Studienarbeit

Das RSA-Verfahren und seine Anwendung

SS 1991

Jürgen Weinelt Steffen Reith Oliver Stock

25. September 1991

Zusammenfassung

In dieser Studienarbeit sollte ein Public Key Cryptosystem implementiert und der dabei auftretende Effizienzverlust auf das Gesamtsystem untersucht werden. Weiterhin sollte eine Schlüsselverteilzentrale implementiert werden, die exemplarisch die Kommunikation zwischen zwei anderen Rechnern handhaben kann. Als Public Key Cryptosystem wurde das RSA-Verfahren gewählt, da es im Moment eines der sichersten und am besten erforschten Public Key Verfahren ist. Zur Verfügung standen IBM PS/2 Systeme, die mit dem Prozessor INTEL 80386 ausgerüstet sind. Als Implementierungssprache wurde MODULA2 gewählt, weil diese Sprache zum einen sehr gute Möglichkeiten der Strukturierung von Programmen zur Verfügung stellt und zum anderen auch systemnahe Programmierung ermöglicht.

Inhaltsverzeichnis

Kapitel 1

Das RSA-Verfahren

Das von Rivest, Shamir und Adlemann erfundene asymetrische Verschlüsselungsverfahren ist eines der heute am besten erforschten Public Key Cryptosysteme. Mit den heute bekannten Algorithmen kann es mit vertretbaren Aufwand nicht entschlüsselt werden. Wie bei jedem Public Key Cryptosystem kann der Schlüssel, der der Dechiffrierung dient, aus dem öffentlichen Schlüssel ermittelt werden. In der Praxis scheitert dies aber an mangelnder Rechengeschwindigkeit und Speicherkapazität.
Weil das RSA-Verfahren eine sehr große Sicherheit bietet, hat diese Verschlüsselungsmethode eine weite Verbreitung erlangt, und wird heute schon in bestehenden Softwarepaketen zur Verfügung gestellt. Nachfolgend möchte ich auf die mathematischen Grundlagen dieses Verfahrens eingehen.

1.1 Grundlagen des RSA Verfahrens

Die grundlegende Idee eines Public Key Cryptosystems ist es, zwei Funktionen E *'Encrypt'* und D *'Decrypt'* zu erzeugen, wobei E veröffentlicht werden kann, ohne das dadurch die Funktion D bekannt wird, bzw. in endlicher Zeit berechnet werden kann.

1.1.1 Der grundsätzliche Ver- und Entschlüsselungsvorgang

Ist $\mathcal{N}$ die Menge der Nachrichten, $N \in \mathcal{N}$ die zu verschlüsselnde Nachricht, $E(N) = N'$ die verschlüsselte und $D(N') = N$ die entschlüsselte Nachricht, so versteht man unter der Verschlüsselung die Abbildung

$$E : \mathcal{N} \longmapsto \mathcal{N}$$

und unter der Entschlüsselung die Abbildung

$$D : \mathcal{N} \longmapsto \mathcal{N}$$

Weiterhin muß die Reihenfolge der Ausführung dieses Verfahrens unbedeutend sein. Dies ist besonders für digitale Unterschriften eine wichtige Eigenschaft.

$$D(E(N)) = N = E(D(N))$$

Betrachtet man diese Eigenschaft näher, so kann man leicht erkennen, daß D die inverse Funktion von E darstellt. Auf diesen Zusammenhang komme ich später noch zu sprechen.

1.1.2 Austausch von Nachrichten

Teilnehmer A möchte eine Nachricht N verschlüsseln und an den Teilnehmer B verschicken, um so ein "Mithören" Dritter zu vermeiden. Dazu ist folgendes Vorgehen notwendig:
Der Teilnehmer A ermittelt aus dem öffentlichen Schlüsselverzeichnis, das mit einem Telefonbuch vergleichbar ist, den öffentlichen Schlüssel E_B von B und verschlüsselt damit seine Nachricht.

$$N \longmapsto N' = E_B(N)$$

Diese verschlüsselte Nachricht wird dann im Rechnernetz übertragen. Weil die Umkehrung der Verschlüsselung nur dem Empfänger bekannt ist, kann ein "Lauscher" den Inhalt der Nachricht nicht bestimmen oder verfälschen.
Der Empfänger bekommt dann die Nachricht und verwendet seinen geheimen Schlüssel D_B um die ursprüngliche Nachricht zu erhalten:

$$D_B(E_B(N)) = D_B(N') = N$$

1.1.3 Die digitale Unterschrift

Die Verwendung eines Public Key Cryptosystems erlaubt auch sogenannte *digitale Unterschriften*. Diese Methode erlaubt die zweifelsfreie Autorisierung des Senders einer Nachricht. Diese Möglichkeit stellt einen der wichtigsten Vorteile gegenüber den symmetrischen Verschlüsselungsverfahren dar, weil sie auch verbindliche Unterschriften erlaubt.
Bei diesem Verfahren kann[1] folgende Vorgehensweise verwendet werden:
Der Teilnehmer A verschlüsselt seine Nachricht mit dem öffentlichen Schlüssel von B, den er aus dem Schlüsselverzeichnis erhalten hat, und die gleiche Nachricht mit seinem geheimen Schlüssel. Dieses Nachrichtenpaar wird dann verschickt.

$$(E_B(N), D_A(N)) = (N_1, N_2)$$

Der Empfänger bekommt dieses Datenpaar und wertet den Teil $E_B(N)$ mit seiner geheimen Funktion D_B aus. Dieser Teil der Nachricht ist noch nicht ausreichend für eine Identifizierung des Senders, weil der öffentliche Schlüssel E_B für alle Teilnehmer in der Schlüsselverteilzentrale, bzw. im Schlüsselverzeichnis erhältlich ist. Da in dieser Nachricht N der Name des Senders enthalten sein muß, kann nun der öffentliche Schlüssel des Senders aus dem Schlüsselverzeichnis ermittelt werden. Sodann verschlüsselt der Empfänger mit der erhaltenen Funktion E_A den empfangenen Datenblock $D_A(N) = N_2$. Ist das Ergebnis dieser Berechnung das gleiche wie bei der ersten Berechnung, ist die Identität des Senders zweifelsfrei bestätigt. Denn D_A hat die Eigenschaft, daß $\forall\ N \in \mathcal{N}$ $E_A(D_A(N)) = N$ ist. Dazu muß natürlich durch die Schlüsselverteilzentrale gewährleistet sein, daß verschiedene Teilnehmer nicht denselben Schlüssel bekommen. Dabei bleibt das "Restrisiko", daß dür ein gewisses $N \in \mathcal{N}$ trotzdem $E_A(D_{A'}(N)) = N, A \neq A'$.

1.2 Die Schlüsselgenerierung

Das RSA-Verfahren arbeitet in Restklassenringen $\mathcal{Z}_n$ bzw. $\mathcal{Z}_{\varphi(n)}$. Deshalb gilt für alle verwendeten Zahlen $0 \leq x < n$.

[1] Die tatsächliche Methode wird von Herrn Jürgen Weinelt in einem späteren Kapitel erläutert

1.2.1 Restklassenringe

Unter einer Restklasse versteht man eine Menge von natürlichen Zahlen, die bei der Division durch eine andere natürliche Zahl den gleichen Rest lassen. Diese Zahlen werden dann einer Restklasse zugeordnet.
Beispiel:
Die Zahlen $\{1, 6, 11, 16, 21, 26, 31, 36, \ldots\}$ lassen bezüglich der Teilung durch 5 alle den gleichen Rest, d.h sie bilden eine Restklasse. Die Menge aller Restklassen mod m bezeichnet man kurz mit $\mathcal{Z}_m$. Definiert man zwei Verknüpfungen $+$ und $*$, die den normalen bekannten Operationen $+$ und $*$ entsprechen, so entsteht ein kommutativer Ring mit Einselement. Deshalb gelten in einem solchen Restkassenring alle bekannten Rechenregeln, wie das Distributiv- und Assoziativgesetz. Die Ausnahme bildet die Division $\frac{a}{b} = a * b^{-1}$, weil in einem Ring nicht alle Elemente bzgl. $*$ invertierbar sein müssen. D.h. es existiert nicht für alle $x \in \mathcal{Z}_m$ ein inverses Element x^{-1}.
Zur Berechnung in diesen Ringen ist es noch wichtig zu wissen, daß man wie mit natürlichen Zahlen rechnet und erst dann das Ergebnis einer Restklasse zuordnet. Dabei soll $\overline{a}$ die Zuordnung von a zu einer Restklasse bedeuten.
Beispiel:

$$\overline{x} + \overline{y} = \overline{x + y} \quad \overline{x} * \overline{y} = \overline{x * y} \quad x, y \in \mathcal{N}$$

D.h. im Restklassenring mod 5 lautet das Ergebnis der Addition $4 + 3 = 7 = 2$. Weil die Zahlen 7 und 2 der gleichen Restklasse angehören. Da $\mathcal{Z}_n$ ein endlicher Ring ist, können die Verknüpfungen $+$ und $*$ auch durch Gruppentafeln definiert werden.

Beispiel für einen Restklassenring mod 5

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

*	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

1.2.2 Die Einheitengruppen

Für die Einheitengruppe $\mathcal{R}^*$ gilt:

$$\mathcal{R}^* = \{x \in \mathcal{R} | x \; ist \; invertierbar\} \text{ also } x \in \mathcal{R}^* \Rightarrow \exists \; y : xy = yx = 1$$

Es zeigt sich, daß x dann zur Einheitengruppe $\mathcal{R}^*$ gehört, wenn x teilerfremd zu n ist, d.h wenn x in $\mathcal{R}$ invertierbar ist.

$$\mathcal{Z}_n^* = \{x \in \mathcal{Z}_n | ggT(x, n) = 1\}$$

Beweis:
In jedem euklidischen Ring existiert eine lineare Darstellung des ggT

$$ggT(x, n) = 1 \Leftrightarrow ax + bn = 1$$

Es gilt:

$$\begin{aligned} ax + \underbrace{bn}_{=0} &= 1 \quad mod \; n \\ \Rightarrow \quad ax &= 1 \quad mod \; n \\ \Rightarrow \quad a &= x^{-1} \quad mod \; n \end{aligned}$$

Deshalb ist x genau dann invertierbar, wenn ggT(x,n)=1 ist.

1.2.3 Die Eulersche φ-Funktion

Unter der Eulerschen φ-Funktion versteht man die Abbildung $\varphi : \mathcal{N} \longmapsto \mathcal{N}$ mit:

$$\varphi(n) = n * \prod(1 - \frac{1}{p}),\ wobei\ p\ \in\ \mathcal{P}\ \wedge\ p|n$$

$\varphi(n)$ ist die Anzahl der invertierbaren Elemente in $\mathcal{Z}_n$, d.h. die Anzahl der Elemente in der dazugehörigen Einheitengruppe $\mathcal{Z}_n^*$.
Beispiel:
In $\mathcal{Z}_{15}$ sind 8 Zahlen invertierbar:

$$\mathcal{Z}_{15}^* := \{1, 2, 4, 7, 8, 11, 13, 14\}$$

Was durch einfaches Ausprobieren leicht nachgeprüft werden kann.
Die Zahl 15 hat die Primfaktorisierung $15 = 3 * 5$. Deshalb berechnet sich

$$\varphi(15) = 15 * (1 - \frac{1}{3}) * (1 - \frac{1}{5}) = 8.$$

Sollte sich, wie beim RSA-Verfahren, n nur aus zwei Primfaktoren $p \neq q$ zusammensetzen, so kann $\varphi(n)$ auch durch

$$\varphi(n) = \varphi(pq) = p * q * (1 - \frac{1}{p}) * (1 - \frac{1}{q}) = (p-1) * (q-1),\ wobei\ p, q \in \mathcal{P}$$

errechnet werden. Diese vereinfachte Berechnung der Eulerschen φ-Funktion ist besonders für die Schlüsselerzeugung von besonderer Bedeutung, weil dort die φ-Funktion von Zahlen dieses Typs ermittelt werden muß.

1.2.4 Die Schlüsselgenerierung

Die eigentliche Schlüsselgenerierung ist dann einfach.

- Wähle zwei große Primzahlen p und q und berechne $n = p * q$
  Um eine ausreichende Sicherheit zu gewährleisten sollten die Zahlen p und q ca. 100 Stellen lang sein und sich in ihrer Länge um einige Dezimalstellen unterscheiden.[2]
  Wird dies nicht beachtet oder werden die Primzahlen zu klein gewählt, so kann das Verfahren durch die Primfaktorisierung der Zahl n "geknackt" werden. Der schnellste hierzu bekannte Algorithmus hat einen Aufwand von

  $$Aufwand = ln(n)^{\sqrt{\frac{ln(n)}{ln(ln(n))}}}\ Operationen$$

  Bei einer 200-stelligen Zahl würde die Faktorisierung $\approx 1.2 * 10^{23}$ Operationen benötigen. Sollte dazu ein normaler 1 MIPS-Rechner verwendet werden, würde dies $\approx 4 * 10^9$ Jahre Rechenzeit benötigen. Dabei ist zu beachten, daß eine Implementation auf Softwarebasis weitaus mehr Operationen für einen Rechenschritt durchführen muß, weil z.B. eine 200-stellige Addition auf einem gewöhnlichen Rechner weitaus mehr Teiloperationen zur Abarbeitung braucht. Deshalb ist der reale Aufwand im Wirklichkeit um einige Größenordnungen höher.
  Die Forderung, daß sich die Primzahlen p und q um einige Dezimalstellen unterscheiden sollen, hat folgende Begründung:

[2] In dieser Studienarbeit konnte aufgrund von Rechenzeitproblemen nur mit erheblich kleineren Primzahlen gearbeitet werden.

Sollten die Primzahlen zu nahe beieinander liegen, so existiert ein effizienter Algorithmus zur Bestimmung der Primfaktoren:
Ansatz:

$n = (a+b)(a-b) = a^2 - b^2$, d.h suche ein a für das $a^2 - n$ eine Quadratzahl. Wenn b^2 keine Quadratzahl ist, dann inkrementiere a und führe das Verfahren noch einmal durch.
Beispiel: n=2773 $\rightarrow \lceil \sqrt{n} \rceil = 53 = a$
$a^2 - n = 36 \Rightarrow b = 6$
$n = (53+6)(53-6) = 59 * 47$.
Weil die zu erzeugenden Primzahlen sehr groß sein müssen, versagen klassische Verfahren wie das "Sieb des Erathostenes" aufgrund von Speicherproblemen und mangelnder Rechenzeit. Deshalb mußte der Miller-Rabin-Test ($\longrightarrow$ 2.1.2) verwendet werden.

- Berechne die Eulersche φ-Funktion von n mit $\varphi(n) = (p-1)(q-1)$
- Wähle ein e $\in \mathcal{Z}_{\varphi(n)}$, sodaß $ggT(e, \varphi(n)) = 1$.
  Da p,q $\in \mathcal{P} \wedge p, q > 2$ sind beide Zahlen ungerade. Deshalb sind die Zahlen $p-1$ und $q-1$ gerade.

$$\begin{array}{rrcl} \Rightarrow & p-1 & = & 2p' \\ \Rightarrow & q-1 & = & 2q' \\ \Rightarrow & \varphi(n) & = & 4p'q' \Rightarrow 4|\varphi(n) \end{array}$$

Aus diesem Grund darf e nie durch 2 oder 4 teilbar sein.
Wählt man einfach eine Zufallszahl, so ist die Wahrscheinlichkeit das e und $\varphi(n)$ teilerfremd sind $\frac{6}{\pi^2} \approx 0.6079$. Auf einen Beweis wird hier verzichtet, weil er keinen Nutzen zur Auffindung von e beiträgt.
Der Einfachheit halber wählt man deshalb $e \in \mathcal{P}$mit $e \not| \varphi(n)$, weil dann die Bedingung $ggT(e, \varphi(n)) = 1$ immer erfüllt ist. Man erhält den Dechiffrierungsschlüssel d mit $ed \equiv 1 \ mod \ \varphi(n)$, der sich aus der Lineardarstellung des ggT

$$ex + \varphi(n)y = 1 \ (Euklid - Berlekamp)$$

als $d := x \ MOD \ \varphi(n)$ ergibt. Jetzt müssen die Zahlen p,q und $\varphi(n)$ gelöscht werden, weil sonst ein Angreifer, der d ermitteln möchte, den Schlüssel aus p und q bzw. $\varphi(n)$ berechnen könnte.

1.3 Die Ver- bzw. Entschlüsselung

Beim RSA-Verfahren muß die zu verschlüsselnde Nachricht N eine ganze Zahl ($0 \leq N < n$) sein, d.h $N \in \mathcal{Z}_n$. Dies erzeugt einige Schwierigkeiten, wie später noch erwähnt wird. Die Ver- bzw. Entschlüsselung geschieht dann folgendermaßen:

$$E(N) = N^e \ mod \ n$$

und

$$D(N) = N^d \ mod \ n$$

Um eine solche Verschlüsselung rückgängig machen zu können, ist es notwendig, daß E(N) und D(N) folgende Eigenschaften besitzen:

$$D(E(N)) = N = E(D(N))$$

Deshalb ist zu zeigen:

$$N^{ed} \equiv N \ (mod\ n)$$

Satz:

$$a \equiv b\ mod\ p\ \wedge\ a \equiv b\ mod\ q \Rightarrow a \equiv b\ mod\ pq$$

Beweis:

$a \equiv b\ mod\ p \Rightarrow p|(a-b) \Rightarrow a - b = rp$
$a \equiv b\ mod\ q \Rightarrow q|(a-b) \Rightarrow a - b = sq$
$\Rightarrow rp = sq \Leftrightarrow r = \frac{sq}{p} \Rightarrow p|sq$
Weil p $\neq$ q (Bedingung beim RSA-Verfahren)
$\Rightarrow sq = s_1 * s_2 * s_3 * .... * s_j * p$ (Primfaktorzerlegung von sq)
Deshalb gilt: $p|s \Rightarrow r = \frac{s}{p}q$
$\Rightarrow tp = s$
$\Rightarrow a - b = sq = tpq$
$\Rightarrow pq|(a-b) \Rightarrow a \equiv b\ mod\ pq$

Deshalb kann der Beweis $E(D(N)) = N = D(E(N))$ in zwei Teilbeweise aufgeteilt werden.
$N^{ed} \equiv N\ mod\ n$ mit $n = pq$
wird in

$$N^{ed} \equiv N\ mod\ p$$

und

$$N^{ed} \equiv N\ mod\ q$$

aufgeteilt.
Beweis:

1. Fall $p|N \Rightarrow N \equiv 0$ mod p, weil $p|(N-0)$ deshalb $N^{ed} \equiv N \equiv 0$ mod p

2. Fall $p \not| N$ (Dies ist der häufigste Fall bei der Anwendung des RSA-Verfahrens)
   Lineardarstellung des ggT: $ggT(e, \varphi(n)) = 1$
   $\Rightarrow ed - a\varphi(n) = 1 \Rightarrow ed = 1 + a\varphi(n)$

$$\begin{array}{rrcccc}
 & N^{ed} & \equiv & N & mod & p \\
\Leftrightarrow & N^{1+a\varphi(n)} & \equiv & N & mod & p \\
\Leftrightarrow & N * N^{a\varphi(n)} & \equiv & N & mod & p \\
\Leftrightarrow & N * N^{\varphi(n)^a} & \equiv & N & mod & p \\
\Leftrightarrow & N * \underbrace{N^{p-1^{q-1^a}}}_{=1} & \equiv & N & mod & p \\
\Leftrightarrow & N & \equiv & N & mod & p
\end{array}$$

Für $N^{ed} \equiv N$ mod q wird der Beweis analog geführt.
Die Beziehung $N^{p-1^{q-1^a}} = 1$ ergibt sich aus dem "Satz von Euler":

$$A^{p-1}\ mod\ p = 1$$

Aufgrund der Kommutativität der Multiplikation ($ed = de$) ist auch gewährleistet, daß die Beziehung D(E(N))=N=E(D(N)) gilt.

Kapitel 2

Spezialroutinen

In diesem Kapitel möchte ich auf einige Algorithmen eingehen, die in der alltäglichen Programmierpraxis normalerweise nicht benötigt werden, aber bei der Implementierung des RSA-Verfahrens unverzichtbar sind.

2.1 Primzahltests

2.1.1 Klassische Verfahren

Bei der Verschlüsselung von Daten mit Hilfe des RSA-Verfahrens werden zur Erzeugung des öffentlichen bzw. des geheimen Schlüssels sehr große Primzahlen benötigt. Die üblichen Methoden wie "Sieb des Erathostenes" versagen hier. Um dies zu veranschaulichen, möchte ich folgendes Beispiel geben:
Um 20-stellige Primzahlen mit dem "Sieb des Erathostenes" zu berechnen bräuchte man mindestens einen Speicherplatz von $10^{20} * 1\ Bit \approx 2 * 10^{13}\ MB$, wenn man jeder Zahl ein 1 Bit zuordnet. Dabei wird jede Zahl durch ihren Index repräsentiert und die Zustände "gestrichen" bzw. "nichtgestrichen" durch das Setzen oder Löschen dieses Bits.
Weiterhin wäre die Speicherausnutzung nicht besonders hoch. Für große n läßt sich die Anzahl der Primzahlen, die kleiner n sind, näherungsweise durch folgende Beziehung wiedergeben:

$$A_p(n) = \frac{n}{ln(n)}$$

Deshalb ergibt sich eine durchschnittliche Primzahldichte von:

$$D(n) = \frac{A_p(n)}{n} = \frac{1}{ln(n)}$$

Bei einer Suche bis zu 20-stelligen Primzahlen würde sich eine Speicherausnutzung von 2% ergeben. Bei 100stelligen Primzahlen sogar nur von 0.4%. Selbst wenn eine Primzahl nur 2 Bytes belegen würde, wäre ein Speicherbedarf von $\frac{2*10^{20}}{20*ln(10)} \approx 5 * 10^{12}\ MB$ notwendig, um alle Primzahlen zu speichern. Dies verdeutlicht, daß es unsinnig wäre alle Primzahlen $< 10^{20}$ zu speichern, um so große Primzahlen zu berechnen.
Aber neben diesem Speicherproblem gäbe es auch noch das Rechenzeitproblem zu lösen, wenn das "Sieb des Erathostenes" verwendet werden würde. Eine grobe Abschätzung dieses Problems ergibt sich durch folgende Überlegung:

Im ersten Schritt sind $\frac{n}{2}$ Zahlen zu streichen. Im zweiten sind dann $\frac{n}{3}$ zu streichen. Bei j Durchläufen sind genau

$$S = n * \sum_{i=1}^{j} \frac{1}{p_i},\ p_i \in \mathcal{P} \wedge p_i \leq \sqrt{n}$$

Streichungen zu machen. Die oben angegebene Summe ist divergent. D.h der Aufwand der Streichungen erreicht nie eine Obergrenze.
Um dies zu zeigen, beginnen wir mit der Zetafunktion[1] $\zeta(s)$. Sie ist definiert als

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}$$

Wie man leicht erkennen kann, liegt bei $\zeta(1)$ ein Pol vor, weil die so entstehende harmonische Reihe divergent ist.
Desweiteren wird hier die *Eulersche Produktdarstellung der ζ-Funktion* verwendet, die durch Reihenentwicklung der einzelnen Faktoren in die oben angegebene Form der ζ-Funktion übergeht.

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s} = \prod_{p} \frac{1}{1 - \frac{1}{p^s}},\ p \in \mathcal{P}$$

Denn ein Faktor der *Eulersche Produktdarstellung der ζ-Funktion* kann in eine geometrische Reihe entwickelt werden:

$$\frac{1}{1 - (\frac{1}{p})^s} = 1 + \frac{1}{p^s} + \frac{1}{p^{2s}} + \frac{1}{p^{3s}} + \dots$$

deshalb kann das Produkt wie folgt geschrieben werden:

$$\begin{aligned} \prod_{p} \tfrac{1}{1-(\frac{1}{p})^s} \quad = \quad & (1 + \tfrac{1}{2^s} + \tfrac{1}{2^{2s}} + \tfrac{1}{2^{3s}} + \dots) \\ & *(1 + \tfrac{1}{3^s} + \tfrac{1}{3^{2s}} + \tfrac{1}{3^{3s}} + \dots) \\ & *(1 + \tfrac{1}{5^s} + \tfrac{1}{5^{2s}} + \tfrac{1}{5^{3s}} + \dots) \\ & \vdots \\ & *(1 + \tfrac{1}{p_i^s} + \tfrac{1}{p_i^{2s}} + \tfrac{1}{p_i^{3s}} + \dots) \end{aligned}$$

Weil jede natürliche Zahl eindeutig in Primfaktoren zerlegt werden kann, erzeugt der oben angegebene Ausdruck genau die Definitionsgleichung der ζ-Funktion.
Da $\lim_{s \to 1} \zeta(s) = \infty$ ist auch $\lim_{s \to 1} ln(\zeta(s)) = \infty$.
Deshalb gilt:

$$ln(\zeta(s)) = ln(\prod_{p} \frac{1}{1 - \frac{1}{p^s}}) = \sum_{p} ln(\frac{1}{1 - \frac{1}{p^s}})$$

Der Ausdruck $ln(\frac{1}{1-p^{-s}})$ kann in eine logarithmische Reihe entwickelt werden:

$$ln(\zeta(s)) = \sum_{p} \sum_{n=1}^{\infty} \frac{p^{-n*s}}{n}$$

die Abspaltung des ersten Glieds ergibt:

$$ln(\zeta(s)) = \sum_{p} p^{-s} + \sum_{p} \sum_{n=2}^{\infty} \frac{p^{-n*s}}{n}$$

[1] ζ-Funktion nach B. Riemann

Für den Logarithmus der ζ-Funktion gilt: $\lim_{s\to 1} ln(\zeta(s)) = \infty$. Kann gezeigt werden, daß $\lim_{s\to 1} \sum_p \sum_{n=2}^{\infty} \frac{p^{-ns}}{n}$ konvergiert, muß $\lim_{s\to 1} \sum_p p^{-s}$ divergieren. Folgende Abschätzung zeigt, daß $\lim_{s\to 1} \sum_p \sum_{n=2}^{\infty} \frac{p^{-ns}}{n}$ konvergiert.

$$\begin{aligned} & \sum_p \sum_{n=2}^{\infty} \tfrac{1}{n*p^{n*s}} && < && \sum_p \sum_{n=2}^{\infty} \tfrac{1}{p^{n*s}} \\ \Rightarrow\ & \sum_p (\tfrac{1}{(1-p^{-s})p^{2s}}) \\ \Rightarrow\ & \sum_p \tfrac{1}{p^s(p^s-1)} && < && \sum_p \tfrac{1}{p(p-1)} \\ \Rightarrow\ & \sum_p \tfrac{1}{p(p-1)} && < && \sum_{n=2}^{\infty} \tfrac{1}{n(n-1)} \leq 1 \end{aligned}$$

Deshalb muß die Reihe $\sum \frac{1}{p} = \lim_{s\to 1} \sum_p p^{-s}$ divergieren.
Bei einem Rechnerexperiment mit 10000 Primzahlen erreichte diese Summe den Wert von ≈ 2. Nehmen wir bei unserer Abschätzung diesen Wert als Obergrenze der Summe an, so erhält man als Untergrenze einen Aufwand von $2 * 10^{20}$ Streichungen. Hier sieht man deutlich, daß dies nicht in endlicher Zeit zu lösen wäre. Aus diesen Gründen muß ein Verfahren gefunden werden, daß hinreichend schnell große Primzahlen finden kann. Dabei darf sich der Algorithmus nicht auf alle kleineren Primzahlen stützen, weil sonst wieder das oben beschriebene Speicherproblem auftreten würde.

2.1.2 Miller-Rabin-Test

2.1.2.1 Allgemein

Ein Primzahltest liefert normalerweise eine Aussage der Form "n ist prim" oder "n ist nicht prim". Ein "probabilistischer Test" oder "Monte-Carlo Test" hingegen sagt aus: "n ist mit hoher Wahrscheinlichkeit eine Primzahl".
Die Wahrscheinlichkeit w, daß n doch zusammengesetzt ist, kann als äußerst klein betrachtet werden. Der folgende Test basiert auf der Methode von Gary Miller, mit Änderungen von Michael Oser Rabin.

2.1.2.2 Grundlage

Es gelte $n - 1 = 2^k * u$, u ungerade. Wenn für m zufällig gewählte Werte $a < n$ die Bedingung $a^u \ mod\ n = 1$ oder $a^{2j*u} \ mod\ n = n - 1$ für ein j mit $0 \leq j < k$ erfüllt sind, kann man mit einer Wahrscheinlichkeit von bis zu 4^{-m} die Zahl n als Primzahl ansehen.

2.1.2.3 Durchführung

Zunächst bringt man $n-1$ in die Form $2^k * u$ und bildet eine gewisse Anzahl von Zufallszahlen a. Für jede ist der oben angegebene Test durchzuführen. Zwei Bedingungen sind gegeben, nur eine davon muß erfüllt sein.

1. Gilt $a^u \ mod\ n = 1$ dann hat das aktuelle a den Test erfüllt und die Primwahrscheinlichkeit wächst.

2. Gilt $a^u \ mod\ n = 1$ nicht, dann ist zunächst zu prüfen, ob $a^u \ mod\ n = n-1$. Ist das nicht der Fall, wird mit $a^{2*u} \ mod\ n = n-1$ oder $a^{2^{2*u}} \ mod\ n = n-1$ usw. geprüft.

Ist nur eine dieser Gleichungen wahr, ist der Test positiv abgeschlossen, andernfalls folgt eindeutig, daß n nicht prim sein kann.

Der Test beruht darauf, daß mindestens eine der Bedingungen für jede Primzahl gilt, für eine zusammengesetzte Zahl jedoch nur mit einer Wahrscheinlichkeit von 25% (1/4). Führt man den Test einmal mit Erfolg durch, bedeutet das eine Wahrscheinlichkeit von 75% (3/4), daß n prim ist. Nach dem nächsten erfolgreichen Durchlauf erhöht sich die Fehlerwahrscheinlichkeit auf 1/16, dann weiter auf 1/64 usw. Hat eine Zahl alle Durchläufe (z.B. 30) bestanden, dann ist sie mit einer Fehlerwahrscheinlichkeit von $4 * 10^{-30} = 8.67 * 10^{-19}$ keine Primzahl. Diese Wahrscheinlichkeit liegt unter der Fehlerwahrscheinlichkeit von Computersystemen.

2.1.2.4 Optimierung des Verfahrens

Zeiteinsparung läßt sich erreichen, wenn die Rechenroutine alte Ergebnisse ausnutzt. Zunächst wird der Variablen w der Wert $a^u \ mod \ n$ zugewiesen und bei $w \neq 1$ auf w^2 gesetzt. Die Zufallsfolge hat als Startwert $a = 2$ und berechnet sich bei jedem Schleifendurchlauf mit $a^2 - 1 \ mod \ n$.

2.1.2.5 Schlußbemerkung zum Miller-Rabin Verfahren

Besonders Zahlentheoretiker begegnen Monte-Carlo Tests mit Skepsis, da Sätze die sich aus dieser Primeigenschaft einer Zahl ergeben mit einer Fehlerwahrscheinlichkeit versehen werden müßten. Für praktische Belange wie z.B. RSA-Verfahren genügt dies völlig.

2.2 Der Berlekampalgorithmus

Soll der ggT[2] zweier Zahlen berechnet werden, so verwendet man den euklidischen Algorithmus[3]. Dieser bekannte Algorithmus liefert den ggT zweier natürlichen Zahlen.

Da aber eine Lineardarstellung des ggT zur Bestimmung des geheimen Schlüssels d gebraucht wird, muß der Berlekampalgorithmus verwendet werden. Dieser Algorithmus stellt eine leichte Abwandlung des euklidischen Algorithmus dar, der als Nebenprodukt noch die in jedem euklidischen Ring existierende lineare Darstellung des ggT liefert. Der Algorithmus läßt sich durch folgenden Pseudocode beschreiben:

hole die Zahl a;
hole die Zahl b;
IF ($a < b$) THEN
  tausche die beiden Zahlen
ENDIF;
r_{n-2}:=a; r_{n-1}:=b;
a_{n-2}:=1; a_{n-1}:=0;
b_{n-2}:=0; b_{n-1}:=1;
r:=1;
n:=0;
WHILE r#0 DO
  q:=r_{n-2} DIV r_{n-1};
  r:=r_{n-2} MOD r_{n-1};
  a:=q*a_{n-1}+a_{n-2};
  b:=q*b_{n-1}+b_{n-2};
  r_{n-2}:=r_{n-1}; r_{n-1}:=r;

[2] größter gemeinsamer Teiler
[3] nach Euklid um 300 v. Chr.

a_{n-2}:=a_{n-1}; a_{n-1}:=a;
b_{n-2}:=b_{n-1}; b_{n-1}:=b;
n:=n+1;
ENDWHILE;
IF n gerade THEN
a:=-a
ELSE
b:=-b
ENDIF;
r_{n-2} ist nun der ggT der Zahlen a und b;
in a_{n-2} und b_{n-2} stehen die Komponenten der linearen Darstellung des ggT.

Das Ergebnis dieser Berechnung läßt sich durch folgende Beziehung beschreiben:

$$r_{n-1} = ggT(a,b) = (-1)^{n-1} a_{n-1} a + (-1)^n b_{n-1} b$$

Kapitel 3

Implementierung der Rechenbibliothek

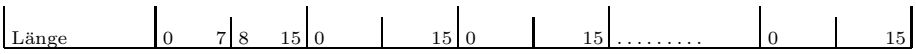
Da Modula2 in Bezug auf die Länge der darstellbaren ganzen Zahlen beschränkt ist, mußte eine Ausweichlösung für dieses Problem gefunden werden. Die meisten Zahlendarstellungen bewegen sich im Bereich von -32768 bis 32767 oder von -2147483648 bis 2147483647, je nach dem welche Datenbusbreite die verwendete CPU besitzt. Diese Zahlenbereiche sind für das RSA-Verfahren aber bei weitem nicht ausreichend, weil hier Zahlen von ca. 200 Dezimalstellen gebraucht werden, um eine ausreichende Sicherheit zu gewährleisten.
Um ein Problem wie das RSA-Verfahren angehen zu können, mußte ein Modul implementiert werden, das mit solch langen Zahlen umgehen kann. Die dabei auftretenden Probleme sollen in diesem Kapitel erläutert werden.

3.1 Die interne Zahlendarstellung

Die einfachste Methode wäre ein ARRAY OF CHAR zu vereinbaren und darin die Zahl als ASCII-Zeichenketten darzustellen. Dies hätte auch den Vorteil, daß man keine Umwandlungsprobleme von String nach LONG-Zahl und zurück hätte. Nebenbei wäre es einfach die Grundrechenarten zu programmieren, weil sie ja den vertrauten Rechenoperationen entsprechen würden.
Bei genauerer Untersuchung dieses Problems entpuppt sich diese Darstellung aber als unbrauchbar, weil Überträge nur durch den Vergleich einzelner Komponenten abprüfbar wären. Dies ist sehr zeitaufwendig, weil die elementaren Vergleichsoperatoren zur Abbildung der Grundrechenarten häufig gebraucht werden.
Die binäre Darstellung hingegen könnte jeden Übertrag nur durch Abfragen eines einzelnen Bits erkennen. Dabei stellt die LONG-Zahl ein sehr langes Bit-ARRAY dar. Einen weiteren Vorteil stellt die bessere Speicherausnutzung durch die binäre Darstellung dar.
Es stellt sich nun die Frage "Wie bildet man die Grundrechenarten auf ein langes Bit-ARRAY ab ? ".
Dazu muß man erst einmal einige grundsätzliche Überlegungen anstellen. Weil Modula2 keinen beliebig langen SET-Typ[1] kennt, muß ein ARRAY OF INTEGER verwendet werden.
Während der Rechenoperationen kann es vorkommen, daß nicht alle Komponenten des Integerarrays mit gültigen Daten gefüllt sind. Wenn man sich die

[1] Beschränkung auf die Breite des Datenbusses des verwendeten Systems

Anzahl der aktuell gültigen Komponenten nicht merkt, müssen alle unbenutzen Teile der LONG-Zahl mit 0 vorbesetzt werden. Weiterhin hätte diese Lösung den Nachteil, daß man die Operationen auch mit unbenutzten Teilen der LONG-Zahl durchführen müßte, was eine wesentliche Verlangsamung der Berechnungen verursachen würde. Aus diesem Grund habe ich mich dazu entschieden im ersten Element die Anzahl der gültigen Komponenten abzulegen.
Jetzt muß nur noch die Erkennung des Überlaufs bei einer Teiloperation realisiert werden. Die erste Idee einfach das Carry-Bit abzufragen scheitert daran, daß Modula2 keine Möglichkeit zur Verfügung stellt das Carry-Bit direkt abzufragen. Deshalb habe ich einen kleinen Trick verwendet. Bei meiner Zahlendarstellung wird das Vorzeichenbit als Pseudo-Carrybit verwendet. Dies hat den Vorteil, daß der Compiler effizienten Code zur Überlaufabfrage erzeugen kann, weil ja nur das Negativbit abgefragt werden muß. Dies kann mit so einfachen Konstrukten geschehen wie "IF $i < 0$ THEN ...". Solch eine Abfrage kann direkt in Maschinensprache (BMI oder BPL beim MC 68000) übersetzt werden. Das nächste Bild soll die verwendete Zahlendarstellung noch einmal verdeutlichen.



Dabei sind das LSB[2] und das MSB[3] von besonderer Bedeutung.
Das MSB wird wie schon oben beschrieben, als Pseudocarrybit verwendet. Durch Testen des LSB kann entschieden werden, ob die vorliegende LONG-Zahl gerade oder ungerade ist. Dies ist bei einigen unten beschriebenen Algorithmen von besonderer Bedeutung.

3.2 Implementation der Grundrechenarten

3.2.1 Addition

Die erste und wohl einer der wichtigsten Grundrechenarten ist die Addition. Diese Operation wird zwar nicht direkt vom RSA-Verfahren benötigt, aber viele Operationen wie die Multiplikation verwenden diese Verküpfung intern.

Um die Addition auf die LONG-Zahlen abzubilden sind einige Überlegungen notwendig, denn es sind 3 Fälle zu unterscheiden. Im ersten Fall sind beide Zahlen gleich lang. Hier kann einfach Komponente mit Komponente addiert werden.
Sollte bei einer solchen Teiloperation ein Überlauf auftreten, so muß im nächsten Schritt ein Übertrag berücksichtigt werden. Sollte in der letzten Komponente noch ein Überlauf auftreten, so muß in das nächste Element eine Eins eingetragen werden, und die Länge im Feld 0 erhöht werden. Ist die maximale Länge erreicht, so wird eine BOOLEAN-Variable Carry auf TRUE gesetzt, um den Überlauf bei der Gesamtoperation anzuzeigen. Dies kann, nachdem diese Variable importiert wurde, abgefragt werden. Die zwei anderen Fälle, in denen die LONG-Zahlen ungleichlang sind, sind etwas aufwendiger zu behandeln. Leider treten sie auch in der Mehrzahl der Fälle auf, was die Gesamtleistung der Algorithmus erniedrigt. Es bieten sich zwei Möglichkeiten an:

1. Man bearbeitet die Zahlen nur bis zu der Länge der kürzesten Zahl und überträgt dann die weiteren Komponenten der längeren Zahl in das Ergebnis. Dabei müßen natürlich immer noch eventuell auftretende Überträge,

[2]Least Significant Bit = Bit 0
[3]Most Significant Bit = Bit 15

die bei der Addition der einzelnen Komponenten entstehen, berücksichtigt werden. In einem solchen Fall muß die nächste Komponente inkrementiert werden.

2. Man verlängert die kürzere Zahl künstlich in dem man vorne Komponenten mit 0 anfügt bis beide Zahlen gleich lang sind. Dann führt man den Algorithmus für gleich lange Zahlen durch.

Ich habe mich bei der vorliegenden Implementierung für den Fall 2 entschieden, weil zum einen der Programmierungsaufwand wesentlich niedriger ist, und zum anderen die Unterschiede bei den Zahlenlängen nicht besonders ausschlaggebend sind. Der Algorithmus ist als Struktogramm im Anhang wiedergegeben.

3.2.2 Subtraktion

Bei der Subtraktion treten im Wesentlichen die gleichen Probleme wie bei der Addition auf. Deshalb unterscheiden sich die beiden Algorithmen nur unwesentlich. Der einzige Unterschied ist die Abprüfung, ob der erste Operand kleiner als der zweite ist, weil dann nämlich die Carryvariable auf TRUE gesetzt werden muß. Dies ist notwendig, weil das vorliegende Modul keine negativen Zahlen unterstützt.

3.2.3 Multiplikation

Im Gegensatz zu den oben beschriebenen Operationen ist die Multiplikation etwas schwerer überschaubar. Ein erster naiver Versuch ergibt sich durch folgende Überlegung:

$$5 * 4 = 5 + 5 + 5 + 5$$

oder allgemein:

$$n * m = m * n = \underbrace{n + \ldots + n}_{genau\ m\ mal}$$

Bei dieser Vorgehensweise ist ein sehr hoher Aufwand notwendig, weil eine Additionschleife sehr oft durchlaufen werden muß, wenn m groß ist. Weil dies im vorliegenden Modul der Fall ist und beim RSA-Verfahren die oft benötigte Potenzierung auch auf diese Multiplikation zurückgreift, ist diese Art der Implementierung nicht sinnvoll und wird hier nicht verwendet.
Statt dessen wird ein Algorithmus, der schon in der Assemblervorlesung im 4. Semester vorgestellt wurde, verwendet, um das Produkt $a * b$ zu berechnen. Bei diesem Algorithmus wird die Zahl b solange rechts rotiert, d.h. die Zahl wird fortgesetzt durch 2 geteilt, bis man die Null erreicht. Sollte während der Ausführung des Algorithmus b ungerade sein, was durch Abfragen des LSB getestet werden kann, so wird eine dritte Variable um den Wert der Zahl a inkrementiert. Während des Schleifendurchlaufs wird die Zahl a mit 2 multipliziert, was einer Linksrotation entspricht. Dieser Algorithmus hat dann nur noch einen Aufwand von ld(b). D.h. bei einer Wortbreite von n sind nur n Additionen und $2 * n$ Rotationen notwendig um das Ergebnis zu berechnen. Dies ist schon ein großer Fortschritt gegenüber dem weiter oben vorgeschlagen Algorithmus.
Der Aufwand in dieser Implementation ist leicht abzuschätzen:
Wenn die Zahl b 3 INTEGER-Komponenten[4] lang ist müssen maximal $3 * 15 = 45$ Rotationen[5] durchgeführt werden, die die ganze LONG-Zahl betreffen. Hier

[4] Dies ist im Längenfeld notiert

[5] Bei dem von uns verwendeten MODULA2-Compiler wird ein 16-Bit INTEGER Typ bereitgestellt

sieht man schon, daß dieses Verfahren bei langen Zahlen immer noch sehr aufwendig ist. Mir ist aber leider kein Algorithmus bekannt, der einen Aufwand von weniger als ld(b) hat. Das Struktogramm dieses Algorithmus ist im Anhang wiedergegeben.

3.2.3.1 Restklassenmultiplikation

Da beim RSA-Verfahren nur eine Multiplikation mit Abbildung in den verwendeten Restklassenring benötigt wird, habe ich einen modifizierten Multiplikationsalgorithmus implementiert, der dies leistet.
Die erste Idee nach einer normalen Multiplikation eine MOD-Operation durchzuführen scheitert an dem zu hohen Rechenzeitbedarf, weil eine Modulofunktion Schiebe- und Subtraktionsoperationen benötigt. Weiterhin kann das Produkt zweier großer Zahlen u.U. Werte erreichen, die nicht mehr darstellbar sind.
Der hier verwendete Multiplikationsalgorithmus benötigt ein Links-Shift und eventuell eine Addition. Bei beiden Funktionen können die verwendeten Variablen die Ordnung des Restklassenrings überschreiten. Bei näherer Überlegung zeigt sich aber, daß sie immer kleiner als $2 * Ordnung$ des Restklassenrings $\mathcal{Z}_n$ sind. Deshalb muß im Fall der Überschreitung nur eine Subtraktion durchgeführt werden, um das Ergebnis wieder in den Bereich des Restklassenrings abzubilden, was eine erhebliche Rechenzeitersparnis ergibt. Ein weiterer Vorteil dieses Verfahrens ist, daß der Betrag der größten auftretende Zahl nur $2 * Ordnung$ des Restklassenrings sein kann, wodurch u.U. erheblich Speicherplatz gespart wird.

3.2.4 Division

Obwohl die Division beim RSA-Verfahren nicht als einzelne Operation benötigt wird, habe ich sie der Vollständigkeit halber im Modul MathLong implementiert.
Die Division von $\frac{a}{b}$ unterscheidet sich im Wesentlichen nicht von der Multiplikation. Der einzige wesentliche Unterschied ist der Beginn des Algorithmus. Hier muß die kleinere Zahl b solange mit 2 multipliziert werden, bis sie größer oder gleich der Zahl a ist. Dies hat schwerwiegende Auswirkungen auf die Umwandlung von LONG-Zahlen in ASCII-Strings, weil hier fortgesetzt durch 10 geteilt werden muß. Da aber 10 klein gegenüber der zu teilenen Zahl ist, sind sehr viele Schiebeoperationen notwendig. D.h. Divisionen bei denen die Zahl $b \ll a$ können nur relativ langsam berechnet werden. Auch zu dieser Funktion existiert ein Struktogramm im Anhang.

3.3 Die erweiterten Rechenoperationen

3.3.1 Die Modulofunktion

Da die Modulofunktion für das RSA-Verfahren von großer Bedeutung ist, mußte auch diese Funktion implementiert werden.
Sie existiert in zwei Varianten. Einmal als normale einzelne Modulofunktion und zum andern als mit der Division kombinierte ModDiv-Funktion. Diese Funktion bietet einen Vorteil:
An mehreren Stellen ist gleichzeitig eine Zahl durch eine andere zu teilen und der Rest der Division zu bestimmen. Da sich diese Operationen im Algorithmus nur durch die Rückgabe unterschiedlicher Variablen unterscheiden, wurden sie in einer Funktion zusammengefaßt um so erheblich Rechenzeit zu sparen.

3.3.2 Potenzierung

Diesem Algorithmus fällt ein zentrale Bedeutung beim RSA-Verfahren zu, weil sowohl die Verschlüsselung als auch die Entschlüsselung durch die Potenzierung geschieht
Da die Verschlüsselungsgeschwindigkeit ein wesentlicher Faktor bei einem Public-Key Cryptosystem ist, wurde bei der Implementierung besonders auf die Effizienz geachtet.
Eine erste naive Überlegung ergibt folgendes Ergebnis:

$$a^b = \underbrace{a * a * \ldots * a}_{genau\ b\ mal}$$

Wie man sieht werden hier sehr viele Multiplikationen benötigt, wenn b eine große Zahl ist. Da dies beim RSA-Verfahren aber der Fall ist, kann diese Methode nicht verwendet werden.
Ein ähnliche Idee wie bei der Multiplikation führt dann zur Methode "Repeated Square and Multiply", die nur noch einen Aufwand von ld(b) hat, was eine wesentliche Verbesserung darstellt.
Zur Potenzierung x^y wird eine mit 1 initialisierte Variable r immer dann mit x multipliziert, wenn x ungerade ist. Sodann wird x durch x^2 ersetzt und y um ein Bit nach rechts rotiert. Diese Routine ist die iterative Formulierung des folgenden rekursiven Ansatzes:

$$a^b = \begin{cases} a^{\frac{b}{2}} * a^{\frac{b}{2}}, & \text{falls b gerade} \\ a^{\frac{b}{2}} * a^{\frac{b}{2}} * a, & \text{falls b ungerade} \end{cases}$$

Die Aufspaltung von a^b in $a^{\frac{b}{2}}$ wird rekursiv so lange fortgeführt, bis $\frac{b}{2}$ gleich 1 ist. Die iterative Formulierung verwendet dazu die Binärdarstellung der Zahl b, die durch Schiebeoperationen gewonnen wird.
Jetzt wird auch klar warum eine Multiplikation mit anschließender MOD-Operation implementiert wurde. Durch diese Routine kann leicht eine Restklassenpotenzierung formuliert werden, mit der die Ver- bzw. Entschlüsselung einer Nachricht berechnet wird.
Um den Algorithmus zu veranschaulichen ist im Anhang ein entsprechendes Struktogamm wiedergegeben.

3.4 Typwandlungsfunktionen

Da wir eine Zahlendarstellung verwenden, die nicht von Computern mit der INTEL-CPU 80386 unterstützt wird, sind die Typwandlungsfunktionen von elementarer Bedeutung. Deshalb wurden mehrere Varianten implementiert.
Um die Verwendung von LONG-Zahlen zu vereinfachen wurden häufig verwendete Zahlen als "Konstanten" im Definitionsmodul vereinbart.
Sollten andere Zahlen verwendet werden, so existiert je eine Routinen zur Wandlung von Strings in LONG-Zahlen und eine zur Umwandlung von positiven Integerzahlen.
Auch der umgekehrte Weg der Umwandlung ist vorhanden. Mit der Routine LONG2Str kann eine LONG-Zahl in einen String umgerechnet werden.
Für die Wandlungsfunktionen von Strings nach LONG-Zahlen beziehungsweise umgekehrt wurden Standardalgorithmen verwendet.
Um einen String in eine LONG-Zahl zu verwandeln wird das Hornerschema verwendet.
Die Ergebnisvariable muß mit 0 initialisiert werden. Dann wird das erste Zeichen im Eingabestring in seine Integerzahl umgewandelt. Die Ergebnisvariable

wird dann mit 10 multipliziert [6]. Dannach wird noch die oben ermittelte Integerzahl zu der Ergebnisvariable dazuaddiert. Dieser Vorgang wird so lange wiederholt, bis das Ende des Eingabestrings erreicht ist.
Bei der Wandlung LONG-Zahl in String wird immer der Rest bezüglich der Division mit 10 bestimmt. Dieser Rest wird in sein CHAR-Äquivalent umgewandelt und in einen String eingetragen. Nach der Beendigung dieses Vorgangs wird der String noch umgedreht, damit die niedrigen Potenzen der Zahl am rechten Ende stehen.
Um die Umwandlung in eine LONG-Zahl zu vereinfachen wurde auch eine Wandlungsfunktion für positive Integerzahlen in LONG-Zahlen vorgesehen. Die Wandlung geschieht dadurch, daß in Komponente 0 der LONG-Zahl die Länge 1 eingetragen wird. In die erste Komponente wird dann nur noch die übergebene Zahl eingetragen. Diese Routine wird z.B. besonders oft vom Miller-Rabin-Test verwendet.

3.5 Die Vergleichoperationen

Bei allen Algorithmen werden arithmetische Vergleichsoperationen benötigt. Mit diesen Operationen können z.B. die Gleichheit oder Ungleichheit zweier LONG-Zahlen ermittelt werden. Alle diese Routinen liefern eine Variable vom BOOLEAN-Typ zurück.
In Anlehnung an die bekannte FORTRAN-77-Syntax wurden Namen wie "LE" Less Equal oder "EQ" Equal vergeben. Alle Vergleichsoperationen haben die gleiche Bedeutung wie ihre FORTRAN-77-Äquivalente und bedürfen deshalb keiner weiteren Erläuterung.
Ich möchte hier noch auf eine Besonderheit eingehen. Von Jürgen Weinelt wurde vorgeschlagen Vergleichsoperationen wie "LT" durch den Aufruf von "NOT(GE)" zu realisieren. Dies würde eine Verkürzung des Programmcodes bewirken, aber auch gleichzeitig eine Verlangsamung der Routinen, weil die Parameter 2 mal über den Stack übergeben werden müßten. Da bei der vorliegenden Implementierung mehr auf Laufzeiteffizienz als auf Speichereffizienz geachtet werden mußte, habe ich mich dafür entschieden jede einzelne Operationen vollständig zu programmieren.

3.6 Hilfsfunktionen

In diesem Abschnitt sollen die Hilfsfunktionen des Modules MathLong erwähnt werden.

3.6.1 Speicherfunktionen

Das wichtigste Funktionenpaar dürften die Funktionen AllocNum und FreeNum sein, die Speicher für LONG-Zahlen anfordern und freigeben. Da LONG einen opaquen Typ darstellt, ist er nach MODULA2 Konvention ein Zeiger. Deshalb muß nicht nur eine Variable von Typ LONG vereinbart werden, sondern es muß auch noch der entsprechende Speicherbereich mit AllocNum reserviert werden. Bei Programmende muß dieser Speicher mit FreeNum wieder freigegeben werden. Um LONG-Zahlen zuweisen zu können wurde noch eine Zuweisungsfunktion Trans implementiert. Dies ist nötig, weil die Zuweisung eines Zeigers auf einen anderen den Speicherbereich auf den der Quellzeiger deutet nicht kopiert. Folgende Anweisungen sollen dies verdeutlichen:

[6] Hier wurde als Basis das vertraute Dezimalsystem verwendet

```
VAR a,b:LONG
⋮
(* Falsche Zuweisung *)
a:=b
⋮
(* Richtige Zuweisung *)
MathLong.Trans(a,b)
⋮
```

3.6.2 Allgemeine Hilfsfunktionen

Um den Zahlenbereich und die Anzahl der verwendeten Bits der LONG Zahlen abfragen zu können wurden mehrere Funktionen implementiert.

1. MaxLong: Diese Funktion gibt ähnlich der MODULA2-Funktion MAX die größte verfügbare LONG-Zahl zurück. Intern werden alle Komponenten der übergebenen Zahl mit -1 belegt. Dann wird das MSB gelöscht um eine gültige LONG-Zahl zu erzeugen. Zuletzt wird noch die entsprechende Länge in die nullte Komponente eingetragen.

2. NStellen: Diese Routinen ermittelt die Anzahl der frei verfügbaren Dezimalstellen und gibt sie als Integerergebnis zurück. Die Berechnung ergibt sich durch folgende Gleichung:

$$\begin{aligned} & Stellen &=& \ log_{10}(2^{Verwendete\_Bits}) \\ \Leftrightarrow\ & Stellen &=& \ Verwendete\_Bits * ln(2)/ln(10) \end{aligned}$$

3. AktStellen: Diese Funktion gibt die Anzahl der verwendeten Dezimalstellen der übergebenen Zahl zurück.

4. NumBits: Das Ergebnis dieser Routine ist die Anzahl der verwendeten Bits. In der vorliegenden Implementierung ist dieses Ergebnis immer durch 15 teilbar, weil 15 Bits pro Integerkomponente verwendet werden.

3.7 Spezialroutinen für das RSA-Verfahren

In diesem Abschnitt gehe ich auf die Teile der MathLong Bibliothek ein, die für das RSA-Verfahren von Bedeutung sind. Da versucht wurde dieses Modul objektorientiert zu programmieren, mußten alle Wandlungsfunktionen, die die Schlüsselverteilzentrale benötigt auch in MathLong implementiert werden.

3.7.1 Fileoperationen

Die wohl wichtigste Funktion für das RSA-Verfahren ist die Verschlüsselungsfunktion für Files.
Das im Labor vorhandene Netzwerk arbeitet nach dem Prinzip der verteilten Speichermedien, d.h. ein Rechner "sieht" das Netz als Festplatte oder Diskettenlaufwerk. Deshalb braucht keine Unterscheidung zwischen File- und Netzwerkzugriffen gemacht werden.
Für die Kodierung eines Files wurde eine Routine RSACode implementiert. Diese Routine erwartet die Deskriptoren eines geöffneten Eingabe- und Ausgabefiles. Weiterhin muß das öffentliche Schlüsselpaar übergeben werden. Als

Besonderheit wird noch eine Prozedurvariable übergeben, die die Routine identifiziert, mit der eine veränderte Quellkodierung erzeugt werden kann. Im vorliegenden Projekt wird bei einem übergebenen Buchstaben nur abgefragt, ob das MSB gesetzt ist und dieses gegebenenfalls gelöscht.
Das Gegenstück dieser Routine stellt RSADeCode dar. Der Aufruf dieser Routine ist mit RSACode vergleichbar. Der einzige Unterschied zeigt sich darin, daß man des geheime Schlüsselpaar übergeben muß. Auch eine Prozedurvariable zur Dekodierung der geänderten Quellkodierung ist vorhanden. Diese wird aber in der vorliegenden Implementierung nicht genutzt.

3.7.2 Blockkodierungsroutinen

Die Schlüsselverteilzentrale muß bei der Herstellung von digitalen Unterschriften manche Datenblöcke zweimal verschlüsseln. Da man dies nur schlecht mit den oben beschriebenen Fileoperationen erreichen kann, wurden spezielle Blockoperationen implementiert. Diese Funktionen können Byteblöcke Ver- und Entschlüsseln um so digitale Unterschriften zu erzeugen.
Zur Verarbeitung von Byteblöcken sind die zwei Prozeduren CodeByteBlock und DeCodeByteBlock vorgesehen. Dabei stellt DeCodeByteBlock das Gegenstück zu CodeByteBlock dar. Beiden Routinen ist entweder das öffentliche oder das geheime Schlüsselpaar zu übergeben. Als Besonderheit ist zu erwähnen, daß als Quelle bzw. Ziel ein Zeiger verwendet wird. Dies ist notwendig, weil durch die Verschlüsselung Byteblöcke länger werden können und MODULA2 keine dynamischen Felder kennt. Bei beiden Routinen wurde keine Prozedurvariable für eine alternative Quellkodierung vorgesehen, weil diese Routinen ausschließlich für die internen Aufgaben der Schlüsselverteilzentrale vorgesehen sind.

3.7.3 Blockwandlungsfunktionen

Da auch Schlüssel verschickt werden[7], wurde ein Routinenpaar implementiert, daß eine Umwandlung von LONG-Zahlen in Byteblöcke und umgekehrt ermöglicht. Diesen Routinen ist ein fest definiertes Bytefeld zu übergeben, indem eine Kopie der LONG-Zahl abgelegt wird.
Zusätzlich wurden noch Möglichkeiten vorgesehen, LONG-Zahlen in Files zu schreiben bzw. zu lesen. Diese Möglichkeiten werden nicht ausgenutzt und wurden nur der Vollständigkeit halber implementiert.

3.8 Das RSA-Kernmodul

Um eine einheitliche Oberfläche für Anwendungsprogramme zu schaffen wurde ein separates Modul RSA geschaffen. In diesem Modul sind alle Routinen die das RSA-Verfahren benötigt zusammengefaßt und in ihrer Bedienung möglichst vereinfacht. So finden sich in diesem Modul die schon oben beschriebenen Routinen sowie Funktionen zur Schlüsselgenerierung und zur Umwandlung von LONG-Zahlenpaaren in Byteblöcke definierter Länge. Die letztgenannte Funktion wird wie schon erwähnt für die Schlüsselverteilzentrale benötigt.
Zur Schlüsselgenerierung wird der Miller-Rabin-Test verwendet, der von Oliver Stock implementiert wurde. Desweiteren werden die Routinen aus dem Modul LongMathLib verwendet, die die Eulersche φ-Funktion berechnen sowie den Berlekampalgorithmus durchführen.

[7] z.B wenn ein Netzteilnehmer den öffenlichen Schlüssel eines anderen Teilnehmers von der Schlüsselverteilzentrale erfragen will

3.9 Besonderheiten der Implementierung

In diesem Abschnitt will ich auf die Besonderheiten eingehen, die sich durch die MODULA2-Implementierung ergeben.
Die wohl größten Schwierigkeiten ergaben sich durch die Zahlendarstellung. Da sowohl arithmetische Operationen wie + und ∗ als auch logische wie AND, OR und NOT auf die einzelnen Komponenten angewendet werden wurden, mußte ein Weg gefunden werden, dies effizient zu lösen.
Deshalb wurde ein Feld deklariert, das aus BITSET-Elementen besteht und durch Pointerhandling an der gleichen Address liegt wie das weiter oben beschriebene INTEGER-Array. Je nachdem welche Operationen durchgeführt werden sollen, kann die eine oder die andere Darstellungsform gewählt werden.
Obwohl dies nicht den Grundgedanken der strukturierten Programmierung entspricht, habe ich mich wegen des zu erzielenden Geschwindigkeitsvorteil für diese Lösung entschieden. Dabei wurden alle Möglichkeiten ausgeschöpft, Programmierungsfehler zu vermeiden, die MODULA2 zur Verfügung stellt, und die betroffenen Codeteile möglichst klein zu halten.
Ein weiterer Trick war das Abschalten der Compileroption, die die Bereichsüberprüfung steuert. Dies war notwendig, weil bei komponentenweisen Addieren ein Überlauf einteten kann, der das Laufzeitsystem normalerweise zu einem Programmabbruch zwingt. Das Erkennen eines Überlaufs wird in dieser Implementierung durch das Abtesten des MSB[8] erreicht.

[8] Wenn das MSB gesetzt ist dann ist die entsprechende Zahl negativ

Kapitel 4

Die Schlüsselverteilzentrale

4.1 Grundlagen des Verfahrens

4.1.1 Das Problem

Jeder Kommunikationspartner $\mathcal{A}$ in einem Netz, der mit einem anderen Netzteilnehmer $\mathcal{B}$ unter Verwendung des RSA-Verfahrens verschlüsselt kommunizieren möchte, muß dazu den öffentlichen Schlüssel von $\mathcal{B}$ kennen; ebenso muß $\mathcal{B}$ wissen, wie der öffentliche Schlüssel von $\mathcal{A}$ lautet. Nach [DATACOM] gibt es folgende drei Möglichkeiten, wie die beiden Teilnehmer die Schlüssel erfahren können:

4.1.1.1 Direkter Austausch

Beim Verbindungsaufbau tauschen beide Teilnehmer ihre öffentlichen Schlüssel direkt aus.

Vorteile:

- effizient
- geringer Verwaltungsaufwand
- keine Schwierigkeiten bei Änderung des Schlüssels

Nachteile:

- fehlende Authentizitätsprüfung

4.1.1.2 lokale Schlüsselverzeichnisse

Jeder Teilnehmer muß die öffentlichen Schlüssel aller[1] anderen Teilnehmer gespeichert halten.

Vorteile:

- besonders einfacher Verbindungsaufbau
- eindeutige Authentisierung

Nachteile:

- hoher Verwaltungsaufwand
- Probleme bei Änderung des Schlüssels

[1] zumindest derer, mit denen er kommunizieren will

4.1.1.3 Schlüsselverteilzentrale

Beim Verbindungsaufbau erhalten beide Teilnehmer den Schlüssel des jeweils Anderen von einer zentralen Stelle.

Vorteile:

- Authentizitätsprüfung möglich
- geringer Verwaltungsaufwand
- keine Schwierigkeiten bei Änderung des Schlüssels

Nachteile:

- keine gravierenden Nachteile

4.1.2 Die Lösung

In der Aufgabenstellung für unsere Studienarbeit wurde die Verwendung einer Schlüsselverteilzentrale mit einseitiger Teilnehmerkommunikation vorgeschrieben, das heißt, nur der Teilnehmer $\mathcal{A}$ nimmt Verbindung mit der Schlüsselverteilzentrale $\mathcal{SVZ}$ auf.

4.2 Das Verfahren

Eine ausführliche Beschreibung des verwendeten Verfahrens findet sich in [DATACOM 3.4.4]. Grundsätzlich läuft ein Verbindungsaufbau wie folgt ab:

1. $\mathcal{A}$ schickt eine Nachricht an $\mathcal{SVZ}$, aus der hervorgeht, daß $\mathcal{A}$ mit $\mathcal{B}$ kommunizieren möchte.
2. $\mathcal{SVZ}$ antwortet mit einer Nachricht an $\mathcal{A}$, die aus zwei Zertifikaten zusammengesetzt ist; das erste Zertifikat ist für $\mathcal{A}$ bestimmt und enthält unter Anderem den öffentlichen Schlüssel von $\mathcal{B}$. Das zweite Zertifikat ist für $\mathcal{B}$ bestimmt und enthält unter Anderem den öffentlichen Schlüssel von $\mathcal{A}$.
3. $\mathcal{A}$ wertet das für ihn bestimmte Zertifikat aus und schickt das zweite an $\mathcal{B}$ weiter.
4. $\mathcal{B}$ wertet das Zertifikat aus.
5. $\mathcal{A}$ erzeugt eine Kontrollmeldung für $\mathcal{B}$.
6. $\mathcal{B}$ wertet die Kontrollmeldung von $\mathcal{A}$ aus. In diesem Moment hat $\mathcal{B}$ $\mathcal{A}$ bereits authentisiert.
7. $\mathcal{B}$ erzeugt nun eine Kontrollmeldung für $\mathcal{A}$.
8. $\mathcal{A}$ wertet die Kontrollmeldung von $\mathcal{B}$ aus. In diesem Moment hat $\mathcal{A}$ $\mathcal{B}$ ebenfalls authentisiert.

Die gegenseitige Authentisierung erfolgt durch digitale Unterschriften.

4.3 Die Implementation

4.3.1 Überblick

Sämtliche Prozeduren, die zum Betrieb der Schlüsselverteilzentrale benötigt werden, befinden sich im Modul *KeyMaster*. Da die Prozeduren, die von den Kommunikationsteilnehmern benutzt werden müssen, sich davon nur unwesentlich unterscheiden, wurden auch sie in den Modul *KeyMaster* integriert.

Der Modul *KeyMaster* importiert neben den Standard-Bibliotheksmoduln nur den Modul *RSA* von Steffen Reith. Dieser Modul enthält eine Schnittstelle zu den zur Ver– und Entschlüsselung benötigten Prozeduren auf relativ hoher Ebene.

Zur Demonstration des Verfahrens wurden außerdem vier weitere Moduln implementiert, nämlich *A*, *B*, *C* und *Schedule*. Diese demonstrieren den Ablauf eines Kommunikationsvorganges auf einem einzigen Rechner. Die Kommunikationsteilnehmer werden von verschiedenen Moduln simuliert, die Datenübertragungen über das Netz durch Zwischenspeicherung der Daten in Files. Die Moduln *A* und *B* enthalten die Steuerung eines Kommunikationsablaufs für jeweils einen der Teilnehmer $\mathcal{A}$ und $\mathcal{B}$, der Modul *C* enthält die Schlüsselverteilzentrale $\mathcal{SVZ}$, und der Modul *Schedule* aktiviert nacheinander die jeweils benötigten Routinen von *A*, *B* und *C*.

4.3.2 Datentypen

4.3.2.1 Dirty–Typen

Für alle Datenstrukturen, die $\mathcal{SVZ}$ benötigt, wurden entsprechende Datentypen deklariert. Leider konnten diese Datentypen nicht durchgehend verwendet werden, denn durch die Verschlüsselung der Daten ändert sich auch die Länge (der Speicherbedarf) dieser Objekte (⟶ 4.3.2.5). Daher wurde für die Behandlung verschlüsselter Daten ein quasi–generischer Datentyp *Dirty* deklariert; dieser Typ ist einfach ein roher, unstrukturierter Datenblock vom Typ ARRAY OF BYTE.

Sämtliche Typen und Prozeduren, die in ihrem Namen das Wort „Dirty“ enthalten, beziehen sich auf die Behandlung solcher unstrukturierter Blöcke.

4.3.2.2 Stamp

Der Datentyp *Stamp* dient zur Speicherung eines sogenannten Zeitstempels. Diese Zeitstempel werden einerseits angegeben, um sicherzustellen, daß die Nachricht, die diesen Stempel trägt, nicht zu alt ist[2], andererseits sind sie auch ein wichtiger Bestandteil der digitalen Unterschriften (⟶ 4.3.2.5). Jeder *Stamp* besteht aus zwei INTEGER–Zahlen, die das Absende–Datum in Zahlenform enthalten.

4.3.2.3 Registration

Der Datentyp *Registration* wird für die Anmeldung der einzelnen Teilnehmer bei der Zentrale benutzt. Bevor ein Verbindungsaufbau zwischen den Teilnehmern stattfinden kann, müssen diese natürlich als erstes der Zentrale ihre öffentlichen Schlüssel mitteilen. Die *Registration* enthält die Teilnehmernummer und den öffentlichen Schlüssel des Absenders.

[2] es handelt sich sozusagen um ein „Haltbarkeitsdatum“...

4.3.2.4 KeyRequestData

Der Datentyp *KeyRequestData* wird für die Schlüsselanforderung von $\mathcal{A}$ an $\mathcal{SVZ}$ benötigt. Er enthält die Benutzernummern von $\mathcal{A}$ und $\mathcal{B}$, sowie einen Zeitstempel.

4.3.2.5 Certificate

Der Datentyp *Certificate* ist die zentrale Datenstruktur, die für den Verbindungsaufbau benutzt wird. $\mathcal{SVZ}$ schickt zwei dieser Zertifikate an $\mathcal{A}$ und $\mathcal{A}$ schickt das zweite dieser beiden an paB. Auch die Kontrollmeldungen, die paA und $\mathcal{B}$ noch austauschen, sind vom Typ *Certificate*.

Ein Zertifikat enthält die Teilnehmernummer eines der Teilnehmer[3], einen öffentlichen Schlüssel eines Teilnehmers[4], sowie einen Zeitstempel und eine digitale Unterschrift. Die digitale Unterschrift entsteht, indem der Zeitstempel mit einem privaten Schlüssel verschlüsselt und in das Datenfeld *signature* eingetragen wird. Man beachte, daß dieses Feld zwar formal als *Stamp* (⟶ 4.3.2.2) deklariert ist, aber mit gutem Grund an letzter Stelle im RECORD steht: wie bereits erwähnt, ändert sich durch die Verschlüsselung der Speicherbedarf dieser Unterschrift. Das Eintragen der Unterschrift erfolgt im Programm daher auch nicht in eine Variable von Typ *Certificate*, sondern das Zertifikat wird vorher in einen *Dirty*–Block (⟶ 4.3.2.1) kopiert.

4.3.2.6 MessageType

Dieser Datentyp dient als Typ–Kennung für die einzelnen Nachrichten, die beim Verbindungsaufbau ausgetauscht werden. Ein Byte mit einem der Werte dieses Aufzählungstyps wird jeder Nachricht vorangestellt, sofern Sie mit den Prozeduren des Moduls *KeyMaster* erzeugt und abgeschickt wird. Nur durch dieses Kennbyte kann der Empfänger der Nachricht feststellen, von welcher Art die nachfolgende Nachricht ist.

4.3.3 Prozeduren

Bei diesen Prozeduren wurden alle Parameter, die in ihrem Umfang eine Länge von wenigen Bytes überschreiten, als „VAR“-Parameter (CALL BY REFERENCE) deklariert, einerseits, um das Laufzeitverhalten zu verbessern, vor allem aber, um nicht mit der unter MS–DOS gültigen Beschränkung der Stack–Größe in Konflikt zu geraten. Die Alternative hätte darin bestanden, ähnlich wie in C explizit Zeiger zu übergeben; etwas anderes geschieht aber bei CALL BY REFERENCE auch nicht, nur die Schreibweise ist wesentlich eleganter, da die Zeiger implizit gehandhabt werden, ohne daß der Programmierer sich weiter darum kümmern muß.

Bei den Beschreibungen der Prozeduren wird $\mathcal{A}$ als Bezeichnung für den Kommunikationsteilnehmer benutzt, der die Schlüssel bei $\mathcal{SVZ}$ anfordert, und $\mathcal{B}$ als Bezeichnung für den anderen (passiven) Partner.

4.3.3.1 HandleRegistration

Diese Prozedur wird von $\mathcal{SVZ}$ benutzt, um die Anmeldung eines der Kommunikationspartner zu behandeln. Es müssen die Kennung des Eingabefiles und der öffentliche Schlüssel von $\mathcal{SVZ}$ übergeben werden.

[3] welche, hängt vom Zweck des Zertifikates ab
[4] ebenfalls abhängig vom Zweck des Zertifikates

4.3.3.2 DecodeKeyMessageAB

$\mathcal{A}$ benutzt diese Prozedur, um seinen Anteil aus der *KeyMessage* zu extrahieren, die $\mathcal{SVZ}$ zusammengestellt hat. Es müssen die *KeyMessage* selbst, der geheime Schlüssel von $\mathcal{A}$ und der öffentliche Schlüssel von $\mathcal{SVZ}$ übergeben werden. Als Ergebnisse erhält man das vollständig entschlüsselte Zertifikat A, das teilweise entschlüsselte Zertifikat B und die Länge von Zertifikat B (in Bytes) zurück.

4.3.3.3 DecodeKeyMessageB

$\mathcal{B}$ benutzt diese Prozedur, um seinen Anteil der *KeyMessage* von $\mathcal{SVZ}$, den ihm $\mathcal{A}$ geschickt hat, auszuwerten. Übergeben werden müssen neben der Nachricht selbst der geheime Schlüssel von $\mathcal{B}$ und der öffentliche Schlüssel von $\mathcal{SVZ}$. Als Ergebnis erhält man das vollständig entschlüsselte Zertifikat B zurück.

4.3.3.4 SendKeyMessage

Der Name dieser Prozedur stellt ein leichtes Understatement dar, denn hier wird der ankommende *KeyRequest* von $\mathcal{SVZ}$ analysiert, es werden beide Zertifikate (je eines für $\mathcal{A}$ und $\mathcal{B}$) erzeugt, zusammengestellt und schließlich abgeschickt. Dazu müssen die Eingabedatei, die den *KeyRequest* enthält, und der geheime Schlüssel von $\mathcal{SVZ}$ übergeben werden.

4.3.3.5 PutKeyMessageB

$\mathcal{A}$ benutzt diese Prozedur, um den zweiten Teil der *KeyMessage* weiter an $\mathcal{B}$ zu schicken. Übergeben werden müssen die Ausgabedatei, das Zertifikat B und die Länge des Zertifikats (in Bytes).

4.3.3.6 SendKeyRequest

$\mathcal{A}$ benutzt diese Prozedur, um die Schlüsselanforderung für den Verbindungsaufbau zwischen $\mathcal{A}$ und $\mathcal{B}$ zu erzeugen und an $\mathcal{SVZ}$ zu schicken. Es müssen die Teilnehmernummern von $\mathcal{A}$ und $\mathcal{B}$, sowie der öffentliche Schlüssel von $\mathcal{SVZ}$ übergeben werden.

4.3.3.7 SendRegistration

Diese Prozedur wird sowohl von $\mathcal{A}$ als auch von $\mathcal{B}$ benutzt, um sich zu Beginn der Kommunikation bei $\mathcal{SVZ}$ anzumelden. Es müssen die Teilnehmernummer des Kommunikationsteilnehmers und sein öffentlicher Schlüssel übergeben werden.

4.3.3.8 BuildDupMsg

Sowohl $\mathcal{A}$ als auch $\mathcal{B}$ benutzen diese Prozedur, um die Kontrollmeldungen zu erzeugen, die im Rahmen der gegenseitigen Authentisierung ausgetauscht werden.

Wenn $\mathcal{A}$ die *DupMessage* erzeugt, müssen die Teilnehmernummer von $\mathcal{A}$, öffentlicher und geheimer Schlüssel von $\mathcal{A}$ und öffentlicher Schlüssel von $\mathcal{B}$ übergeben werden.

Wenn $\mathcal{B}$ die *DupMessage* erzeugt, müssen die Teilnehmernummer von $\mathcal{A}$, öffentlicher Schlüssel von $\mathcal{A}$, geheimer Schlüssel von $\mathcal{B}$ und nochmals der öffentliche Schlüssel von $\mathcal{A}$ übergeben werden.

In beiden Fällen erhält man die fertige *DupMessage*, sowie ihre Länge (in Bytes) als Ergebnis zurück.

4.3.3.9 DecodeDupMsg

Sowohl $\mathcal{A}$ als auch $\mathcal{B}$ benutzen diese Prozedur, um die Kontrollmeldungen auszuwerten, die im Rahmen der gegenseitigen Authentisierung ausgetauscht werden.

Wenn $\mathcal{A}$ die *DupMessage* von $\mathcal{B}$ auswertet, müssen als Parameter der öffentliche Schlüssel von $\mathcal{B}$ und der geheime Schlüssel von $\mathcal{A}$ übergeben werden.

Wenn $\mathcal{B}$ die *DupMessage* von $\mathcal{A}$ auswertet, müssen als Parameter der öffentliche Schlüssel von $\mathcal{A}$ und der geheime Schlüssel von $\mathcal{B}$ übergeben werden.

Als Ergebnis erhält man das vollständig entschlüsselte Zertifikat zurück, das in der *DupMessage* enthalten war.

4.3.3.10 SendDupMsg

Diese Prozedur wird sowohl von $\mathcal{A}$ als auch von $\mathcal{B}$ benutzt, um die fertig verschlüsselte *DupMessage* abzuschicken. übergeben werden müssen die Ausgabedatei, die *DupMessage* und ihre Länge (in Bytes).

Kapitel 5

Zusammenfassung

In diesem Abschnitt möchten wir eine Zusammenfassung über das RSA-Verfahren geben und eventuelle Einsatzgebiete aufzeigen. Gleichzeitig sollen Mängel des Verfahrens beleuchtet werden, um die Grenzen dieser Methode zu veranschaulichen.

5.1 Abschätzung der Einsatzgebiete

Eines der Hauptnachteile des RSA-Verfahrens ist die niedrige Verschlüsselungsgeschwindigkeit, was auch nicht durch eine Hardwareimplementierung aufgefangen werden kann, weil die Restklassenpotenzierung eine aufwendige Operation ist. Solange kein effizienterer Algorithmus für die Berechnung von $N^e \ mod \ n$ zur Verfügung steht, wird dies der Hauptnachteil dieses Verfahrens bleiben. Dieser Nachteil wird auch nicht durch den Einsatz schnellerer Computer entschärft, weil dann längere Primzahlen p und q verwendet werden müssen, um eine Entschlüsselung unmöglich zu machen.
Da das RSA-Verfahren vergleichsweise langsam[1] ist, kann es auf keinen Fall in Netzen mit hoher Datenrate eingesetzt werden. D.h. eine Anwendung in LANs oder vergleichbaren Netzwerken entfällt. Auch in langsameren Netzen ist ein Einsatz dieses Verfahrens nur schwer vorstellbar, weil der Hard- und Softwareaufwand in keinem Verhältnis zum Nutzen dieses Verfahrens steht.
Einige Einsatzgebiete lassen sich aber dennoch erkennen. Oft ist nur eine Identifikation eines Rechners bei einem Anderen notwendig. Dies ist meist nicht zeitkritisch und wird nur einmal am Beginn der Kommunikation durchgeführt. Hier kann das RSA-Verfahren eingesetzt werden, weil die Identifikationsschlüssel in den meisten Fällen nicht sehr lang sind. Außerdem muß bei einem solchen Verfahren eine digitale Unterschrift erzeugt werden, was mit dem RSA-Verfahren relativ leicht möglich ist.
Auch der Einsatz in *"hybriden Verschlüsselungsverfahren"* ist denkbar. Unter diesem Begriff verstehe ich folgende Methode:

1. Berechne die Parameter eines schnellen symmetrischen Verschlüsselungsverfahrens.
2. Verschlüsseln dieser Parameter mit dem langsamen asymetrischen Verfahren [2].
3. Verschicken der verschlüsselten Werte an den Netzpartner.

[1] In unserer Implementation wurde etwa eine Geschwindigkeit von 70 Bit/s bei der Ver- bzw. Entschlüsselung erzielt
[2] z.B dem RSA-Verfahren

4. Der Empfänger entschlüsselt diese Parameter und initialisiert sein symmetrisches Verfahren mit diesen Werten.

5. Der Datenaustausch geschieht nun mit Hilfe des schnellen symmetrischen Verfahren.

Diese Vorgehensweise hat den Vorteil das auch digitale Unterschriften möglich sind, obwohl das schnelle symetrische Verfahren benutzt wird, das eigentlich keine digitalen Unterschriften zuläßt. Als Beispiel für diese Idee könnte das RSA-Verfahren und das Vernam-Verfahren[3] angeführt werden. Man könnte die Parameter eines einfachen Pseudozufallszahlengenerators mit Hilfe des RSA-Verfahrens verschlüsseln und an den Netzteilnehmer verschicken. Bei einer Probeimplementierung des Vernamverfahrens wurden eine Verschlüsselungsgeschwindigkeit von 200 kBit[4] erreicht. Da das Vernam-Verfahren sehr leicht in Hardware abzubilden ist, wären in der Praxis viel höhere Verschlüsselungsgeschwindigkeiten möglich.

5.2 Mängel des RSA-Verfahrens

Obwohl das RSA-Verfahren als eines der sichersten Verfahren bekannt ist, hat es einige Mängel, die auf den ersten Blick nicht auffallen. Wie schon weiter oben erwähnt, ist die Wahl der Primzahlen p und q nicht unkritisch. Sie dürfen nicht zu nahe zusammenliegen, weil sonst ein schneller Faktorisierungsalgorithmus für die Zahl $n = p*q$ existiert. Weiterhin sollten die Zahlen $p-1$ und $q-1$ möglichst große Primfaktoren enthalten und der $ggT(p-1, q-1)$ sollte möglichst klein[5] sein. Dies ist notwendig um eine Iterationsattacke zu erschweren. Unter einer Iterationsattacke versteht man folgendes Vorgehen:

1. Verschlüssele die verschlüsselte Nachricht so lange mit der Encrypt-Funktion $E(N)$ bis man die ursprüngliche verschlüsselte Nachricht wieder erhält.

2. Die im vorletzten Schritt erhaltene Nachricht ist die entschlüsselte Information.

Obwohl dieses Vorgehen anscheinend das RSA-Verfahren unbrauchbar macht, kann man zeigen, daß eine Iterationsattacke den gleichen Aufwand hat, wie eine Faktorisierung der Zahl n. Deshalb ist es wichtig die oben gemachten Einschränkungen der Wahl von p und q zu beachten.
Die scheinbare Sicherheit des RSA-Verfahrens begründet sich darauf, daß die Primzahlen p und q zufällig gewählt werden. Dies ist aber in der Realität nicht der Fall, weil spezielle Algorithmen verwendet werden müssen um große Primzahlen zu finden. Deshalb könnte ein Faktorisierungsalgorithmus gefunden werden, der die Eigenschaften der Primzahlen ausnützt, die z.B. durch den Miller-Rabin-Test gefunden werden.
Sollte ein effizienter Faktorisierungsalgorithmus mit polynomialen Zeitverhalten gefunden werden, so wird das RSA-Verfahren unbrauchbar. Zum jetzigen Zeitpunkt ist aber nicht bekannt, ob es möglich ist einen solchen Algorithmus zu finden.
Trotz all dieser Mängel ist das RSA-Verfahren eines der sichersten Public Key Cryptosysteme und kann bei Beachtung einiger Regeln bedenkenlos eingesetzt werden.

[3] Diese Verfahren wurde 1926 von Vernam vorgeschlagen
[4] MC68030 (25MHz)
[5] Der teilerfremde Zustand ist nicht zu erreichen, da p-1 und q-1 gerade sind

Kapitel 6

Literaturverzeichnis

Dr. v. Koch, *Vorlesungsskript zu "Kryptographie",* WS 1990/91

Eckart Winkler, *Wenn lang nicht langt,* c't April 1989 Verlag Heinz Heise GmbH & CoKG

o. Autor, *"DATACOM",* Kapitel 3.4.4

M.R. Schroeder, *Number Theory in Science and Communication,* Springer Verlag, 1990

W.Scharlau H.Opolka, *Von Fermat bis Minkowski,* Springer 1980

Donald Ervin Knuth, *The Art of Computer Programming,* 2d ed. Volume 2, Addison Wesley, 1981

Fritz Reinhardt / Heinrich Soeder, *dtv-Atlas zur Mathematik,* Deutscher Taschenbuch Verlag, Band 1 & 2, 1990

Evangelos Kranakis, *Primality and Cryptography,* Teubner 1986

Beth / Heß / Wirl, *Kryptographie,* Teubner 1983

Kapitel 7

Verwendete Symbole

$\mathcal{N}$ Menge der natürliche Zahlen
$\mathcal{Z}$ Menge der ganzen Zahlen
$\mathcal{Z}_n$ Restklassenring der Ordnung n
$\mathcal{P}$ Menge der Primzahlen $\{2, 3, 5, 7, 11, 13, \ldots\}$
$\lceil\ \rceil$ Aufrunden auf die nächste ganze Zahl
$p|q$ p teilt q ohne Rest
$\prod_{i=1}^{n} a_i = (a_0 * a_1 * \ldots * a_{n-1} * a_n)$
$\sum_{i=1}^{n} a_i = (a_0 + a_1 + \ldots + a_{n-1} + a_n)$
$\zeta(s)$ Riemannsche Zetafunktion
$\mathcal{R}^*$ Einheitengruppe (Menge aller invertierbaren Elemente in $\mathcal{R}$)