB1  C2E4 28A0 7F85 354B 56C0
uid                  Alfred Nussbaumer (Lehrer am Stiftsgymnasium Melk) <alfred.nussbaumer@schule.at>
sub   4096g/9D4FF124 2006-01-22 [expires: 2006-01-23]

-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1

Informatik ist schoen und Lehrer luegen nicht, sagt ein Informatiklehrer
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.0 (GNU/Linux)

iD8DBQFD1AqiKKB/hTVLVsARAiz3AJ43GzZP8BYg6tkmknqkNiFsTZdOKACdFAxs
jNXjxIYPEF3cLdtexx0llBk=
=wJii
-----END PGP SIGNATURE-----
```

11. `gpg --gen-revoke > <datei>` - ein Revokations-Zertifikat anfertigen. Dieses dient dazu, seinen eigenen öffentlichen Schlüssel aus einem (öffentlichen) Schlüsselsystem wieder zurückzurufen. Das Zertifikat wird bei der Generierung am besten gleich in eine Datei mit dem angegebenen Namen umgelenkt.
12. `gpg --list-keys` - zeigt alle öffentlichen Schlüssel des Schlüsselbundes an
13. `gpg --list-sigs` - zeigt alle öffentlichen Schlüssel und alle Signaturen an
14. `gpg --fingerprint` - zeigt alle öffentlichen Schlüssel und alle so genannten *Fingerprints* an.

```
nus@ice:~/.gnupg> gpg --fingerprint
/home/nus/.gnupg/pubring.gpg
----------------------------
pub   1024D/354B56C0 2006-01-22 [expires: 2006-01-23]
      Key fingerprint = 0569 937B 4B81 723C 1EB1  C2E4 28A0 7F85 354B 56C0
uid                  Alfred Nussbaumer (Lehrer am Stiftsgymnasium Melk) <alfred.nussbaumer@schule.at>
sub   4096g/9D4FF124 2006-01-22 [expires: 2006-01-23]
```

15. `gpg --delete-key <USERID>` - löscht einen öffentlichen Schlüssel aus dem Schlüsselbund
16. `gpg --edit <USERID>` - öffnet den Schlüssel der mit ¡USERID¿ ausgewählten Person. Auf der Kommandozeile (*Key-Management-Menü*) sind dann zahlreiche Kommandos möglich, die man sich am besten mit dem Befehl `help` anzeigen lässt. Der Editiermodus wird mit dem Befehl `quit` wieder beendet.

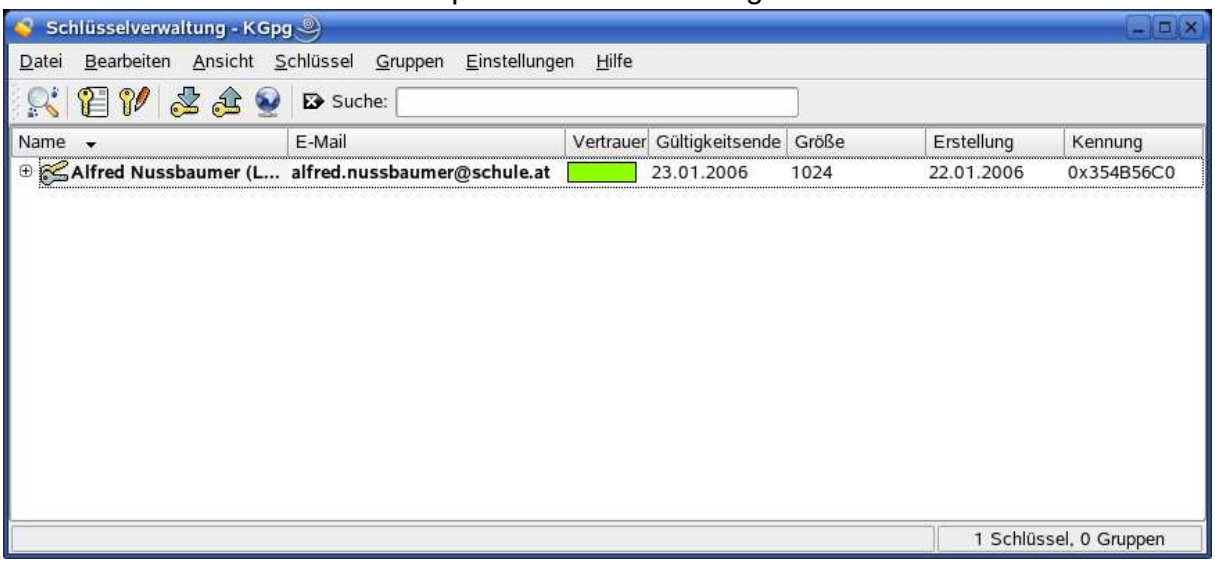
```
nus@ice:~/.gnupg> gpg --edit lehrer
gpg (GnuPG) 1.4.0; Copyright (C) 2004 Free Software Foundation, Inc.
This program comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it
under certain conditions. See the file COPYING for details.

Geheimer Schluessel ist vorhanden.

pub  1024D/354B56C0  created: 2006-01-22  expires: 2006-01-23  usage: CS
                     trust: uneingeschraenkt Gueltigkeit: uneingeschraenkt
sub  4096g/9D4FF124  created: 2006-01-22  expires: 2006-01-23  usage: E
[ultimate] (1). Alfred Nussbaumer (Lehrer am Stiftsgymnasium Melk) <alfred.nussbaumer@schule.at>

Befehl>
```

Zahlreiche Hinweise findet man auch mit `man gpg` - hier werden weitere Optionen angeführt und beschrieben.

8.4.3 KGPG

Für Linux / KDE steht beispielsweise das GPG-Backend KGPG zur Verfügung:

Beim erstmaligen Aufruf muss man einzelne Einstellungen vornehmen. Anschließend werden Schlüssel und Besitzer in einer Liste angeführt. Die mit `gpg` auf der Konsole zur Verfügung stehenden Befehle können hier bequem aus Menüs ausgewählt werden:



8.4.4 Programme für Windows

Auf dem Windows-Desktop stehen einige Programme für GnuPG, PGP, zur Verfügung (zum Teil in 30-Tage-Testversionen, anschließend kostenpflichtig[21]), zB:

- Windows Privacy Tools - `http://winpt.sourceforge.net/en/download.php`: Erzeugen und Verwalten von GnuPG-Schlüsseln
- CryptoStudio - `http://www.cryptostudio.com/download.html` - Signieren, Verschlüsseln, Entschlüsseln und Verizifieren ... (für alle Betriebssysteme verfügbar)

8.5 E-Mail

Derzeit werden die meisten Mails unverschlüsselt übertragen und gespeichert. Solche Mails können von jedermann gelesen werden - etwa von Systemadministratoren[22]. Ferner können Mails absichtlich mit einem fremden Absendernamen verschickt werden - für wichtige Nachrichten ist daher die Authentifizierung des Absenders erforderlich. Um Mails zu verschlüsseln, zu entschlüsseln

[21] **Tipp:** Suchbegriff mit Google eingeben ...

[22] Systemadministratoren mit entsprechenden Netzwerkbefugnissen dürfen dezidiert keine fremden Mails lesen (Strafbestand!)

oder zu signieren werden üblicherweise asymmetrische Public-Key-Verfahren eingesetzt. Voraussetzung ist, dass auf dem lokalen System eine entsprechende Verschlüsselungssoftware installiert ist, und dass die öffentlichen Schlüssel allgemein zugänglich sind ...

8.5.1 Nachrichten verschlüsseln

Berta möchte Anton eine verschlüsselte Nachricht senden, die nur er (und sonst niemand!) entschlüsseln kann. Dazu muss sie Anton öffentlichen Schlüssel kennen: Sie verschlüsselt die Nachricht mit Antons öffentlichen Schlüssel und überträgt sie (auch auf unsicheren Verbindungen) an Anton. Da nur er selbst im Besitz seines privaten Schlüessels sein kann, ist nur er in der Lage die verschlüsselte Nachricht zu entschlüsseln und zu lesen. Will Anton eine verschlüsselte Antwort an Berta senden, so muss er Bertas öffentlichen Schlüssel kennen...

8.5.2 Nachrichten signieren

Möchte Anton eine Nachricht an Berta versenden und signieren, so muss er zuerst ein Schlüsselpaar aus einem öffentlichen und geheimen Schlüssel erzeugen und den öffentlichen Schlüssel (auch auf unsicheren Verbindungen) Berta zukommen lassen. Wichtig dabei ist, dass Berta sicher sein kann, dass der übermittelte Schlüssel tatsächlich der öffentliche Schlüssel von Anton ist[23]. Dann bildet Anton einen **Hashwert** von seiner Nachricht. Diesen Hashwert verschlüsselt Anton mit seinem privaten Schlüssel und sendet die Nachricht und den verschlüsselten Hashwert der Nachricht an Berta.

Berta bestimmt nun ebenfalls den Hashwert von Antons Nachricht und entschlüsselt den mitgesendeten Hashwert mit Antons öffentlichen Schlüssel. Gelingt dies, so kann Berta sicher sein, dass die Nachricht von Anton stammt (Nachweis der Authentitzität). Stimmen außerdem der Hashwert, den Berta aus der übermittelten Nachricht erhalten hat, und der entschlüsselte Hashwert überein, so ist auch gezeigt, dass die Nachricht bei der Übertragung nicht verändert wurde (Nachweis der Datenintegrität).

8.6 SSH

8.6.1 Grundlagen

Secure Shell ermöglicht eine verschlüsselte und authentifizierte Verbindung zwischen Client (zB. Rechner “ice”) und Server (zB. Rechner “edu”). Um die beiden Rechner gegeneinander authentifizieren zu können, erhält jeder Rechner bei seiner Installation einen eindeutigen Schlüssel (zB RSA, privater und öffentlicher Schlüssel). Verbindet nun der Client “ice” zum Server “edu”, so kontrolliert `ssh`, ob der öffentliche Schlüssel des Servers dem Client schon bekannt ist: In diesem Fall ist der Rechnername, der Verschlüsselungs-Algorithmus und der öffentliche Schlüssel des entfernten Rechners am lokalen Rechner in der Datei `$HOME/.ssh/known_hosts` eingetragen:

```
edu.gymmelk.ac.at ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAIEAxBHyhFGpIdDNma5X/Po2Voa2Cjzxs7ON+mgbOXm
Ot4mHGt+bfR92JDTHa5Hhokyz2D2zyUW3X7LiQkTiTAB4Lzooi8U5i6uRJZA5wFSjGUB4Zrkf4uVti6bDgQiWKIdp2usd
JL8yoAxppnjKtlpU2Xdaw8VnxasR6yMTdh88omU=
10.0.1.2 ssh-dss AAAAB3NzaC1kc3MAAACBAMwcBR8i9tuRM1JLVkOwSSVVmCqwS8BUMwwrgUJDoKCc+umAjJJIq2ux
dxz24MHMb9OWJCNQwerxMJ3cjKrCDVCRHU6bBcpcb43ASaa1VMQ/homozwV9kxMQTospfrPNrFK3GbjJs2ORoT3gAj8UN
kfy9xJnKWGAo6KSeB+ni/HnAAAAFQDJXnLf0WIBV/ZKivKc0hu/W7nZjwAAAIEAnrEC+vkPUdn0LgNdHAzzHlz8/JYDxo
tOLrAGFl6ztvxCh267p87v5o9QMHIHbXjuUUcgPVkzeLot8ARGeGQ/1K6SSA3UbqGIE64iFGhYdywJwwlcdfjLM7HymBV
HabWeFo9c8d9hLyq1NCnbZCDEjbxQh9ssYNGvTPFN++JEw+MAAACAYTRX7QoAct6wisc6eecyNYYQFydYd98arcBRZAnL
EchIXjVxcFY+uY3+kEWOEEdm5lvqxuYq6UA7oRYHnu0YdR9X5S7BbWtaOXUZOX2XRNL0LMBTfQgmM0yelU3eux+7N51/L
XXwFNxUolkrak/BSd8y5vz8aaLP/XQ22Ya0sNk=
```

[23] Beispielsweise publiziert Anton seinen öffentlichen Schlüssel auf seiner Homepage, und Berta ist sich sicher, dass sie den Schlüssel auf Antons Homepage liest ...

Ist der öffentliche Schlüssel des Servers noch nicht bekannt, lässt `ssh` noch bestätigen, dass der Schlüssel gespeichert werden soll:

```
nus@ice:~> ssh edu.gymmelk.ac.at -l nus
The authenticity of host '10.1.1.1 (10.1.1.1)' can't be established.
RSA key fingerprint is ea:bc:f0:04:18:8f:79:b4:d8:5a:da:aa:ff:80:50:59.
Are you sure you want to continue connecting (yes/no)?
```

Da der RSA key fingerprint dem Server "edu" eindeutig zugeordnet ist, kann der Benutzer / die Benutzerin sofort erkennen, ob plötzlich unter dem Namen "edu" eine andere Maschine läuft (= *Man/Woman-in-the-middle-attack*). Weiß man sicher, dass man sich zum ersten Mal zum Server "edu" verbindet, kann man getrost "yes" eingeben: Damit wird der öffentliche Schlüssel des entfernten Servers "edu" in das lokale Verzeichnis `known_hosts` eingetragen.

Wird beim Anmelden am Server "edu" plötzlich ein **neuer** öffentlicher Schlüssel übergeben, liegt möglicherweise eine *Man/Woman-in-the-Middle*-Attacke vor. `ssh` warnt deutlich bei diesem Problem[24]:

```
nus@ice:~/.ssh> ssh edu.gymmelk.ac.at -l nus
key_read: uudecode AAB3NzaC1yc2EAAAABIwAAAIEAxBHyhFGpIdDNma5X/Po2Voa2Cjzxs7ON+mg
bOXmOt4mHGt+bfR92JDTHa5Hhokyz2D2zyUW3X7LiQkTiTAB4Lzooi8U5i6uRJZA5wFSjGUB4Zrkf4uV
ti6bDgQiWKIdp2usdJL8yoAxppnjKtlpU2Xdaw8VnxasR6yMTdh88omU=
 failed
The authenticity of host '10.1.1.1 (10.1.1.1)' can't be established.
RSA key fingerprint is ea:bc:f0:04:18:8f:79:b4:d8:5a:da:aa:ff:80:50:59.
Are you sure you want to continue connecting (yes/no)?
```

Ist man sich sicher, dass eine solche Attacke ausgeschlossen werden kann (beispielsweise, weil der Informatikdienst mitgeteilt hat, dass der Server neu aufgesetzt oder gegen einen Rechner mit dem gleichen Namen ausgetauscht werden musste), kann man mit "yes" fortsetzen. Andernfalls sollte man das Login an dieser Stelle sofort abbrechen ...
Anmerkung: Der öffentliche Schlüssel eines Rechners ist in der Datei `/etc/ssh/ssh_host_rsa_key.pub` gespeichert:

```
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAIEAxBHyhFGpIdDNma5X/Po2Voa2Cjzxs7ON+mgbOXmOt4mH
Gt+bfR92JDTHa5Hhokyz2D2zyUW3X7LiQkTiTAB4Lzooi8U5i6uRJZA5wFSjGUB4Zrkf4uVti6bDgQiW
KIdp2usdJL8yoAxppnjKtlpU2Xdaw8VnxasR6yMTdh88omU= nus@edu.gymmelk.ac.at
ssh_host_rsa_key.pub lines 1-1/1 (END)
```

Vergleiche diesen Eintrag mit der entsprechenden Zeile im Verzeichnis `$HOME/.ssh/known_hosts`!

8.6.2 scp

Secure Copy verwendet das SSH-Protokoll zum verschlüsselten Übertragen von Daten.

```
nus@ice:~/latex/krypto> scp kryptographie.pdf nus@www.gymmelk.ac.at:public_html/informatik/krypto
nus@www.gymmelk.ac.at's password:
```

Die wichtigsten Parameter sind `-P` (Angabe einer alternativen Portnummer) und `-r` (rekursives Kopieren des angegebenen Verzeichnisses) und `-C` (Komprimieren).

8.7 MD5

MD5 ist ein verbreiteter Hash-Algorithmus (vgl Abshnitt 2.3.1).

[24] Zu Übungszwecken wurde der Eintrag des öffentlichen Schlüssels mit dem Editor verändert; der Autor hat die ersten beiden Einträge, `AA`, gelöscht ...

8.7.1 md5sum

Auf Unix-artigen Betriebssystemen (wie z.B. Linux) steht die Funktion `md5sum` zur Verfügung. Sie berechnet den MD5-Wert einer Datei. Der Hersteller einer Datei bestimmt diesen Summenwert und veröffentlicht ihn gemeinsam mit seiner Datei. Soll diese Datei nun übertragen werden, so kann der Empfänger der Datei den Summenwert seines Downloads bestimmen und mit dem veröffentlichten Summenwert der originalen Datei vergleichen. Simmen beide MD5-Werte berein, so ist *mit großer Wahrscheinlichkeit* die Datei beim Download nicht verändert oder beschädigt worden.

Im folgenden Beispiel wird überprüft, ob der Download einer KNOPPIX-CD fehlerfrei war:

```
nus@ice:~/downloads> md5sum KNOPPIX_V3.8.2-2005-05-05-DE.iso
8ca2b9f2a0b83af6e6838961003dccd3  KNOPPIX_V3.8.2-2005-05-05-DE.iso
```

Den MD5-Wert erhalten wir auf der gleichen Seite wie die ISO-Abbildungsdatei:

Den Inhalt öffnet man durch einfachen Klick auf die md5-Datei:

Ist in diesem Beispiel der Download *mit großer Wahrscheinlichkeit* fehlerfrei verlaufen[25]?

8.8 Firewall

Eine Firewall soll einzelne Rechner oder (meistens) Computernetzwerke vor hohem Datenverkehr oder vor schädlichen Code schützen. Dies kann auf Hardwareebene und / oder auf Softwareebene erfolgen.

8.8.1 DMZ

Eine **Demilitarisierte Zone** (*Demilitarized Zone*) wird vom Internet und vom Intranet durch Paketfilter getrennt. In diesem Netzwerk stehen beispielsweise die Web-, Mailserver und Name-Server, die ja von außen “ungehindert” erreichbar sein müssen. Der Paketfilter gegen das Internet lässt beispielsweise Anfragen an den Webserver zu; der Paketfilter gegen das Intranet unterbindet beispielsweise alle Anfragen von außen: Im Fall eines Einbruches einen Rechner des DMZ ist das Intranet vor dem Eindringling abgeschottet.

[25] Allerdings ist der Zeitaufwand beim Download und das Datenaufkommen enorm - viele Breitbandanwender hätten damit ihr monatliches Datenvolumen von 500 MByte bereits berschritten!

Im einfachsten Fall sind die DMZ-Rechner zwischen dem Router ins Internet und der Firewall gegen das Intranet angeordnet (*Dirty Firewall*). Wesentlich besser ist es, die DMZ in ein eigenes Netzwerksegment zu positionieren, das an der Firewall hängt (*Protected Firewall*).

8.8.2 Application-Level-Firewall

Arbeitet auf der Applikations-Schicht (Schicht 7), indem beispielsweise HTML-Seiten auf Schadensroutinen überprüft werden (zB. Viren).

8.8.3 Paketfilter

Ein *Paketfilter* dient zur Kontrolle von Daten auf den unteren Ebenen des Schichtenmodells fÃ¼r die DatenÃ¼bertragung im Intranet/Internet. Dabei wird Ã¼blicherweise kontrolliert, welche Datenpakete von welchem Rechner (IP-Adresse und Port-Nummer) zu welchem Rechner (IP-Adresse und Port-Nummer) laufen dürfen.

8.8.4 Personal Firewall

Mit diesem Namen werden lokale Softwarelösungen (zB SuSEFirewall, Zonealarm, etc.) bezeichnet, die einen mit dem Internet verbundenen Rechner vor unerwünschten Zugriffen schützen (sollen). Ebenso sollen unerwartete Zugriffe vom PC ins Internet unterbunden werden (etwa wenn sich ein Mail-Virus vom Rechner ausgehend im Internet weiter verbreiten möchte). Die Schutzfunktion ist dennoch zur Zeit umstritten.

8.8.5 WLAN

Der Datenverkehr in WLAN (Funknetzen) sollen grundsätzlich verschlüsselt werden. Für WLAN kommen heute die Standards WEP, WPA und WPA2 in Frage. Lässt sich der Einsatz von WEP nicht vermeiden, sollten aber folgende teils grundlegende teils umstrittene Behelfsmaßnahmen beachtet werden, um Angriffe so genannter Scriptkiddies und zufällige Zugriffe fremder Personen auf das WLAN zu unterbinden:

- das Standard-Passwort des Access-Points ändern bzw. überhaupt erst mal ein Passwort setzen.
- der WEP-Schlüssel sollte mindestens 128-Bit lang und eine lose Kombination aus Buchstaben, Ziffern und Sonderzeichen sein.
- die Zugriffskontrollliste (ACL = *Access Control List*) aktivieren, um vom Access Point nur Endgeräte mit bekannter MAC-Adresse (*Media Access Control* - Adresse) zuzulassen. Die MAC-Adresse lässt sich aber mittels Treiber beliebig einstellen, sodass eine mitgelesene zugelassene MAC-Adresse leicht als eigene ausgegeben werden kann.
- die SSID des Access-Point sollte keine Rückschlüsse auf verwendete Hardware, Einsatzzweck oder Einsatzort zulassen.
- Umstritten ist die Deaktivierung der SSID-Übermittlung (Broadcasting). Sie verhindert das unabsichtliche Einbuchen in das WLAN, jedoch kann die SSID bei deaktiviertem Broadcasting mit einem Sniffer mitgelesen werden, wenn sich etwa ein Endgerät beim Access-Point anmeldet.

- WLAN-Geräte (wie der Access-Point) sollten nicht per WLAN konfiguriert werden, sondern ausschließlich über eine kabelgebundene Verbindung.
- im Access-Point sollte, sofern vorhanden, die Fernkonfiguration abgestellt werden.
- WLAN-Geräte ausschalten, wenn sie nicht genutzt werden.
- regelmäßige Firmware-Updates vom Access-Point durchführen, um sicherheitsrelevante Aktualisierungen zu erhalten.
- Reichweite des WLANs durch Reduzierung der Sendeleistung bzw. Standortwahl des WLAN Gerätes beeinflussen (dient nicht zur aktiven Sicherheit, lediglich der eventl. Angriffsbereich wird begrenzt)
- DHCP-Server deaktivieren und IP-Adressen manuell zuweisen

Zertifikate sollen die Authentizität von Webservern ausweisen. Recherchiere X.509-Zertifikate und lies zur Zertifizierung in der Online-Hilfe der Browsersoftware und im Internet nach!

IPsec wurde 1998 als Ersatz für das unsichere Internetprotokoll IP entwickelt. Mit Hilfe des IKE (*Internet Key Exchange* und verschiedenen Authentifizierungsmethoden sollen Vertraulichkeit, Authentizität und Integrität bei der Datenübertragung im Internet gewährleistet werden. Lies dazu im Internet nach!

VPN und **VPN-Tunnel** werden in überregionalen Netzwerken von Firmen und Organisationen tagtäglich eingesetzt. Wozu dient VPN und was sind die Grundfunktionen? Beschreibe, wie eine VPN-Verbindung zwischen zwei Rechnern (zwischen zwei Rechnernetzwerken) grundsätzlich realisert werden kann!

Recherchiere und beschreibe die so genannte **Man/Woman-in-the-Middle** - Attacke!

Recherchiere und beschreibe die **Replay** - Attacke!

9 Glossar

AES: *Advanced Encryption Standard*, Nachfolger von DES. Symmetrische Blockverschlüsselung mit einer Blocklänge von 128 Bits und Schlüssellängen von 128, 192 und 256 Bits.

Authentication: (Authentifizierung)

BLowfish: Von Bruce Schneier entwickelter und veröffentlichter Verschlüsselsungsalgorithmus.

CA: *Certification Authority*, Zertifizierungsinstanz - stellt digitale Zertifikate aus.

CBC: *Cipher Block Chaining*, jeder verschlüsselte Text wird mit dem nächsten Klartext mit XOR verbunden und anschließend mit dem Schlüssel verschlüsselt.

CFB: *Cipher Feedback*

Cipher: (engl. = Ziffer, Chiffre), bezeichnet einen Verschlüsselungsalgorithmus (z.B. DES, AES, IDEA, ...).

Ciphertext: Verschlüsselter Text.

Cross-Zertifizierung: Zwei CAs anerkennen sich gegenseitig und bestätigen dies durch gegenseitiges Ausstellen digitaler Zertifikate.

DES: *Data Encryption Standard*, 1975 von IBM veröffentlicht, 1994 - 1998 als Verschlsselungs-Standard. Verschiedene Betriebsarten:

Digitale Signatur: Elektronische Daten werden mit einem privaten Schlssel verschlsselt. Dabei wird eine Prfsumme (Hash-Wert) berechnet. Durch die Signatur wird die Authentizität des Unterschreibenden und die Integrität des Dokuments sichergestellt.

ECB: *Electronic Code Book*, mehrere Klartexte werden mit dem gleichen Schlüssel verschlüsselt.

GnuPG: *GNU Privacy Guard*, ein Ersatz für PGP, der ohne RSA oder andere patentierte Kryptoverfahren auskommt.

Hashwert: Ein aus den Zeichen einer Nachricht mit Hilfe wohldefinierter Algorithmen berechneter Wert (z.B. Prüfsummen). Wesentlich für die Überprüfung der Vollständigkeit einer bertragenen Nachricht.

IDEA: *International Data Encryption Algorithm*, ein an der ETH Zrich um 1990 - 1992 entwickeltes Verschlüsselungsverfahren; verwendet eine Schlüssellänge von 128 Bits.

IEFT: *Internet Engineering Task Force*, Internationale Vereinigung, die Internet-Standards entwickelt.

MAC: *Message authentification code*, an einen Klartext wird eine Prfsumme angehängt, die dem Empfänger die Authentizität der Nachricht berprften lässt.

MD: *Message Digest*, ein Hash-Wert der Nachricht, mit dem sicher gestellt ist, dass die Nachricht während der Übertragung nicht verändert wurde.

OFB: *Output Feedback*

OpenSSL: OpenSource-Lösung fr SSL/TLS.

PGP: *Pretty Good Privacy*, Software fr die Verschlsselung von E-Mails und deren Signierung.

PIN: *Personal Identification Number*, persönliche Identifzizierungsnummer.

PKCS: *Public-Key Crytography Standard*

PKI: Public-Key-Infrastruktur.

Private Key: Der geheime Schlüssel(teil) eines Public-Key-Verfahrens. Wird zum Entschlüsseln eingehender Nachrichten und zum Signieren ausgehender Nachrichten verwendet.

PSE: *Personal Security Environment*, persönliche Sicherheitsumgebung. Hier sind die Informationen für sichere Verbindungen, wie der private Schlüssel gespeichert. Diese Daten müssen durch Passwort oder PIN gesichert werden.

Public Key: Der öffentliche Schlüssel(teil) eines Public-Key-Verfahrens. Der öffentliche Schlüssel eines Teilnehmers wird verwendet, um eine Nachricht an ihn zu verschlüsseln. Mit dem öffentlichen Schlssel eines Teilnehmers wird die Gültigkeit seiner Signatur überprüft.

Public Key Verfahren: Asymmetrisches Verschlüsselungsverfahren, das ein Schlsselpaar aus einem öffentlichen und einem geheimen Schlüssel verwendet.

RA: *Registration Authority*, Registrierungsstelle. Gibt digitale Zertifikate für Personen oder Systeme aus.

Schlssel: Informationen, an Hand der Daten verschlüsselt und entschlüsselt werden können. Symmetrische Verfahren verwenden zum Verschlüsseln und Entschlüsseln den gleichen Schlüssel. Asymmetrische Verfahren verwenden zum Verschlüsseln einen öffentlich bekannten Schlüssel und zum Entschlüsseln einen geheimen, privaten Schlüssel.

SET: *Secure Electronic Transaction*, Üertragungsprotokoll für geschützte und authentische Datenübertrag (Internet-Banking, Kreditkartennummern).

S/MIME: *Secure Multipurpose Internet Mail Extension*, für Verschlüsselung und Signature von E-Mails erweitertes E-Mail-Format MIME.

SSH: *Secure Shell*, Verbindung mit einem entfernten Rechner, bei der alle übertragenen Daten verschlüsselt werden (vgl.: mit Telnet werden alle Daten im Klartext bertragen).

SSL: *Secure Socket Layer*, Internet-Protokoll für die Übertragung von verschlüsselten Nachrichten via TCP/IP (entwickelt von Fa. Netscape). Als Protokoll dient üblicherweise HTTPS.

Symmetrische Verschlüsselung: Ein gemeinsamer (geheimer) Schlüssel wird sowohl zum Verschlüsseln als auch zum Entschlüsseln von Nachrichten verwendet.

TLS: *Transport Layer Security*, Protokoll zur Authentifizierung und Verschlüsselung für TCP/IP-Netzwerke. Nachfolger von SSL (TLS1 entspricht weitgehend SSL3).

Triple-DES: Verwendet zwei verschiedene Schlüssel zu je 56 Bits: Der Klartext wird zunächst mit dem ersten Schlüssel DES-verschlüsselt, dann mit dem zweiten Schlüssel entschlüsselt und schließlich mit dem ersten verschlüsselt (damit ist DES als Spezialfall enthalten, wenn beide Schlüssel gleich sind). Als Variante werden auch drei verschiedene Schlüssel verwendet.

TrueCrypt: Freie Software zum Verschlüsseln von Festplatten, Verzeichnissen oder einzelnen Dateien (Linux, MacOS, Windows 2000/XP/Vista/7).

X.509: Schema zur Erstellung von Zertifikaten fr SSL/TLS - Authentifizierung.

Zertifikat: Ein wohldefinierter Datensatz, um Server und Clients gegeneinander zu authentifizieren. Verwendet das X.509-Schema. Enthält den Public Key des Eigentümers und die Signatur des Ausstellers.