Praktikumsbericht

Betriebssysteme

von

Ronald Kriemann und Christian Rost

10. August 1998

Inhaltsverzeichnis

1. Einführung

In diesem Praktikum über Betriebssysteme ging es um die Implementierung eines Bankservers. Besonderer Schwerpunkt lag dabei auf der Sicherung der Kommunikation mit kryptographischen Mitteln. Es wurden verschiedene Verschlüsselungsverfahren und Kommunikationsprotokolle untersucht und auf einem UNiX-System getestet.

Die Sicherung der Kommunikation ist notwendig, da bei Online-Bankgeschäften gewisse Sicherheitsbedürfnisse vorliegen. Zu diesen gehören etwa die korrekte Authentifizierung des Kunden und der Bank, die Geheimhaltung des Kommunikationsinhaltes (Vertraulichkeit), die Garantie, daß Daten nicht geändert werden (Integrität) und der Beweis des Empfanges von Daten (Verbindlichkeit).

In den folgenden Abschnitten werden diese Verfahren vorgestellt. In Kapitel 2 sind dies diverse Hilfsalgorithmen, deren Verwendung sich durch das gesamte Praktikum zieht. Die wichtigsten kryptographischen Verfahren werden in Kapitel 3 beschrieben, gefolgt von den Kommunikationsprotokollen in Kapitel 4. In Kapitel 5 wird näher auf die eigentliche Implementierung eingegangen.

2. Hilfsalgorithmen

In diesem Abschnitt werden einige Algorithmen vorgestellt, die an den verschiedensten Stellen und in unterschiedlichen Zusammenhängen innerhalb des Praktikums verwendet wurden.

2.1. Base64

Ein Problem bei der Übertragung von Bitfolgen innerhalb von Programmen oder über Netzwerke ist die Interpretation einiger weniger Zeichen durch das Betriebssystem oder die verwendete Programmiersprache. So wird etwa das ASCII-Zeichen \000 als String-Ende in der Sprache C genutzt.

Ein Verfahren um diese Probleme zu umgehen, stellt *Base64* dar. Der Name leitet sich von der Anzahl der verwendeten Zeichen ab. Diese beträgt 64 und enthält nehmen dem Alphabet und den Zahlen 0-9 die Zeichen "+", "/" und "=". Das Gleichheitszeichen dient der Kennzeichnung von Bits, die dem zu kodierenden Bitstream hinzugefügt wurden, um ein Bitanzahl zu erreichen, die einem Vielfachen von 6 entspricht. Dies ist notwendig, da genau 6 Bits der Bitfolge in ein Zeichen aus Base64 kodiert werden ($2^6 = 64$).

2.2. MD5

MD5 ist ein Verfahren zur (Einweg-) Hashwertbildung einer gegebenen Bitfolge, d.h. ein gegebenes $m \in \mathbb{B}^*$ wird auf ein $m' \in \mathbb{B}^k$ abgebildet. Im Falle von MD5 ist $k = 128$.

Als Hashfunktion werden dabei einige Anforderungen an MD5 gestellt. Die Wichtigste ist die Vermeidung von Kollisionen, d.h. es sollte für zwei $m_1, m_2 \in \mathbb{B}^*$ "möglichst" gelten: $\text{md5}(m_1) \neq \text{md5}(m_2)$. Aufgrund der unterschiedlichen Mächtigkeit von $\mathbb{B}^*$ und $\mathbb{B}^{128}$ ist dies aber im allgemeinen nicht möglich. Als hinreichend wird aber angesehen, falls es innerhalb des Wertebereiches $\mathbb{B}^{128}$ keine Häufungspunkte gibt, die Werte aus $\mathbb{B}^*$ also gleichmäßig verteilt werden.

Aus kryptographischen Gründen werden weitere Forderungen an MD5 erhoben. Zum einen sollen aus der Kenntnis des Hashwertes keinerlei Rückschlüsse auf den ursprünglichen Inhalt der Bitfolge möglich sein. Zum anderen soll bei "ähnlichen" Bitfolgen, also Folgen, die nur in wenigen Bits verschieden sind, ein "stark" abweichender Hashwert gebildet werden. Verbunden hiermit ist die Forderung, daß jedes Bit der Folge in die Bildung des Hashwertes mit einbezogen wird.

Das Verfahren selbst wird beschrieben im *Request for Comment* Nr. 1321 (RFC1321). Dort wird auch eine Beispielimplementierung vorgestellt.

2.3. Zufallszahlen

In vielen Bereichen der Kryptographie (und anderen Gebieten) werden Zufallszahlen benötigt. Die dabei auftretenden Ansprüche unterscheiden sich allerdings teilweise sehr. Nachfolgend werden einige Klassen von Zufallszahlen vorgestellt.

Echte Zufallszahlen sind nicht vorhersagbar. Man erhält sie etwa aus gewissen Hardwaredaten, speziellen Geräten (z.B. Geigerzähler) oder durch Messung von personenabhängigen Daten (etwa Tastaturanschlägen). Einige UNiX-Systeme (z.B. Linux) stellen ein Device bereit, über welches solche Zufallszahlen ausgelesen werden können.

Pseudozufallszahlen genügen statistischen Tests (etwa dem Spektraltest). Üblicherweise werden sie mit Hilfe eines Zufallszahlengenerators berechnet, also über einen mathematischen Algorithmus. Hierbei ist eine lange Periode erwünscht.

Kryptographisch sichere Pseudozufallszahlen sollen dagegen nicht voraussagbar sein, zumindest nicht ohne die Kenntnis eines Schlüssels. Bei Anwendung als Zufallsbitstrom soll es also unmöglich sein, aus dem Vorhandensein einer beliebig langen Teilfolge auf das nächste Bit zu schließen.

3. Verschlüsselungsverfahren

In diesem Abschnitt werden die verwendeten kryptographischen Verfahren zum Ver- bzw. Entschlüsseln einer Nachricht, i.allg. als Bitstrom vorliegend, beschrieben.

3.1. IDEA

Das *IDEA*-Verfahren (der Name steht für *International Data Encryption Algorithm*) ist ein Blockverschlüsselungsverfahren. Die verwendete Blockgröße beträgt 64 Bit. Der (geheime) Schlüssel zum Kodieren hat eine Länge von 128 Bit.

IDEA basiert auf drei verschiedenen algebraischen Gruppenoperationen in $\mathbb{B}^{16}$

- XOR ($\oplus$),
- Addition modulo 2^{16} ($\boxplus$) und
- Multiplikation modulo $2^{16}+1$ ($\odot$).

($\mathbb{B}^{16}, \oplus$) ist eine abelsche Gruppe, in der jedes Element zu sich selbst invers ist. Die Gruppe bzgl. $\boxplus$ ist zyklisch und wird durch 1 erzeugt. Zu $\odot$ ist zu sagen, daß, da $2^{16}+1$ eine Primzahl ist, $\mathbb{Z}_{2^{16}+1}$ einen Körper und somit $(\mathbb{Z}_{2^{16}+1}, \odot)$ eine multiplikative Gruppe bildet.

Die Verschlüsselung arbeitet in 9 Runden. Dazu werden 52 Schlüssel mit je 16 Bit Länge benötigt, $Z_1^i, \ldots, Z_6^i$, $i = 1, \ldots, 8$, und $Z_1^9, \ldots, Z_4^9$. Zur Bestimmung dieser Schlüssel sei $K \in \mathbb{B}^{128}$ der IDEA-Schlüssel. Die ersten 8 Schlüssel $Z_1^1, \ldots, Z_6^1, Z_1^2, Z_2^2$ erhält man durch Kopieren der entsprechenden Bitpositionen von K, also Bit 0-15 ergeben Z_1^1 usw. Durch zyklisches Linksverschieben der Bits in K und Zuweisung nach obigem Schema erhält man nacheinander alle anderen Schlüssel.

Den Graphen des gesamten Verschlüsselungsprozesses kann man Abbildung 3.1 entnehmen. Dabei sind $X_1, \ldots, X_4$ die 16-bittigen Eingabewerte und $Y_1, \ldots, Y_4$ die 16-bittigen Ausgabewerte.

Die Entschlüsselung erfolgt mit dem gleichen Algorithmus, allerdings anderen Schlüsseln. Dabei wird jeweils das Inverse bzgl. $\boxplus$ und $\odot$ gebildet.

Eine genaue Beschreibung des IDEA-Algorithmus kann [LM91] entnommen werden.

3.1.1. Sicherheit von IDEA

Notwendig für die Sicherheit von IDEA sind die folgenden beiden Eigenschaften des Algorithmus. Der dritte Punkt zeigt eine Schwachstelle von IDEA auf.

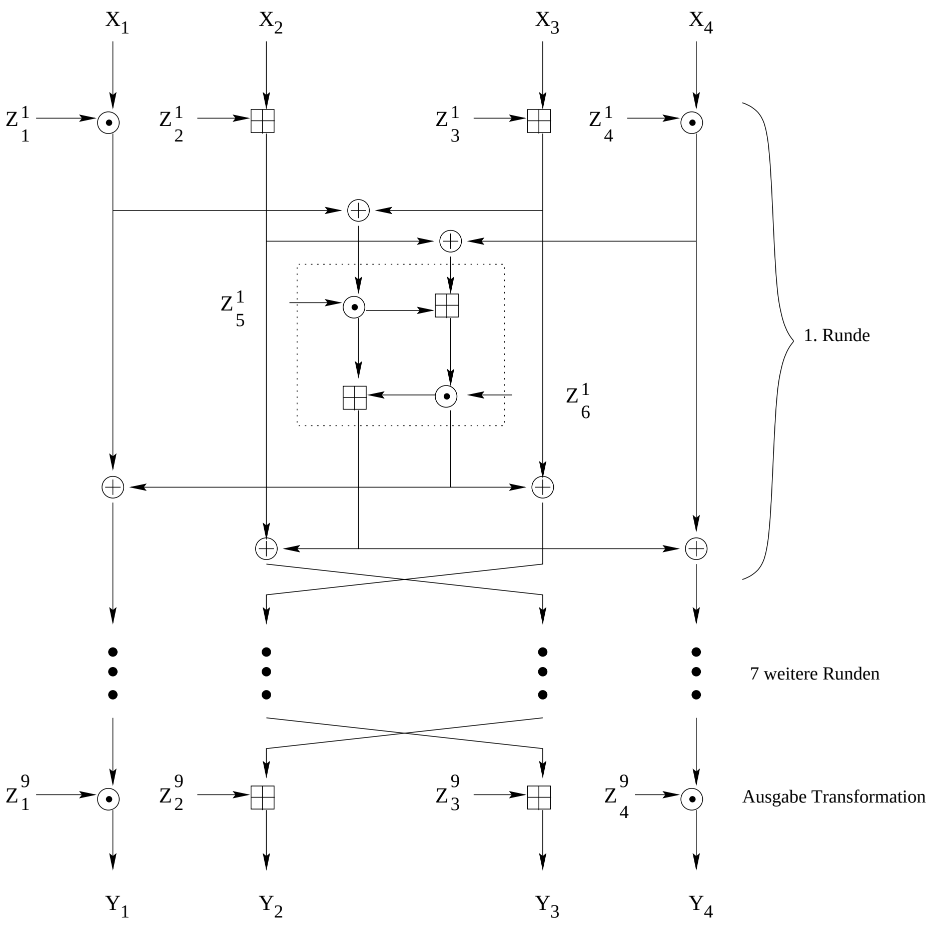

Abbildung 3.1: Berechnungsgraph des IDEA-Verschlüsselungsprozesses

Konfusion

Unter *Konfusion* versteht man die Verschleierung des Zusammenhangs zwischen Klar- und Chiffretext. Diese wird durch folgende Eigenschaften des Algorithmus erreicht:

1. $(\mathbb{B}^{16}, \oplus)$ ist zu den andern Gruppen nicht isomorph.
2. $(\mathbb{B}^{16}, \boxplus)$ und $(\mathbb{Z}_{2^{16}+1}, \odot)$ sind zyklisch aber Isomorphismus entspricht nicht der trivial Abbildung.
3. für jeweils zwei der drei Operatoren gilt das Distributivgesetz nicht. Außerdem gilt auch nicht das verallgemeinerte Assoziativgesetz: $a \boxplus (b \oplus c) \neq (a \boxplus b) \oplus c$.

Diffusion

Diffusion ist die Verteilung statistischer Eigenschaften des Klartextes über den Chiffretext. Hieraus folgend ergibt sich, daß jedes Ausgabebit von jedem Eingabebit abhängt. Dies wird durch die Multiplikations-Additions-Struktur im IDEA-Algorithmus (siehe Abbildung 3.1, gepunktete Sektion) erreicht.

Schwache Schlüssel

Es kann gezeigt werden, daß es eine Klasse von Schlüsseln (sog. *schwache Schlüssel*) gibt, deren Benutzung und Wert effizient ermittelt werden kann. In [DGV] ist diese Klasse angegeben und es wird eine Modifikation von IDEA vorgestellt, die diese Probleme eliminiert.

3.2. RSA

Das **RSA**-Verfahren (der Name leitet sich aus den Namen der Entwickler **R**ivest, **S**hamir und **A**dleman ab) ist ein unsymmetrischen Verschlüsselungsverfahren. Dies bedeutet, daß Sender und Empfänger verschiedene Schlüssel zum ver- bzw. entschlüsseln benutzen. Die dabei verwendeten Schlüssel sind der *private* und der *öffentliche* Schlüssel. Ein Anwender generiert beide Schlüssel und veröffentlicht den freien Teil (den öffentlichen Schlüssel).

3.2.1. Mathematische Grundlagen

Definition 3.1 (Eulersche Funktion) die *Eulersche Funktion* $\phi : \mathbb{N} \to \mathbb{N}$ ist definiert durch

$$\phi(n) = |\{m \in \mathbb{N} : m < n, \mathrm{ggt}(m,n) = 1\}|, \quad \forall n \in \mathbb{N}.$$

Dabei ist $\mathrm{ggt} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, die Funktion, die zwei natürlichen Zahlen ihren größten gemeinsamen Teiler zuordnet.

Sei $\mathcal{P}$ die Menge aller Primzahlen. Dann gilt für $p, q \in \mathcal{P}$:

$$\begin{aligned} \phi(p) &= p - 1 \\ \phi(p \cdot q) &= \phi(p) \cdot \phi(q) = (p-1)(q-1). \end{aligned}$$

Satz 3.2 (Satz von Euler) Seien $a, n \in \mathbb{N}$ mit $\mathrm{ggt}(q, n) = 1$. Dann gilt:

$$a^{\phi(n)} \equiv 1 \bmod n.$$

Bemerkung 3.3 (i) Seien $p, q \in \mathcal{P}$, $n := pq$ und $m \in \mathbb{N}$ mit $m < n$. Dann gilt mit dem Satz von Euler :

$$m^{\phi(n)} = m^{(p-1)(q-1)} \equiv 1 \bmod n.$$

(ii) Seien $m, n \in \mathbb{N}$ mit $m < n$ und $\mathrm{ggt}(m, n) = 1$. Dann existiert ein $m' \in \mathbb{N}_{<n}$ mit $m \cdot m' \equiv 1 \bmod n$.

3.2.2. Das RSA-Verfahren

Seien $p, q \in \mathcal{P}$ und $n = pq$. Wähle $e, d \in \mathbb{N}_{<n}$ mit $ed \equiv 1 \bmod \phi(n)$. Das Paar (e, n) ist der öffentliche, das Paar (d, n) der private Schlüssel.

Das Verschlüsseln mit dem öffentlichen Schlüssel erfolgt durch Potenzieren von M mit e:

$$C = M^e \bmod n.$$

C entspricht der kodierten Nachricht.

Das Entschlüsseln wird durch Potenzieren mit d durchgeführt. Es gilt:

$$\begin{aligned} C^d \bmod n &= (M^e \bmod n)^d \bmod n \\ &= M^{ed} \bmod n \\ &= M^{k \cdot \phi(n)+1} \bmod n \\ &= M \cdot \left(M^{\phi(n)}\right)^k \bmod n \\ &= M \cdot (1)^k \bmod n \\ &= M, \quad \text{für ein } k \in \mathbb{N}. \end{aligned}$$

Das Verschlüsseln mit dem privaten Schlüssel und das nachträgliche Entschlüsseln erfolgt analog, wobei e und d die Rollen tauschen.

Das RSA-Verfahren läßt sich auch zum Signieren von Nachrichten einsetzen. Dabei wird z.B. ein Hashwert der Nachricht mit dem privaten Schlüssel kodiert und an die Nachricht angehängt. Die Überprüfung der Korrektheit der Nachricht erfolgt mittels nochmaliger Bildung eines Hashwertes und Vergleich mit der entschlüsselten Signatur der Nachricht. Im einzelnen sind die Schritte zum Signieren und Überprüfen einer Nachricht M:

Signieren von M	1. $H = \mathrm{hash}(M)$ 2. $S = E_{priv}(H)$ 3. $M' = (M, S)$
Überprüfen von $M' = (M, S)$	1. $H = \mathrm{hash}(M)$ 2. $S = E_{pub}(S)$ 3. falls $H = S$ ist M korrekt.

3.2.3. Sicherheit von RSA

Ein "Schwachpunkt" bei RSA ist die Möglichkeit, aus den Zahlen e und n (dem öffentlichen Schlüssel) die Zahl d (und somit den privaten Schlüssel) zu bestimmen. Notwendig hierzu ist die Kenntnis von $\phi(n) = (p-1)(q-1)$. Zur Bestimmung von $\phi(n)$ genügt z.B. eine Faktorisierung von n in p, q.

Allerdings ist der Aufwand für die Faktorisierung nicht polynomiell. Somit reicht ein hinreichend großes n, um die Sicherheit des RSA-Verfahrens zu gewährleisten. Typisch sind hier $n \in \mathbb{B}^m$ und $m \in \{1024, 2048, 3072, \ldots\}$. Allerdings nimmt für wachsendes m auch der Aufwand für das Ver- bzw. Entschlüsseln zu, sodaß hier ein Kompromiß gefunden werden muß zwischen Sicherheit und Schnelligkeit des Verfahrens.

4. Protokolle

Im folgenden werden die Teilnehmer einer Kommunikation mit A, B, C usw. bezeichnet. Die Schlüssel des jeweiligen Partners werden mit dem Namen indiziert (z.B K_A für einen Schlüssel von Teilnehmer A). Die jeweiligen RSA-Schlüssel, soweit vorhanden, werden mit K^{pub} bzw. K^{priv} bezeichnet. ID_A bezeichnet einen Identifikator für Teilnehmer A. Kryptographische Funktionen werden mit $E(\cdot)$ dargestellt und mit dem verwendeten Schlüssel indiziert, falls dieser nicht aus dem Kontext hervorgeht. Das Signieren einer Nachricht wird analog mit $S(\cdot)$ bezeichnet.

4.1. Angriffe auf Protokolle

In diesem Abschnitt sollen einige Angriffsmöglichkeiten auf Protokolle und entsprechende Lösungsansätze vorgestellt werden.

A und B haben einen gemeinsamen Schlüssel K. Ein einfaches Protokoll wäre z.B.:

$$\boxed{\begin{array}{ll} A \to B: & \mathrm{ID}_A \\ B \to A: & \mathrm{OK} \\ A \to B: & E_K(\ldots) \end{array}}$$

Hierbei wäre zunächst ein einfacher Angriff ohne Kenntnis von K möglich. Dann erfolgt ein Abhören der Verbindung und späteres Wiederverwenden der übertragenen Packete. Eine solche Attacke wird als *Replayattacke* bezeichnet.

Eine Lösung für dieses Problem besteht in der Einführung von Werten innerhalb der Kommunikation, die sich mit der Zeit ändern, z.B. der aktuellen Zeit oder Zufallswerten (sog. *Nonces*).

$$\boxed{\begin{array}{lll} A \to B: & \mathrm{ID}_A & \\ B \to A: & \mathrm{OK}\ N & \text{(mit Nonce } N\text{)} \\ A \to B: & E_K(N, \ldots) & \end{array}}$$

Dieses Protokoll liefert Sicherheit, bzgl. Replayattacken, für B. Um diese Sicherheit auch für A zu gewährleisten, müssen auch für B Nonces eingeführt werden.

Ein weiteres Problem ergibt sich bei Benutzung eines festen Schlüssels K. Damit werden z.B. bestimmte kryptoanalytische Verfahren erleichtert, falls große Datenmengen verschlüsselt werden. Auch *Brute-Force-Attacken* auf K haben eine größere Chance, falls feste Flosskeln über längere Zeiträume hinweg verwendet werden.

Abhilfe schafft hier die Verwendung von Schlüsseln, die für jede Kommunikations einzeln aufgebaut werden, sogenannten *Sitzungsschlüsseln* (siehe auch Abschnitt 4.2).

$A \rightarrow B$:	ID_A	
$B \rightarrow A$:	$E_K(K')$	(K' ist neuer Sitzungsschlüssel)
$A \rightarrow B$:	$E_{K'}(\ldots)$	

Durch Einführung von Nonces können auch hier, oben beschriebene Attacken abgewehrt werden.

4.2. Schlüsselaustausch

Ziel der hier vorgestellten Protokolle ist der Austausch von Schlüsseln, um einen sicheren Kanal zur Übertragung von Daten zu gewährleisten, bzw. um generell das Mitlesen von Nachrichten zu unterbinden.

Man unterscheidet zwei Arten von schlüsselbasierter Kommunikation. Zum einen ist dies das *symmetrische* Verfahren, bei dem beide Partner einen identischen Schlüssel besitzen. Beim *asymmetrischen* Verfahren exitieren je ein privater und ein öffentlicher Schlüssel pro Teilnehmer.

Bei beiden Arten ist der Austausch der Schlüssel, abgesehen von den eingesetzen kryptographischen Algorithmen, das Hauptproblem, um eine abhörsichere Kommunikation zustande zu bringen. In vielen Fällen ist der direkte physische Austausch (etwa schriftlich oder per Diskette) unhandlich oder gar unmöglich. Bei den asymmetrischen Verfahren ergibt sich eine gewisse Sicherheit durch die Bekanntgabe des öffentlichen Schlüssels, etwa in speziell hierfür vorgesehenen Verzeichnissen (z.B. öffentliche Keyserver).

Ansätze zur Überwindung dieser Probleme sind etwa:

- die Verwendung von Hashwerten (sog. Fingerprints) des Schlüssels, welche dann über einen sicheren Kanal verschickt werden.
- Schlüsselaustauschprotokolle (siehe auch Abschnitt 4.2.1)
- die Nutzung von Zertifikaten (z.B. X.509)
- ein dezentraler Ansatz auf Vertrauensbasis, wie er z.B. in PGP vorkommt.

4.2.1. Schlüsselaustausch mit RSA

Obwohl durch das RSA-Verfahren bereits ein sicherer Kanal möglich ist (siehe Abschnitt 3.2.3), ist es sinnvoll einen Schlüsselaustausch durchzuführen, um beiden Kommunikationspartnern einen symmetrischen Schlüssel zur Verfügung zu stellen. Hierfür spricht z.B. die höhere Geschwindigkeit von symmetrischen Verschlüsselungsverfahren (z.B. IDEA, siehe Abschnitt 3.1).

Ein einfaches Verfahren ohne Vorbedingungen

Zur Aufnahme einer Kommunikation von A nach B generiert A einen RSA-Schlüssel und sendet den öffentlichen Anteil an B. B generiert daraufhin einen Sitzungsschlüssel K und sendet diesen (RSA-verschlüsselt) an A.

$$\boxed{\begin{array}{ll} A \rightarrow B: & \mathrm{ID}_A, K_A^{pub} \\ B \rightarrow A: & E_{K_A^{pub}}(K) \end{array}}$$

Dieses Protokoll ist sicher gegen reines Abhören, bedingt durch die Verwendung des RSA-Verfahrens. Allerdings besteht die Möglichkeit, daß sich ein dritter Teilnehmer (C) zwischen A und B schaltet, die Nachricht von A abfängt und den eigenen RSA-Schlüssel an B sendet, woraufhin B eine Kommunikation mit C beginnt. Diese Form der Attacke wird *Man-in-the-middle-attack* genannt.

$$\boxed{\begin{array}{lll} A \rightarrow C: & \mathrm{ID}_A, K_A^{pub} & \text{(C fängt Nachricht ab)} \\ C \rightarrow B: & \mathrm{ID}_A, K_C^{pub} & \text{(C beginnt Kommunikation mit B)} \\ B \rightarrow C: & E_{K_C^{pub}}(K) & \\ C \rightarrow A: & E_{K_A^{pub}}(K) & \end{array}}$$

Eine Lösung für dieses Problem bildet das sogenannte *Interlock-Protocol*. Dabei werden zunächst die öffentlichen Schlüssel der jeweiligen Kommunikationspartner ausgetauscht und dann abwechselnd die gerad- bzw. ungeradzahligen Anteile der Sitzungsschlüssel K_1 und K_2, jeweils mit dem öffentlichen Schlüssel des Partners kodiert.

$$
\begin{array}{lll}
A \to B: & \mathrm{ID}_A, K_A^{pub} & \\
B \to A: & \mathrm{ID}_B, K_B^{pub} & \\
A \to B: & \mathrm{OB}(E_{K_B^{pub}}(K_1)) & (\mathrm{OB} = \text{Odd Bits}) \\
B \to A: & \mathrm{OB}(E_{K_A^{pub}}(K_2)) & \\
A \to B: & \mathrm{EB}(E_{K_B^{pub}}(K_1)) & (\mathrm{EB} = \text{Even Bits}) \\
B \to A: & \mathrm{EB}(E_{K_A^{pub}}(K_2)) &
\end{array}
$$

Für eine *Man-in-the-middle-attack* werden beide Bitfolgen der Nachricht benötigt, um den Schlüssel zu dekodieren. Da diese aber abwechselnd übertragen werden, ist ein vorheriger Austausch wie in der ursprünglichen Version nicht möglich.

Verfahren mit Vorbedingungen

Bei den folgenden Verfahren wird davon ausgegangen, daß die Kommunikationspartner den öffentlichen Schlüssel des jeweils anderen Teilnehmers kennen. Dann kann jede übertragene Nachricht mittels RSA verschlüsselt werden.

$$
\begin{array}{ll}
A \to B: & E_{K_B^{pub}}(\mathrm{ID}_A) \\
B \to A: & E_{K_A^{pub}}(S_{K_B^{priv}}(K))
\end{array}
$$

Allerdings ist dieses Protokoll anfällig gegen Replayattacken (siehe Abschnitt 4.1). Als Lösung bietet sich wieder die Einführung von Nonces an. Das dann entstehende Protokoll wurde von Needham und Schroeder ('78) vorgestellt.

$$
\begin{array}{ll}
A \to B: & E_{K_B^{pub}}(\mathrm{ID}_A, N_1) \\
B \to A: & E_{K_A^{pub}}(N_1, N_2) \\
A \to B: & E_{K_B^{pub}}(S_{K_A^{priv}}(N_2, K))
\end{array}
$$

Dabei sind N_1 und N_2 jeweils neu generierte Zufallswerte und K der Sitzungsschlüssel.

Schlüsselaustausch über Keyserver

Das nachfolgende Verfahren geht zurück auf Dunning und Sacco ('81). Hierbei verwaltet ein Keyserver die öffentlichen Schlüssel aller Teilnehmer. Eine Kommunikation kommt dann zustande, indem vom Keyserver die Schlüssel des Kommunikationspartners angefordert werden.

$$\begin{array}{ll} A \to KS: & \mathrm{ID}_A, \mathrm{ID}_B \\ KS \to A: & S_{K_{KS}^{priv}}(\mathrm{ID}_B, K_B^{pub}), S_{K_{KS}^{priv}}(\mathrm{ID}_A, K_A^{pub}) \\ A \to B: & E_{K_B^{pub}}(S_{K_A^{priv}}(K, T), S_{K_{KS}^{priv}}(\mathrm{ID}_B, K_B^{pub}), S_{K_{KS}^{priv}}(\mathrm{ID}_A, K_A^{pub})) \end{array}$$

Dabei ist K der Sitzungsschlüssel.

Allerdings hat B nun die Möglichkeit, in einer nachfolgenden Kommunikation mit C, sich als A auszugeben, da er er über einen signierten Schlüssel von A verfügt.

$$B \to C: \quad E_{K_C^{pub}}(S_{K_A^{priv}}(K, T), S_{K_{KS}^{priv}}(\mathrm{ID}_C, K_C pub), S_{K_{KS}^{priv}}(\mathrm{ID}_A, K_A^{pub}))$$

Abhilfe schafft die explizite Verwendung der Kommunikationsteilnehmer innerhalb der Nachricht: $S_{K_A^{priv}}(K, T, \mathrm{ID}_A, \mathrm{ID}_B)$.

5. Implementierung

Ziel des Praktikums war der Aufbau eines Servers für Bankgeschäfte verschiedenster Art. In diesem Abschnitt sollen diese Geschäftsarten vorgestellt und deren Implementierung auf dem verwendeten Betriebssystem bzw. in der genutzten Programmiersprache beschrieben werden.

Das Betriebssystem, auf dem das Programm entwickelt und getestet wurde ist UNiX (speziell Solaris von Sun und Linux).

Als Programmiersprache wurde C++ verwendet, da die objektorientierte Programmierung bei einigen Algorithmen einen wesentlichen Vorteil bzgl. Kapselung von Daten und Algorithmen erlaubt (siehe z.B. Abschnitt 5.4).

Auf sprachliche Besonderheiten von C++ wird nicht näher eingegangen, genauso wird auf eine Beschreibung der verwendeten UNiX-Funktionalität verzichtet.

5.1. Client-Server-Modell

Ein Server für Bankgeschäfte sollte zumindest zwei Anforderungen erfüllen. Zum einen muß er Daten über alle Kunden und Konten konsistent verwalten können. Zum anderen sollte eine Kommunikation zwischen der Bank und entsprechenden Kunden möglich sein.

Die Realisation beider Anforderungen innerhalb eines Prozesses ist möglich, aber aufwendig. Wesentlich einfacher gestaltet sich die Trennung in einen Prozeß für die Verwaltung der Daten und einen Prozeß für die Kommunikation mit den Kunden. Durch diese Trennung ist außerdem für die Konsistenz der Daten gesorgt, da nur ein Prozeß direkte Kontrolle über die Datenbestände hat.

In Abbildung 5.1 ist das Zusammenspiel der einzelnen Prozesse, neben dem verwendeten Kommunikationsmodell dargestellt.

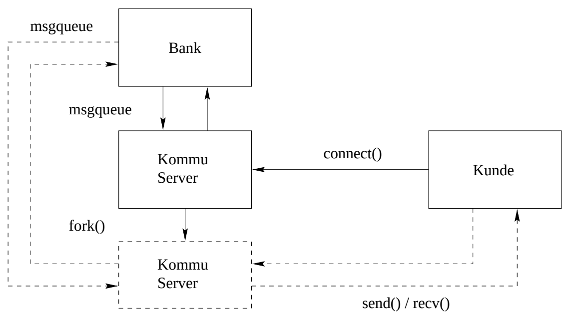

Abbildung 5.1: Schemata für das verwendete Client-Server-Modell

Die Kommunikation zwischen Bank und Kommunikationserver (Kommu-Server) wird über eine Message Queue (SYSV-IPC) abgewickelt, welche eine direkte Kommunikation, gesteuert über die Prozess-ID erlaubt.

Der Kommu-Server stellt einen UNiX-üblichen Socket-basierten Server dar, der auf Anfragen auf einem bestimmten Port wartet. Ein Kunde kontaktiert den Server auf diesem Port, woraufhin der Kommu-Server eine Kopie von sich selbst erzeugt (*fork*()) und die Kommunkation mit dem Kunden beginnt. Der ursprüngliche Prozeß ist somit in der Lage auf weitere Kundenanfragen zu reagieren.

5.2. Der Bankserver

Wie bereits in Abschnitt 5.1 angedeutet, dient der Bankserver der reinen Verwaltung der Daten. Diese Daten enthalten Informationen über Kunden, Konten und die jeweiligen Zugriffsberechtigungen. Wie in Abschnitt 5.5 beschrieben kommt die Verwaltung von digitalem Geld hinzu.

Sämtliche Kundendaten werden in einer Liste (siehe auch Abschnitt A.4) gespeichert. Dazu wird für jeden Kunden ein Objekt angelegt. Die Klasse, der diese Objekte angehören, ist folgendermaßen definiert:

```
class TKunde
{
    TString  name;
    int      nummer;
    TString  passwort;
    TString  sessionPassword;
    TString  onePassword;
    TString  pubkey;
};
```

Hinzu kommen Methoden für den Zugriff bzw. die Manipulation der internen Datenfelder. Die verschiedenen Passwortfelder dienen den unterschiedlichen Authentifizierungsmöglichkeiten.

Analog verläuft die Speicherung der Kontendaten. Die hierfür verwendete Klasse enthält folgende Elemente:

```
class TKonto
{
    int                  kontoNr;
    TList<TTransaktion> *aktionen;
    double               kontoStand;
    TAcl                *acl;
};
```

Hierbei ist `TKonto.aktionen` eine Liste aller durchgeführter Transaktionen (Überweisungen etc.). In `TKonto.acl` werden die Berechtigungen für die verschiedenen Zugriffsmöglichkeiten (Lesen, Schreiben etc.) in einer *Access-Control-List* (ACL) gespeichert.

Für jeden einzelnen Datentypen (`TKonto`, `TKunde` etc.) wurden zusätzlich Methoden implementiert, dies es den Objekten gestatten, die in ihnen gespeicherten Daten auf einem dauerhaften Trägermedium (etwa Festplatte) abzuspeichern, bzw. die dort gesicherten Daten einzulesen.

5.3. Client-Server-Kommunikation

In diesem Abschnitt soll der Befehlssatz zur Manipulation der Kunden- und Kontodaten vorgestellt werden. Diese Befehle können vom Kunden an den Kommunikationsserver gesendet werden. Je nach Art des Schlüsselaustausches werden die Daten dabei kodiert oder unkodiert übermittelt.

Befehl	Parameter	Beschreibung
`KTNR`	Integer	Setzt die aktuell gültige Kontonummer
`NAME`	String	„ Kundennamen
`NUMMER`	Integer	„ Kundennummer
`PASSWORT`	String	Setzt das Passwort für Kommunikation ohne RSA
`ZKTNR`	Integer	Setzt die Zielkontonummer für Transaktionen
`ZNAME`	String	„ Zielnamen „
`ZNR`	Integer	„ Zielkundennummer „
`BETRAG`	Float	Setzt den zu transferierenden Betrag
`ZWECK`	String	Setzt den Zweck für eine Transaktion
`ENDE`		Beendet die Kommunikation
`OP RECHTE`		Beginnt Befehl zum Ändern von Rechten (wird mit +/− beendet)
+/−	`ABHEBEN`	Setzt/Löscht das Recht zum Abheben von Geld
	`KONTOAUSZUG`	Setzt/Löscht das Recht für Kontoauszüge
`OP KSTAND`		Liefert aktuellen Kontostand zurück
`OP KAUSZ`		Liefert Kontoauszug zurück
`OP UEBERW`		Tätigt Transaktion mit vorher festgelegten Werten (`ZKTNR, ZNAME, ZWECK, BETRAG`)

5.4. RSA

In diesem Abschnitt soll auf die Implementierung des RSA-Verfahrens eingegangen werden. Diese steht stellvertretend für alle kryptographischen Verfahren (z.B. IDEA, siehe Abschnitt 3.1 und A.1), da die nach außen sichtbare Funktionalität im wesentlichen identisch ist (Ver- bzw. Entschlüsselungsmethoden).

5.4.1. Große Zahlen

Für das RSA-Verfahren wird ein Datentyp für große Zahlen benötigt. Dies sind üblicherweise Zahlen aus $\mathbb{B}^{1024}$ oder noch größer. Um eine möglichst einfache und intuitive Benutzung dieser Zahlen zu ermöglichen, wurde eine Klasse verwendet, die die üblichen mathemtischen Operatoren zur Verfügung stellt und somit der Umgang mit den Objekten im üblichen C++-Stil erfolgen kann.

```
template<int N>
class TBigInteger : public TDynArray {
public:
    // Addition und Subtraktion
    TBigInteger<N> operator += ( const TBigInteger<N> );
    TBigInteger<N> operator -= ( const TBigInteger<N> );

    // Multiplikation, Division und Modulo
    TBigInteger<N> operator *= ( const TBigInteger<N> );
    TBigInteger<N> operator /= ( const TBigInteger<N> );
    TBigInteger<N> operator %= ( const TBigInteger<N> );

    // Methoden zur Ordungsdef.
    const bool operator < ( const TBigInteger<N> ) const;
    const bool operator > ( const TBigInteger<N> ) const;
    const bool operator == ( const TBigInteger<N> ) const;
    const bool operator != ( const TBigInteger<N> ) const;
    const bool operator <= ( const TBigInteger<N> ) const;

    // Potenzrechnung bzgl. Modulo
    TBigInteger<N> powerMod ( TBigInteger<N>,
                              TBigInteger<N> );
};
```

Abgeleitet wurde die Klasse von einer Klasse zur Verwaltung dynamischer Arrays mit der üblichen Funktionalität (siehe Abschnitt A.2).

Durch die Verwendung eines Templates, welches an dieser Stelle für die Basis des Zahlensystems steht, ist man in der Lage, für die verschiedensten Aufgaben angepaßte Zahlentypen zu verwenden. So ist z.B. für das RSA-Verfahren die Basis 2^{15} und könnte ohne Sourceänderung auf leistungsfähigeren Systemen (z.B. 64-bittig) auf 2^{31} erweitert werden.

5.4.2. Die RSA-Klasse

Das RSA-Verfahren wurde ebenfalls in einer Klasse gekapselt. Diese enthält im wesentlichen Methoden zum Ver- bzw. Entschlüsseln.

```
class TRSA {
protected:
    // die verwendeten Schlüssel
    TBigInteger<RSA_BASE>  e, d, n;

public:
    // Methoden zum Verschlüsseln
    TString encryptPublic  ( TString );
    TString encryptPrivate ( TString );

    // Methoden zum Entschlüsseln
    TString decryptPublic  ( TString );
    TString decryptPrivate ( TString );

    // Signieren einer Nachricht
    TString sign ( TString );
};
```

Hierbei ist *TString* eine Klasse zur Verwaltung der C-üblichen Null-terminierten Strings (siehe Abschnitt A.3).

Die kryptographischen Methoden gestalten sich, aufgrund der implementierten Operatoren für große Zahlen als sehr einfach:

```
TString
TRSA::encryptPublic ( TString input )
{
    TBigInteger<RSA_BASE>   msg;
    TString                 string;

    // Umwandlung von String in eine große Zahl
    msg.fromString( input );

    // ausführen des Potenzierens
    msg.powerMod( e, n );

    // Umwandlung in einen String und Rückgabe
    // des Ergebnisses
    return string = msg.toString();
}
```

5.5. Digitales Geld

Bisher war es lediglich möglich, Transaktionen von Geldbeträgen direkt mit der Bank abzuwickeln (z.B. per Überweisung). Es fehlte ein Analogon zum Geld, um auch zwischen verschiedenen Kunden direkt Transaktionen auszuführen. Ein Verfahren, welches dies ermöglicht soll in diesem Abschnitt beschrieben werden.

In Entsprechung zum realen Geld soll auch beim digitalen Geld der Bezug Geld zu Kunde nicht möglich sein bzw. nur in Ausnahmesituationen (Straftatbestand eines Kunden). Diese Anonymität kann auf verschiedenen Wegen erreicht werden. Ein Beispiel wäre die Verwendung von anonymen Konten, wobei allerdings als Bedingung voraus gesetzt werden muß, daß die einzelnen Banken nicht zusammenarbeiten. Anonymes, digitales Geld wäre eine weitere Möglichkeit.

Die Eigenschaften, die von einem Verfahren für digitales Geld gewünscht werden, sind somit:

1. Bezahlung ohne direkte Teilnahme der Bank; Beteiligung der Bank nur beim Ausstellen und Einlösen des Geldes.
2. Unfälschbarkeit: digitale Münzen sollen nur von der Bank ausstellbar sein.
3. Geld kann nicht mehrfach ausgegeben (vom Kunden) bzw. mehrfach eingelöst (vom Geschäft) werden.
4. Fälschliches Beschuldigen von (3) soll nicht möglich sein.
5. Anonymität: Identität des Kunden nicht feststellbar, auch nicht, wenn mehrere Münzen ausgegeben werden.
6. Kunde bei Betrugsversuch aufspürbar.
7. Kundenverfolgung mit Treuhänder doch möglich.
8. Münzverfolgung mit Treuhänder - beim Einlösen wiedererkennen

Allerdings sind einige dieser Punkte oft nicht erfüllbar. So ist etwa (7) und (8) nur bei Onlinesystemen möglich (etwa VISA, Mastercard etc).

Aufbau einer digitalen Münze

Ein erster Ansatz für eine Münze[1] ist:

$$E_{K_{Bank}^{priv}}(\text{Wert}, \text{ID}_{Bank}, \text{Seriennummer})$$

Hierbei gewährleistet die Einführung einer Seriennummer, daß eine Münze nicht mehrfach ausgegeben werden kann. Andererseits ist somit der Bezug von Münze zu Kunde möglich, da die Bank zu jeder Seriennummer die Kundenidentifikation abspeichern kann.

Alternativ könnte man folgenden Ansatz wählen:

$$E_{K_X^{priv}}(\text{Seriennummer}, g(\text{Seriennummer})).$$

Dabei wird der Wert der Münze durch den Schlüssel X bestimmt und g stellt eine Funktion dar, die z.B. eine Bit-Invertierung realisiert. Verwendet wird dieser Münzaufbau in *DigiCash* ('90).

[1] Im folgenden wird der Zusatz "digital" weggelassen. Falls nicht anders gekennzeichnet ist mit einer Münze immer die digitale Form gemeint

Verfahren zur Anonymität

Um die volle Anonymität des Kunden zu gewährleisten, darf die Bank beim Ausstellen, keinerlei Kenntnis über den Inhalt einer Münze haben. Ein Verfahren hierzu stammt von Chaum ('82). Dabei schickt der Kunde eine Nachricht M mit einer Zufallszahl k (dem sog. Blindfaktor) an die Bank. Diese signiert nun die Nachricht, kann aber ohne Kenntnis von k diese nicht dekodieren:

$$\begin{array}{ll} C \to B: & t = m \cdot k^e \bmod n \\ B \to C: & t^d \bmod n = (m \cdot k^e)^d \bmod n = m^d \cdot k \bmod n \end{array}$$

Dabei bilden (e, n) und (d, n) den öffentlichen bzw. den privaten Schlüssel der Bank. Die signierte Münze ergibt sich durch: $S = t^d \cdot k^{-1} \bmod n = m^d \bmod n$.

Allerdings ergibt sich durch das blinde Signieren das Problem, daß die Bank dazu gebracht werden kann, beliebige Nachrichten zu signieren. Dieser Schwachpunkt kann mit dem folgenden Verfahren zumindest mit einer gewissen Wahrscheinlichkeit beseitigt werden:

$$\begin{array}{ll} C \to B: & m_1 \cdot k_1^e \bmod n \\ C \to B: & \vdots \\ C \to B: & m_N \cdot k_N^e \bmod n \\ B \to C: & (m_i \cdot k_i^e)^d \bmod n, \quad i \in \{1, \ldots, N\} \end{array}$$

Es werden N Münzen verschickt. Diese haben alle den gleichen Wert aber verschiedene Seriennummern. Letztere werden alle vom Kunden generiert. Die Bank wählt nun zufällig eine dieser N Münzen aus und fordert die Blindfaktoren aller anderen Münzen an. Falls der Aufbau und der Inhalt dieser Münzen korrekt ist, wird die ausgewählte Münze signiert. Die Wahrscheinlichkeit eines Betruges liegt somit bei $\frac{1}{N}$.

Verhindern von mehrfachem Ausgeben einer Münze

Hierbei soll der Benutzer ermittelt werden können, wenn er eine Münze mehrfach ausgibt. Dazu bekommt er zunächst einen eindeutigen Identifikator $I \in \mathbb{B}^l$, mit z.B. $l = 256$. Dieser wird dann in zwei Teile zerlegt, wobei der Kunde eine Zufallsbitfolge $I^0 \in \mathbb{B}^l$ wählt und $I^1 = I \oplus I^0$ setzt. Durch dieses Verfahren, auch *Secret Splitting* genannt, erhält der Kunde ein neues ID-Paar mit $I = I^0 \oplus I^1$.

Die Komponenten dieses ID-Paares werden nun einzeln so verschlüsselt, daß

(a) nur der Kunde diese entschlüsseln kann und

(b) jeder die Korrektheit der Entschlüsselung feststellen kann.

Die Grundidee dieses sog. *Bit-Commitment-Protocols* ist dabei, daß bei festgelegtem M eine Zufallszahl R gewählt und $(f(R, M), R)$ bekanntgemacht wird. Hierbei ist f eine beliebige (kryptographische) Hashfunktion, z.B. MD5 (siehe Abschnitt 2.2). Die Korrektheit eines bekanntgegebenen M' kann durch Vergleich von $f(R, M')$ mit $f(R, M)$ ermittelt werden.

Diese Methode wird nun in den Münzen angewandt. Seien dazu $R^0, R^1 \in \mathbb{B}^l$ und $P^i = (f(R^i, I^i), R^i)$, $i \in \{0, 1\}$. Eine Münze hat dann folgende Form:

$$E_{K_{Bank}^{priv}}(\text{Wert}, \text{ID}_{Bank}, \text{Seriennummer}, (P_1^0, P_1^1), \ldots, (P_M^0, P_M^1))$$

Die Bank erhält beim blinden Signieren zu allen Münzen, zu denen sie die Blindfaktoren kennt, auch die ID-Paare I_i^0, I_i^1. Die Überprüfung der Korrektheit erfolgt zum einen durch den Vergleich $I_i^0 \oplus I_i^1 = \text{ID}_{Kunde}$ und zum anderen durch den Test $(f(R_i^j, I_i^j), R_i^j) = P_i^j$, mit $j \in \{0, 1\}$.

Beim Geschäftsvorgang, dem Überweisen von Münzen, erfolgen dann folgende Schritte vom Geschäft:

- Überprüfen der Signatur der Münze (siehe Abschnitt 3.2),
- Geschäft $\rightarrow$ Kunde : $b_1, \ldots, b_M, \quad b_i \in \mathbb{B}$ zufällig,
- Kunde $\rightarrow$ Geschäft : $I_1^{b_1}, \ldots, I_M^{b_M}$,
- Geschäft überprüft : $(f(R_i^{b_i}, I_i^{b_i}), R_i^{b_i}) = P_i^{b_i}$.

Während des Einlösens einer Münze bei der Bank, sendet das Geschäft folgende Informationen an die Bank:

$$(\text{Seriennummer}, I_1^{b_1}, \ldots, I_M^{b_M}).$$

Diese überprüft dann, ob es bereits eine Münze mit dieser Seriennumer gibt. Falls nicht, wird die Seriennummer, zusammen mit der ID-Folge gespeicher. Falls die Seriennummer bekannt ist, wird die dort gespeicherte ID-Folge mit der neu erhaltenen Folge verglichen. Existiert ein $i \leq M$ derart, daß I_i^0 und I_i^1 vorliegen, dann kann mittels $I_i^0 \oplus I_i^1$ die Identität des Kunden ermittelt werden. Falls kein solches i existiert besteht die Möglichkeit, daß das Geschäft versucht, eine Münze mehrfach einzulösen.

A. Ausgewählte Klassendefinitionen

In diesem Abschnitt sollen die Definitionen einiger Klassen, die in den vorangegangenen Abschnitten nur erwähnt wurden, vorgestellt werden. Diese sind nicht vollständig und sollen nur einen Überblick über die Funktionalität geben.

A.1. TIdea

```
class TIdea
{
public:
    // creates encryption and decryption subkeys
    void generateSubKeys(const TString & text = "");

    // create the en- de-cryption key-set
    void buildEncryptKeys( const TString & text );
    void buildDecryptKeys( const TString & text );

    TString encode(const TString &in);
    TString decode(const TString &in);
};
```

A.2. TDynArray

```
template <class T>
class TDynArray
{
protected:
    T*    pData;      // the array with the stored elements
    uint  nSize;      // number of the element with the max. index

public:
    // constructor
    TDynArray ( uint size );

    // return size of the vector
    virtual const uint getSize () const;

    // set a new size but keep the old elements
    virtual void setSize ( uint n );

    // access the vector as an array
    virtual T& operator [] ( uint i );

    // copy operator
    virtual TDynArray<T> & operator = ( const TDynArray<T>& vec );
};
```

A.3. TString

```
class TString : public TDynArray<uchar>
{
public:
    // constructor
    TString ( size_t size = 0 );
    TString ( const char string[], const size_t size );

    // access string status
    virtual const size_t length () const;

    // set string length
    virtual void setLength ( size_t n );

    // import (export) from (to) normal strings
    virtual void copyTo ( char buffer[], size_t size ) const;
    virtual void copyFrom ( const char buffer[], size_t size );

    // access substrings [pos,pos+len) and [pos,length)
    virtual TString substr ( size_t pos1, size_t len ) const;
    virtual TString index ( size_t pos ) const;

    // locate substrings or char.
    virtual TString strchr ( char c ) const;
    virtual TString strstr ( const TString & needle ) const;

    // return substrings, seperated by delim
    // (remove all char before delim in string)
    virtual TString strtok ( char delim );

    //
    // various operators for conversion and manipulation
    //

    virtual TString & operator += ( const TString &string );
    virtual TString & operator << ( const TString &string );

    virtual const bool operator == ( const TString &string ) const;
    virtual const bool operator != ( const TString &string ) const;

    virtual TString & operator >> ( char & c );
    virtual TString & operator >> ( uchar & c );

    virtual TString & operator >> ( int & n );
    virtual TString & operator >> ( long & n );

    virtual TString & operator >> ( float & f );
    virtual TString & operator >> ( double & f );

    virtual TString & operator <<= ( size_t n );
    virtual TString & operator >>= ( size_t n );
};
```

A.4. TList

```
template <class T>
class TList : public TMemory
{
protected:
    TItem<T>  *pFirst; // pointer to the first element
    TItem<T>  *pLast;  // and to the last one
    int    nSize;      // act. size of the list

public:
    // constructors
    TList ();

    // return the number of items stored
    const int getSize () const;

    // add a new object to the list
    virtual void add ( T &object );

    // add a new object to the begin of the list
    virtual void insert ( T &object );
    virtual void insertAfter ( T &after, T &object );
    virtual void insertBefore ( T &before, T &object );

    // replace one item by another
    virtual void replace ( T &oldObject, T &newObject );

    // remove item from list (access by element)
    // (NOT THE ELEMENT HIMSELF !!!)
    virtual void remove ( T &object );

    // remove the item (access by container class)
    virtual void remove ( TItem<T> &item );

    // search for an object and return the item
    virtual TItem<T>* search ( T &object );

    // clear the list (NOT THE ELEMENTS !!!)
    virtual void clear ();
};
```

Literatur

[DGV] Joan Daemen, René Govaerts, and Joos Vandewalle. Weak keys for idea.

[LM91] X. Lai and J.L. Massey. A proposal for a new block encryption standard. In *Advances in Cryptology-Eurocrypt*, pages 389–404. Springer-Verlag, Berlin, 1991.