Studienarbeit zum Thema

Real time & Fault Tolerant Techniques for Security

Bernhard Rabe
rabe@informatik.hu-berlin.de
Matrikel 143377

Lehrstuhl für Rechnerarchitektur und Kommunikation
Institut für Informatik
Mathematisch-Naturwissenschaftliche Fakultät II
Humboldt-Universität zu Berlin

16. Oktober 2001

Betreuer:
Dr. Andreas Polze

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Einleitung und Motivation

Palmtop-Computer, Mobiltelefone, elektronische Datenbanken und andere tragbare elektronische Geräte erfreuen sich wachsender Beliebtheit. Sie beinhalten schon heute viele persönliche und sicherheitsrelevante Informationen, wie Telefonnummern, Email, Adressen und Geburtstage. Im Laufe der Zeit werden *Personal Digital Assistants* (PDA) mehrere solcher Funktionen übernehmen, z.B. Mobiltelefon, Kalender, Datenbank, Notizbuch, Internetzugang (*SmartPhones*). Solche Anwendungsfälle enthalten mehr oder minder zu schützende Daten. Doch wenn PDA's auch als Zugangskontrolle oder gar als Schlüssel für zu schützende Inhalte benutzt werden, stellt sich die Frage, wie solche Daten auf einem PDA's vor Missbrauch zu schützen sind. Die einfachste Antwort ist ein Passwort. Doch ist es sinnvoll ein (oder mehrere) Passwort mit einem anderen Passwort zu schützen? Der Ansatz der hier Mittelpunkt steht, ist es die Daten so zu speichern das mit einem solchen PDA's keine Möglichkeit besteht die Daten zu Missbrauchen. Es stellt sich also die Frage wie man Schlüssel im weitesten Sinne auf mehrere PDA's so verteilen kann, das eine gewisse Anzahl der PDA's nötig ist um die Information oder Schlüssel zu nutzen.

1.2 Ziel der Arbeit

Ziel dieser Arbeit ist es ein für eine festgelegte Situation, in diesem Fall **Der elektronische Tresor**, zu untersuchen wie man Daten (Schlüssel) so mit mobile Geräten (PDA's) nutzen kann, das diese möglichst sicher gegen Angriffe sind. Dazu sollen Fehlertoleranz- und Echtzeittechniken eingesetzt werden. Als Ergebnis soll eine Softwarelösung entstehen die das beschriebene Szenario realisiert und es ermöglicht diese Techniken auf ihre Eignung für solche Anwendungen zu testen.

1.3 Verwandte Arbeiten

Im Anschluß werden einige Arbeiten vorgestellt die es zum Ziel Daten verteilt sicher zu speichern. Dabei werden stationäre (geografisch getrennte) Systeme betrachtet. Doch die Ansätze kann man auf diese Arbeit anwenden, da es sich hier nur um andere Formen von Verbindungen handelt, logische statt physischer, da mobile Geräte betrachtet werden.

1.3.1 Secure Distributed Data Repository

Ziel ist es einen verteilten Datenspeicher zu realisieren, welcher auch unter dem Namen *electronic vault* (e-vault) bekannt ist und die Daten sicher über ein Netzwerk verteilt, so das die Integrität gewährleistet ist selbst wenn einige Server Fehlfunktionen haben. Die Server können dabei auch geografisch verteilt angeordnet werden um geografisch begrentzte Ausfälle zu kompensieren (Katastrophen).
Wenn die Information über verschiedene Maschinen (Server) verteilt wird, so erhält der Speicherplatz der Information eine wichtige Bedeutung. Der einfachste Ansatz ist die Information vollständig an alle Server zu übertragen. Dadurch hat man n-fache Redundanz bei n Servern. Andere Algorithmen zur Verteilung von Informationen mit geringeren Platzoverhead sind bekannt, dazu gehört auch der *Information Dispersal Algorithm* (IDA) von

M. Rabin 2.2, [1]. Ein anderes Problem bei der Informationsreplikation ist das die Information erhalten werden kann, wenn man in einen Server einbrechen kann der die replikierte Information enthält.
In [12] beschreibt das Design und die Implementation eines *online, distributed data repository* (e-vault), welcher die Informationen sicher über mehrere Server mit Hilfe des IDA (2.2) verteilt. Das System besteht aus einem *Gateway* (GW), welcher die Anfragen von Clienten entgegennimmt und die Information an die internen Server weitergibt und von ihnen erhält. Um zu überprüfen ob alle Server ein gültiges IDA-Teil besitzen wird eine verteilte digitale Signatur eingesetzt. Jeder Server trägt seinen Teil der Signatur durch einen Hashwert der erhaltenen Information bei. So kann der GW prüfen ob genügend Server zum Wiederherstellen der Information vorhanden sind und welche das sind.

1.3.2 AIDA-based Real-Time Fault-Tolerant Broadcast Disks

[10] stellt ein *Broadcast Disks*-Protokoll vor auf Basis des *Adaptive Information Dispersal Algorithm* [1] (AIDA)[11] vor. *Broadcast Disks* ist ein Mechanismus, der Kommunikationsbandbreite nutzt um ein Speichermedium zu emulieren. Dabei werden kontinuierlich und wiederholt Daten an die Klienten gesendet, was den Effekt hat, daß der *Broadcast*-Kanal als Menge von Speichern (disks), von denen Klienten Daten holen, agiert. Der AIDA erlaubt es, den Kompromiß zwischen Bandbreite und Zuverlässigkeit zu kontrollieren. Es wird gezeigt, wie AIDA Rechtzeitigkeit und Fehlertoleranz Eigenschaften garantiert werden können. Den Einsatz von AIDA läßt sich am einfachsten mit einem Beispiel erklären (Abildung 1.1).
Wir haben 2 Dateien A und B, welche periodisch übermittelt werden. Annahme A bestehe aus 5 Blöcken $A_1 \dots A_5$ und B aus 3 Blöcken $B_1 \dots B_3$. Die *Broadcast*-Periode für diese *Broadcast Disk* beträgt 8 Zeiteinheiten bis ein Block wiederholt wird.

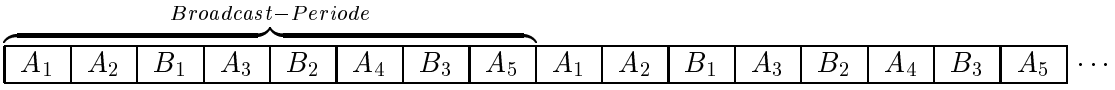

Abbildung 1.1: *Beispiel Broadcast Program*

Jetzt werden A und B nun mit AIDA verteilt, so das A in 10 Teile, von denen 5 zum Rekonstruieren benötigt werden und B in 6 Teile, von denen 3 zum Rekonstruieren werden. In Abbildung 1.2 ist ein *Broadcast Program* zu sehen welches A und B periodisch durch ihre Blöcke repräsentiert sendet. Die erste *Broadcast*-Periode dauert immer noch 8 Zeiteinheiten. Jede *Broadcast*-Periode enthält genügent Blöcke um A und B vollständig zu Rekonstruieren. Durch AIDA werden jetzt in verschiedenen *Broadcast*-Perioden verschiedene Blöcke von A und B gesendet, so das mit der 2. *Broadcast*-Periode ein Programm-Daten Zyklus entsteht, der alle Blöcke von A und B enthält. Wenn jetzt ein Fehler ein Fehler in einem Block auftritt, so dauert es jetzt maximal 2 Zeiteinheiten bis Ersatz übermittelt wird.

Bei einer größeren Anzahl von Dateien kann man mit den Bedingungen für n (Anzahl aller Teile) und m (benötigte Anzahl) zwischen wichtigen und weniger wichtigen Informationen unterscheiden und ihre Fehlererholungzeiteinheiten beeinflussen.

[1] Es handelt sich um einen zustandsabhängigen Rabin-IDA (2.2, [1]), welcher in Abhängigkeit von Fehlererholung, Bandbreite usw. die Art der Zerlegung bestimmt.

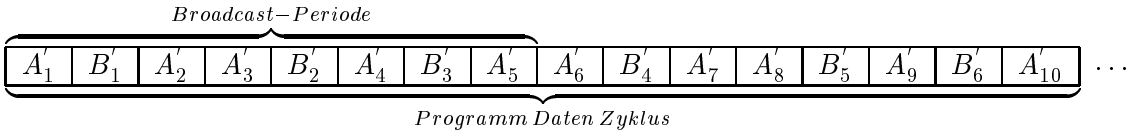

Abbildung 1.2: *Beispiel Broadcast Program mit AIDA*

1.3.3 Secure Distributed Storage and Retrieval

Ziel dieser Arbeit[13] ist es die Verteilung der Daten mittels Rabin-IDA [1] schon beim Verteilen zu sichern. Denn IDA ist fehlertolerant beim Wiederherstellen der Information. Es wird garantiert, das nur der Eigner berechtigt ist die Information zu erhalten. Das wird durch den Einsatz von Kryptografietechniken wie *Blinding* und *Thresold Cryptography* erreicht. *Blinding* kann wie folgt erklärt werden. Ein Server hat einen geheimen Schlüssel DK, welcher Entschlüsselung im *Public Key*-Verschlüsselungschema $\mathbf{E}$ erlaubt. Ein Verschlüsselungschema ist *Blindable* wenn die Funktionen $\mathbf{E}_{EK}$ und $\mathbf{E}_{DK}$ homomorph sind: $\mathbf{E}(ab) = \mathbf{E}(a)\mathbf{E}(b)$. Die Struktur entspricht der von 1.3.2, einem *Gateway* (GW) und interne Server. Beim Abgeben einer Information erhält der Klient einen Beweis, dafür das die Verteilung erfolgreich war.

1.4 Gliederung

Im Anschluß an dieses Kapitel wird ein Überblick über die verwendeten Algorithmen und Verfahren gegeben. Dazu zählen vorallem die Verteilungsalgorithmen von Shamir 2.1 und Rabin 2.2. Desweiteren werden für die Algorithmen benötigte mathematische Grundlagen gegeben. Im Anschluß wird der betrachtete Anwendungsfall in Kapitel 3 und die damit verbundene Implementationslösung in Kapitel 4 erläutert. Den Schluß bilden einige Implementationsdetails in Kapitel 5 und ein Ausblick auf weiterführende Arbeit in Kapitel 6.

2 Grundlagen

Das Prinzip der Verteilungsalgorithmen läßt sich am einfachsten an einem Beispiel erklären. Shamir [2] motiviert das mit folgendes Szenario:

i. 11 Wisssenschaftler arbeiten an einem geheimen Projekt.

ii. Sie wollen ihre Arbeit so sichern, daß diese nur geöffnet werden kann, wenn 6 oder mehr Wissenschaftler anwesent sind.

Welche ist die kleinste Zahl von Schlüsseln, die jeder Wissenschaftler für die Sperren haben muß? Die minimalste Lösung braucht 462 Sperren und 252 Schlüssel pro Wissenschaftler. Dieses Zahl der Schlüssel und Sperren ist unpraktisch und steigt exponential, wenn die Zahl der Wissenschaftler sich erhöht. Im folgendem sind Techniken und Methoden erläutert, die ich in dieser Arbeit verwendet habe.

2.1 Shamir's Secret Sharing

Das Geheimnis seien irgendwelche Daten D. Das Ziel ist es D in n Teile $D_1, \ldots, D_n$ zu teilen in der Art, das [2]:

(1.) Die Kenntnis von irgendwelchen k oder mehr D_i Teilen, ermöglicht D einfach zu berechnen.

(2.) Die Kenntnis von $k-1$ oder weniger D_i Teilen gibt keine Information über D.

Ein solches Schema nennt man (k, n) *Threshold Scheme*. Shamir's Schema basiert auf Polynominterpolation. Wir haben ein Polynom $q(x)$ vom Grad $k-1$, so daß $q(x_i) = y_i$ für alle i. Ohne Einschränkungen kann man D in Zahlendarstellung annehmen. Um D in Teile D_i zu teilen, wählt man zufällig ein Polynom vom Grad $k-1$ $q(x) = a_0 + a_1 x + \ldots + a_{k-1} x^{k-1}$, wobei $a_0 = D$ und berechnen:

$$D_1 = q(1), \ldots, D_i = q(i), \ldots, D_n = q(n).$$

Mit irgendeiner Menge k dieser D_i's (mit ihren Indizies), kann man die Koeffizienten von $q(x)$ mittels Interpolation und $D = q(0)$ berechnen. Um das Polynom interpolieren zu können braucht man einen Zahlenbereich indem das möglich ist. Im einfachsten Fall können das die Zahlen modulo einer Primzahl p sein, wobei p größer als D und n sein muß. Die Koeffizienten $a_1, \ldots, a_{k-1}$ in $q(x)$ werden zufällig über [0,p) gewählt und die Werte von $D_1, \ldots, D_n$ modulo p berechnet. Mit Hilfe des Lagrange Formalismus und den Indizies i der D_i's kann man die Koeffizienten $a_1, \ldots, a_{k-1}$ und dann $q(0) = D$ gerechnen. Alle Berechnungen müßen dabei modulo p berechnet werden. Eigenschaften des (k, n) *Threshold Scheme's*:

(1) Jedes Teil hat die Größe der orginalen Daten

(2) Wenn k fest gewählt wird, so können D_i dynamisch angefügt oder gelöscht werden

(3) Es ist einfach D_i's neu zu wählen ohne D zu ändern, indem ein neues Polynom $q(x)$ mit gleichen a_0 gewählt wird.

2.2 Rabin's Information Dispersal Algorithm

Die Information $D = b_1, \ldots, b_N$ sei eine Folge von Zeichen *(Zahlen)*. Ziel ist es D in n Teile aufzuteilen, so daß mit $k = n - m$ Teilen Verlust die Information D vollständig wiederhergestellt werden kann [1]. Als Zahlenbereich für die Berechnungen kann man wieder die Reste modulo p (p=Primzahl, $p > b_i$ mit $i \in \{1, \ldots, N\}$) also $\mathbb{Z}_p$ benutzen. Ein anderer günstigerer Zahlenbereich ist ein Galois Field $GF(2^s)$, welcher später genauer erlaütert wird. Jetzt beschränken wir uns auf $\mathbb{Z}_p$. Wir wählen n Vektoren $a_i = (a_{i1}, \ldots, a_{im}) \in \mathbb{Z}_p^m$, $1 \leq i \leq n$, so daß m verschiedene Vektoren linear unabhängig sind. Wie das zu erreichen

ist werde ich später anhand der *Vandermonde Matrix* erläutern.
D wird in Sequenzen der Länge m zerlegt, so daß:

$$D = (b_1, \ldots, b_m), (b_{m+1}, \ldots, b_{2m}), \ldots$$

$S_i = (b_{(i-1)m+1}, \ldots, b_{im})$ mit $i = 1, \ldots, n$ und $D_i = c_{i1}, c_{i2}, \ldots, c_{iN/m}$ wobei

$$c_{ik} = a_i \cdot S_k = a_{i1} \cdot b_{(k-1)m+1} + \cdots + a_{im} \cdot b_{km}$$

$$\Longrightarrow \qquad \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} \cdot \begin{pmatrix} S_1 \\ S_2 \\ \vdots \\ S_{N/m} \end{pmatrix}^T = \begin{pmatrix} D_1 \\ D_2 \\ \vdots \\ D_n \end{pmatrix}.$$

Wenn wir m Teile von D erhalten haben, sagen wir $D_1, \ldots, D_m$, so können wir D rekonstruieren. Sei dabei $A = (a_{ij})_{1 \le i,j \le m}$ eine $m \times m$ Matrix, in der die $i$te Spalte a_i ist. Dann ist:

$$\begin{pmatrix} S_1 \\ S_2 \\ \vdots \\ S_{N/m} \end{pmatrix}^T = A^{-1} \cdot \begin{pmatrix} D_1 \\ D_2 \\ \vdots \\ D_m \end{pmatrix}$$

Der wohl entscheidenste Unterschied zu *Shamir's Secret Sharing*[2](2.1) ist die Länge eines Teils. Beim IDA ist die Länge eines $D_i = |D|/m$. So hat man bei m Teilen genau die Länge des Originals (bis auf Statusinformationen). So ist für $n/m \sim 1$ der IDA längeneffizient.

Vandermode Matrix

Die *Vandermonde* Matrix hat die Form [3]

$$A = \begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 & \ldots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & x_2^3 & \ldots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & x_n^3 & \ldots & x_n^{n-1} \end{bmatrix} = [x_i^{j-1}]_{1 \le i,j \le n}$$

Abbildung 2.3: *Vandermonde Matrix*

wobei $x_i \in \mathbb{R}$ und $x_i - x_j$ invertierbar für $i \neq j$ (die x_i's seien eindeutig). Wie man leicht sieht sind die Vektoren alle linear unabhängig.

Bei den Daten in heutigen Rechnern handelt es sich fast ausschließlich um Byte-Zeichenketten, also um 8-Bit Symbole. Deshalb ist es ungünstig als Bildbreich für die vorgestellten Algorithmen $\mathbb{R}$ zu verwenden. Das würde unnötigen Aufwand beim Lesen und Schreiben der

Daten und nicht zuletzt höheren Speicherbedarf verursachen. Als Lösung wäre wie schon erwähnt $\mathbb{Z}_p$ denkbar. Da aber p eine Primzahl sein soll und die kleinste Primzahl$\geq$255 257 ist, erhält man Reste $[0 \ldots 256]$, was den 8-Bit Rahmen sprengt. Eine andere Lösung für einen Zahlenbereich wird im folgenden vorgestellt.

Galois Field

Ein **Galois Field** $GF(2^s)$ enthält die Zahlen $0 \ldots 2^s - 1$. Addition und Subtraktion von Elementen aus $GF(2^s)$ sind sehr einfach. Sie werden mit der XOR-Operation realisiert. Zum Beispiel im $GF(2^8)$:

$$11 + 7 = 00001011 \oplus 00000111 = 00001100 = 12$$

$$11 - 7 = 00001011 \oplus 00000111 = 00001100 = 12$$

Abbildung 2.4: *Addition und Subtraktion über* $GF(2^8)$.

Multiplikation und Division sind etwas komplexere Operationen. Für $s \leq 16$ kann man 2 Logarithmustabellen benutzen, jede mit der Länge $2^s - 1$ um die Multiplikation und Division durchzuführen:

- `gflog`: Diese Tabelle ist für die Indizies $1 \ldots 2^s - 1$ definiert, und ordnet den Indizies ihren Logarithmus im Galois Field zu.
- `gfilog`: Diese Tabelle ist für die Indizies $0 \ldots 2^s - 2$ definiert, und ordnet den Indizies ihren inversen Logarithmus im Galois Field zu, so daß gilt:
  `gflog[gfilog[i]]=i` und `gfilog[gflog[i]]=i`

Mit den 2 Tabellen (Abbildung 2.5), kann man 2 Elemente aus GF(2^8) Multiplizieren indem man ihre Logarithmen addiert. Der Wert des inversen Logarithmus mit dem Index der Summe ist dann das Produkt. Um 2 Zahlen zu Dividieren, subtradiert man ihre Logarithmen. Die folgende Implementation nutzt die Tatsache das der inverse Logarithmus einer Zahl i ist gleich dem inversen Logarithmus von (i mod $(2^s - 1)$).

i	0	1	2	3	4	5	6	7	8	9	10	11	12		255
gflog	—	0	1	25	2	50	26	198	3	223	51	238	27	...	175
gfilog	1	2	4	8	16	32	64	128	29	58	116	232	205		—

Abbildung 2.5: `gflog` *und* `gfilog` *für* $GF(2^8)$.

Man muß den Logarithmus von 0 als Spezialfall betrachten, denn dieser ist $-\infty$.

Die Tabellen `gflog` und `gfilog` werden mit folgendem Algorithmus erstellt:

```
#define NW 1<<8
#define prime_poly 0435
Byte *gflog, *gfilog;

void setup_tables(){
  Byte log, b=1;
  gflog = new Byte[NW];
  gfilog = new Byte[NW];
  for(log=0; log < NW-1; log++){
    gflog[b] = log;
    gfilog[log] = b;
    b = b<<1;
    if(b & NW) b = b ^ prim_poly;
  }
}
```

Abbildung 2.6: *C++ Quelltext zur Berechnung von* `gflog` *und* `gfilog`.

Multiplikation:	Division

`#define NW (1<<8)`
`typedef unsigned char Byte;`
`Byte mul(Byte a, Byte b){`
`  int sumlog;`
`  if(a==0 || b==0) return 0;`
`  sumlog=gflog[a]+gflog[b];`
`  if(sumlog >= NW-1) sumlog-=NW-1;`
`  return gfilog[sumlog];`
`}`
	`#define NW (1<<8)`
	`typedef unsigned char Byte;`
	`int div(Byte a, Byte b){`
	`  int difflog;`
	`  if(a==0) return 0;`
	`  if(b==0) return -1;`
	`  difflog=gflog[a]-gflog[b];`
	`  if(difflog < 0) difflog+=NW-1;`
	`  return gfilog[difflog];`
	`}`

Abbildung 2.7: *Multiplikation und Division über* $GF(2^8)$.

$$5 * 6 = \texttt{gfilog}[\texttt{gflog}[5] + \texttt{gflog}[6]] = \texttt{gfilog}[50 + 26] = \texttt{gfilog}[76] = 30$$

$$9 * 11 = \texttt{gfilog}[\texttt{gflog}[9] + \texttt{gflog}[11] = \texttt{gfilog}[223 + 238]] = \texttt{gfilog}[206] = 83$$

$$5 \div 6 = \texttt{gfilog}[\texttt{gflog}[5] - \texttt{gflog}[6]] = \texttt{gfilog}[50 - 26] = \texttt{gfilog}[24] = 143$$

$$9 \div 11 = \texttt{gfilog}[\texttt{gflog}[9] - \texttt{gflog}[11]] = \texttt{gfilog}[223 - 238] = \texttt{gfilog}[240] = 44$$

Abbildung 2.8: *Beispiel für Multiplikation und Division über* $GF(2^8)$.

2.3 Kryptographie

Dieser Abschnitt soll einen kurzen Überblick über die in dieser Arbeit relevanten Kryptographietechniken geben. Kryptographie erlaubt es Daten vor unberechtigten Zugriff zu schützen, die Integrität von Daten zu prüfen und Authentifikation als eine Möglichkeit der Identifikation. Der Hauptaugenmerkt liebt hier jedoch bei den Verschlüsselungstechniken.

2.3.1 Symmetrische Verschlüsselungstechniken

Bei symmetrischen Verfahren werden für das Ver- und Entschlüsseln von Nachrichten der gleiche symmetrische Schlüssel verwendet. Vorteil der symmetrischen Verfahren ist ihre Einfachheit und die daraus resultierende Geschwindigkeit beim Ver- und Entschlüsseln. Der Nachteil dieser Verfahren ist die Verteilung der Schlüssel. Jeder der in Besitz des Schlüssel kann alle Nachrichten zwischen Parteien die mit diesem Schlüssel verschlüsseln mithören. Deshalb verlangen diese Verfahren nach einer sicheren Verteilung z.B. mit Hilfe asymmetrischer Verfahren (2.3.2).
Symmetrische Verfahren sind zum Beispiel *Data Encryption Standard* (DES), RC2 und RC4 [4].

2.3.2 Asymmetrische Verschlüsselungstechniken

Hierbei werden für das Ver- und Entschlüsseln unterschiedliche Schlüssel verwendet. Das unter dem Namen Verschlüsselungsverfahren mit öffentlichen Schlüsseln bekannte Verfahren (*Public Key Encryption*) arbeitet mit 2 Schlüsseln, dem öffentlichen Schlüssel (*Public Key*) zum Verschlüsseln einer Nachricht an den Empfänger und dem privaten Schlüssel (*Private Key*) zum Entschlüsseln der Nachricht am Empfänger. Die Kenntnis nur des öffentlichen Schlüssels macht es nicht Möglich eine Nachricht zu Entschlüsseln, dazu wird der private Schlüssel benötigt.
Um eine Kommunikation zu sichern erzeugt jede Partei ein Schlüsselpaar (privater/öffentlicher), um dann den öffentlichen Schlüssel an jede Partei zu versenden. Wenn eine Nachricht an eine Partei gesendet werden soll, so wird diese mit dessen öffentlichen Schlüssel verschlüsselt. Nur der Empfänger mit dem passenden privaten Schlüssel kann die Nachricht Entschlüsseln. Um die öffentlichen Schlüssel bekannt zu machen werden diese häufig an zentraler Stelle veröffentlicht.
Das wohl verbreiteste asymmetrische Verfahren ist *RSA* (Rivest, Shamir, Adleman)[4]. *RSA* basiert auf der Komplexität der Primfaktorenzerlegung, welche heute nicht in akzeptabler Zeit berechenbar ist. Deshalb sind asymmetrische Verfahren im Vergleich zu symmetrischen Verfahren (2.3.1) um den Faktor 1000...10000 langsamer.

2.3.3 Secure Socket Layer

Das SSL-Protokoll soll eine sichere Kommunikation über das Internet ermöglichen. Prinzipell setzt sich SSL aus einem asymmetrischen Teil (2.3.2) und dem symmetrischen Teil (2.3.1) zusammen.

1. Der Klient baut eine Verbindung zum Server auf und macht ihn über die von unterstützten Verschlüsselungsverfahren und Hashfunktionen bekannt. Der Server

wählt aus diesem das für ihn geeignete Verfahren aus.

2. Der Server übermittelt dem Klienten ein Zertifikat mit seinem öffentlichen Schlüssel.

3. Der Klient erzeugt eine Zufallszahl und verschlüsselt diese mit dem öffentlichen Schlüssel und sendet sie an den Server. Aus dieser Zahl erzeugen sowohl Server und Klient einen symmetrischen Schlüssel für ein symmetrischen Verschlüsselungsverfahren (2.3.1).

4. Die Authentifizierung erfolgt indirekt durch die Erzeugung des symmetrischen Schlüssels, denn nur wenn er den richtigen privaten Schlüssel besitzt kann er die Zahl richtig entschlüsseln.

5. Jetzt können beide Parteien Nachrichten Ver- und Entschlüsseln.

Der Vorteil von SSL ist es ohne eine 3. Partei auszukommen, die öffentlichen Schlüssel halten muß. Als Nachteil ist der Punkt zu sehen, das nicht (ohne 3.Partei) überprüft ob der Schlüssel authentisch ist.

2.4 Microsoft Windows CE

Windows CE[15] als Betriebssystem für eingebette System, ist für viele Plattformen und Anwendungen erhältlich. Es unterstützt neben einer Teilmenge der Win32 API, der Programmierschnittsteller alle 32-Bit Windows Desktop-Betriebssysteme auch eingeschränkte Echtzeitfunktionalität. Neben dem Betrieb auf den in dieser Arbeit verwendeten PDA's, ist also in Zukunft eine Verwendung anderer mobiler Geräte denkbar, ohne die Software an eine neue Programmierschnittstelle anpassen zu müssen.

2.5 IrDA-Standard

Die *Infrared Data Association* (IrDA) ist eine Industriell basierte Gruppe von über 150 Firmen, die Kommunikationsstandards speziell für *low cost*, kurze Distanzen, viele Plattformen, Punkt zu Punkt Kommunikation für einen großen Geschwindigkeitsbereich entwickelt haben. Diese Standards sind auf vielen Plattformen implementiert und für viele eingebettete Anwendungen verfügbar. Der IrDA-Stack (Abbildung 2.5) setzt sich aus mehreren Schichten zusammen, die jeweils die darüber und darunter liegenden Schichten benutzen. Diese Schichten teilen sich auf in benötigte und optionale Schichten.

Benötigte IrDA-Protokolle

- *Physical Layer*: Spezifiziert optische Eigenschaften, Codierung der Daten und Rahmengröße für verschiedene Geschwindigkeiten.

- IrLAP: *Link Access Protocol*. Stellt die grundlegende zuverlässige Verbindung her.

- IrLMP: *Link Managment Protocol*. Multiplexed Dienste und Anwendungen einer IrLAP Verbindung.

- IAS: *Information Access Service*. Stellt „gelbe Seiten“der Dienste eines Gerätes zur Verfügung.

Optionale Protokolle

Die Verwendung der optionalen Protokolle sind abhängig von der speziellen Anwendung.

- TinyTP: *Tiny Transport Protocol*. Fügt für einen Kanal Flusskontrolle hinzu. Das ist eine wichtige Funktion und wird in vielen Fällen benötigt.
- IrORBEX: *The Object Exchange Protocol*. Einfacher Transfer von Dateien und anderen Datenobjekten.
- IrCOMM: Seriell- und Parallelportemulation, ermöglicht es bestehenden Anwendungen die serielle oder parallele Kommunikation nutzen Infrarot ohne Änderung zu verwenden.
- IrLAN: *Local Area Network*.

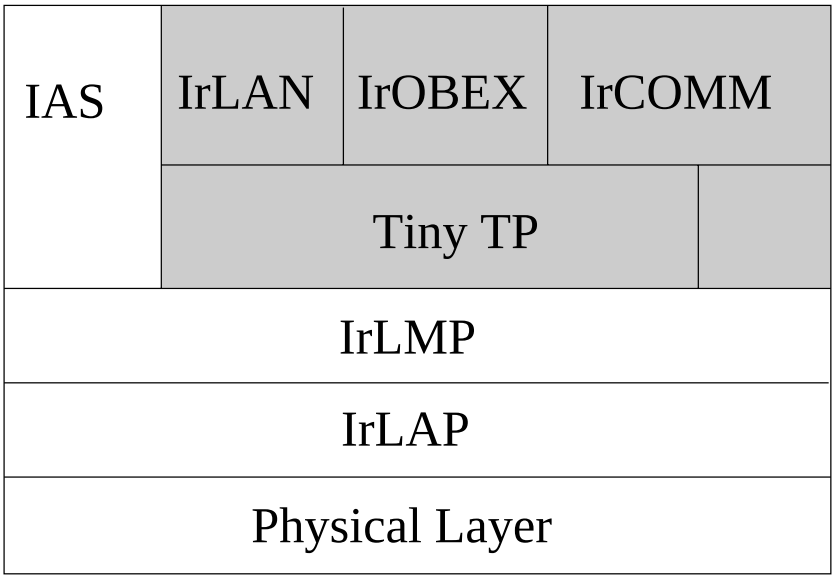

Abbildung 2.9: *IrDA-Protokollstack*

Aus dieser Situation ergaben sich auch die Entscheidungen für die Realisierung, denn *IrLAN* ist nicht standardiesiert und unter nicht Windows implementiert. Die einzelnen Kommunikationsmöglichkeiten werden in 4.2 erläutert. Andere Probleme ergaben sich aus der Realisierung der einzelnen Dienste. So ist es nicht möglich ein Gerät eindeutig an seiner DeviceID zu identifizieren, da diese dynamisch im *IrLAP* festgelegt wird und man keinen Einfluß auf diese hat. Desweiteren ist es mittels der WIN32-API nicht möglich zur Laufzeit seine eigene DeviceID zu erfahren, denn der dafür notwendige *IAS*-Dienst gibt keine Informationen über das eigene Gerät aus (was einem Telefonbuch widerspricht). Eine andere Einschränkung ist die Beschränkung auf Punkt zu Punkt Verbindungen innerhalb des Standards.

3 Anwendungsfall - Elektronischer Tresor

Im folgenden sollen die Konzepte der erstellten Softwarelösung erläutertet werden.

3.1 Szenario

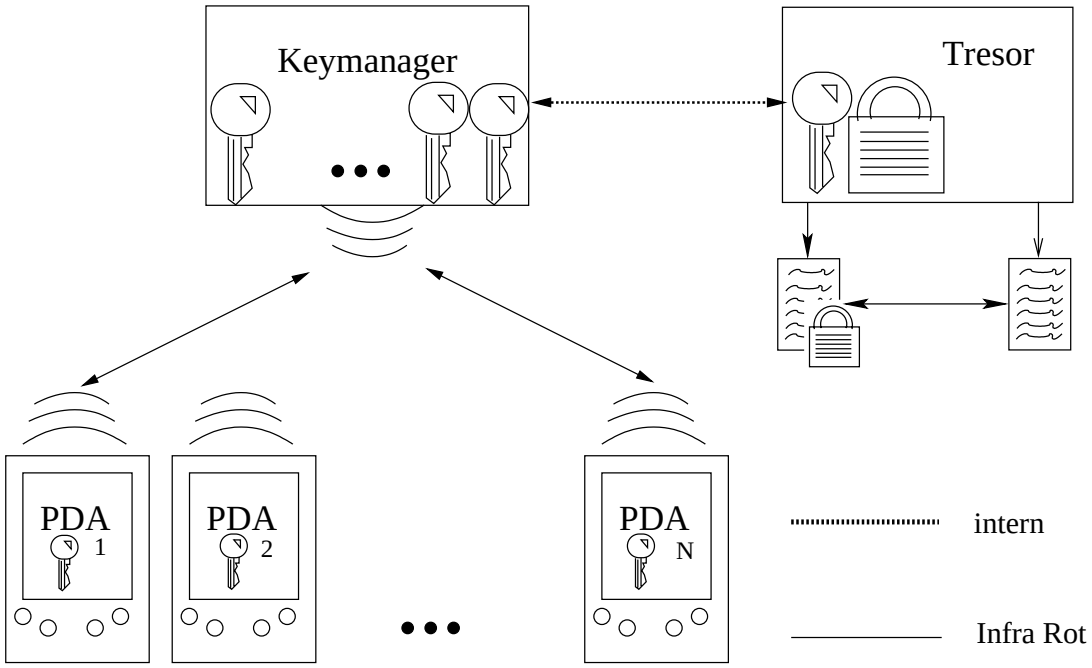

Abbildung 3.1: *Schema elektronischer Tresor*

Als Beispielanwendung soll eine elektronischer Tresor implementiert werden. Der für einen Tresor üblichen Schlüssel oder Zahlenkombinationen sollen in diesem Fall durch einen elektronischen Schlüssel (genau genommen eine Zeichenkette) realisiert werden. Der übliche Weg ist das den (oder die) Schlüssel ein Verantwortlicher beisich trägt. Man hat in diesem Fall bei einem Verlust keine Möglichkeit den Tresor zu öffnen. Deshalb wird in der Beispielanwendung mit Hilfe der im Kapitel 2.1 und 2.2 beschriebenen Algorithmen, der Schlüssel mit einer variablen Redundanz ($n \leq m$) versehen und dann diese Teilschlüssel an die n Personen weitergegeben. Die Personen sind in diesem Fall PDA's. Das heißt die Schlüsselträger sind mobil und deshalb sollte die Übertragung der Teilschlüssel auch drahtlos funktionieren. Im Anwendungsfall wurde als drahtlose Kommunikation Infrarot eingesetzt[2]. Der Tresor könnte natürlich auch mobil sein, zum Beispiel könnte eine Kurier so Daten transportieren. Im Anwendungsfall wird der Tresor durch einen Desktopcomputer realisiert, welcher auch eine Infrarotschnittstelle besitzt. Die Kommunikation mit den PDA's wird durch den KeyManager realisiert, der auch das Zerlegen und Zusammensetzen der Teilschlüssel erledigt. Die eigentliche Tresorkomponente liefert und empfängt den Schlüssel vom KeyManager. Diese Kommunikation wird nicht über Infrarot sondern

[2]es sind auch andere Kommunikationsschnittstellen vorstellbar z.B. WLAN, GSM, Bluethoots

intern realisiert[3]. Um den Schlüssel wiederherstellen zu können (den Tresor zu Öffnen) werden m ($n \leq m$) Schlüsselträger benötigt. Den Inhalt des Tresors stellen dabei mit dem Tresorschlüssel verschlüsselte Dateien dar. Nach dem Verteilen des Tresorschlüssels gibt es ohne m Teilschlüssel keine Möglichkeit die Daten zu entschlüsseln. Bei der Übertragung der Schlüsselteile sind 2 Szenarien denkbar:

1. Alle Schlüsselteile werden gleichzeitig an die PDA's Übertragen, wobei alle innerhalb einer bestimmten (kurzen) Zeit Antworten müssen ob die Übertragung erfolgreich war.

2. Die Schlüsselteile werden nacheinander an die PDA's Übertragen, wobei festgelegt werden muß wieviele PDA's überhaupt Schlüsselteile holen und wie lange die ganze Übertragung insgesamt dauer darf bevor eine unsichere Situation eintritt (Diebstahlversuch).

1. parallele Übertragung

Auf den ersten Fall werde ich nur theoretisch eingehen, da die vorhandenen Hardware ein solches Szenario nicht zuließ [4]. Da m parallele Verbindungen zum KeyManager bestehen, kennt dieser die Identitäten aller Schlüsselträger vor der eigentlichen Übertragung. Der KeyManager kann den PDA's den Befehl zu übermitteln der Teilschlüssel geben und einen Timer für die Übertragung starten. Nach dem dieser Valid-Timer abgelaufen ist müssen alle Teilschlüssel übertragen worden seien, sonst handelt es sich um einen Fehlversuch. Bei einem Fehlversuch wird vermerkt werden welche PDA's die Zeit überschritten haben, beziehungsweise nicht geantwortet haben. So könnte man diese PDA's als unsicher erklären und von weiteren Verbindungen ausschließen. Um sicherzustellen, das alle Teilschlüssel gültig sind wird die digitale Signatur geprüft, die vor dem Verteilen angebracht wurde. Auch im Fall einer ungültigen Signatur kann anhand der Identität ein Fehler oder ein neu Übertragen, oder eine Ersatzübertragen veranlaßt werden.

2. seqentielle Übertragung

Der zweite Fall ist der, den ich hier näher untersucht habe. Bedingt durch die vorhandene Hardware sind nur IrDA-Verbindungen möglich und die sind seriell (siehe 4.2). Das heißt, die Teilschlüssel können nur nacheinander Übertragen werden. Das bedeutet das man vor der Übertragung wissen muß vieviele Teilschlüssel benötigt werden um den Originalschlüssel wiederherstellen zu können. Da der KeyManager den PDA's nicht den Befehl zum Senden geben kann, da nur jeweils eine Verbindung aktiv sein kann, muß man sich für die Zeitbeschränkung der Übertragung einen anderen Modus einfallen lassen. Man könnte natürlich auch eine Zeit für die gesamte Übertragung vereinbaren nach der alle notwendigen Teilschlüssel Übertragen sein müssen. Doch es gibt das Problem das man den PDA's nicht gleichzeitig sagen kann das Sie mit der Übertragung beginnen sollen. Es ist denkbar das für jeweils eine Aktion (Senden oder Empfangen) einen zeitlich begrenzten Server benutzt. Das soll heißen, das die PDA's erst die gültige Serveradresse herausfinden müssen bevor Sie Aktionen initieren können. Dieses Verfahren wird im Kapitel 4 näher erläutert.

[3] Man könnte im verteilten Fall eine Middlewarelösung (DCOM/COM+) benutzen

[4] Dazu wäre zum Beispiel ein WLAN-Interface notwendig, um „normale“ Sockets zu benutzen.

3.2 Benutzte Hard- und Software

Es wurde folgende Hard- und Software benutzt:

- Host-System (Desktop PC): Microsoft Windows 2000 Professional (Build 2195, Servicepack 2, Deutsch)
- Hewlett-Packard Jornada 420 (Palm-size PC): Microsoft Windows CE 2.11 (Englisch)
- Compaq iPAQ H3630 (Pocket PC): Microsoft Windows CE 3.0 (Deutsch)

Alle Geräte sind mit einer IrDA-kompatiblen (IrDA 1.1) Infrarotschnittstelle ausgerüstet.

4 Realisierung

4.1 Auswahl des Verteilungsalgorithmus

Um zu schützende Daten sicher bzw. mit skalierbarer Redundanz aufzubewahren muß man diese auf verschiedene Datenträger verteilen. Es wird also prinzipell ein Algorithmus gesucht, der die Funktionalität eines RAID-Systems (*Redundant Array of Independent Disks*) hat. Das heißt die Daten werden so auf n Datenträger verteilt, daß m davon ausreichen um die Daten wiederherzustellen mit $n \leq m$. So hat man $\frac{n}{m} - 1$ -fache Redundanz der gespeicherten Daten.
Bei der Suche nach Algorithmen mit solchen Eigenschaften, traf ich hauptsächlich auf 2 Verfahren. Das sind zum einen *Shamir's Secret Sharing*[2] und zum anderen *Rabin's Information Dispersal Algorithm*[1], welche in 2.1 und 2.2 beschrieben sind. Die Implementierung von 2.1 über $\mathbb{Z}_p$ mit $p = 257$ zeigte, das dieser Zahlenbereich für 8-Bit Byte Daten ungeeigned ist, da man die Reste [0...p-1] erhält.
Man bräuchte also ein 9-Bit Byte, welches in gängigen Programmiersprachen nicht zur Verfügung steht und außerdem einen Overhead verursacht. Der Fall das das Zeichen `256` auftritt ist bei gleichwahrscheinlichen Zeichen $\frac{1}{256}$, im praktische Einsatz erfahrungsgemäß noch deutlich geringer. Das gleiche Problem tritt natürlich auch bei 2.2 über $\mathbb{Z}_p$ auf. Eine Alternative dazu ist ein*Galois Field* $GF(2^s)$[5]. Mit s=8 enthält $GF(2^8)$ genau die Zahlen [0...255]. Die Operationen über $GF(2^8)$ habe ich in 2.2 beschrieben.
Nachdem das Problem mit dem Overhead gelöst ist, stellte sich die Frage: Soll man den einfachen und einfach skalierbaren Shamir Algorithmus oder den platzoptimaleren Rabin IDA benutzen?
2.1 hat den Vorteil, daß man die Daten nicht irgendwie bearbeiten muß. Der eigentlich Schlüssel wird zeichenweise mit dem jeweiligen Index aus dem Polynom erstellt. Zum Wiederherstellen braucht man nur den Index des Teils und die entsprechenden Werte der kodierten Zeichen.
2.2 erfordert eine Vorbereitung der Daten. Zuerst müssen die Daten in Stücke zur Länge m hergestellt werden. Da zeigt sich schon das erste Problem, wenn sich die Länge der Daten nicht ohne Rest durch m teilen läßt. In diesem Fall muß man die Daten so verlängern, daß sich die Länge der Daten ohne Rest teilen lasssen. Ich füge das Zeichen 255 an die Daten an. Um nach dem Dekodieren zu wissen wieviele Füllzeichen angehängt wurden, wird vor

den Zerteilen in den ersten beiden Bytes die Anzahl der ürsprünglichen Zeichen ohne Füllzeichen gespeichert. Nach dem Wiederherstellen werden nur soviele Zeichen gelesen wie in den ersten beiden Bytes gespeichert sind. Die eventuell verbleibenden Zeichen sind dann die zuvor angehängten Füllzeichen.
Der Vorteil von IDA ist die Größe der einzelnen Teile, die kleiner sind als der Schlüssel selber. Als Nachteil sind die relativ aufwendigen Rechnungen die vorzunehmen sind (Matrixmultiplikation, Matrixinvertierung), der Speicherplatzbedarf bei sehr großen Daten (Matrixgröße) und nicht zuletzt das die Kenntnis auch nur eines Teilschlüssels einen Teil des Geheimnisses verrät. Das wird durch die Matrixoperationen verursacht, wie man sich leicht überlegen kann. Deshalb sollte man das Geheimnis vor dem Aufteilen verschlüsseln. Das wirft jedoch die Frage auf, wo der Schlüssel zum Entschlüsseln aufbewahrt werden soll.
Trotz der Nachteile des IDA habe ich mich primär für diesen entschieden, da die Schlüsselteile auf mobilen Rechnern übertragen und gespeichert werden sollen. Somit ist die Größe hier wichtiger ist als der Rechenaufwand, zumal die Rechnungen auf dem Hostrechner ausgeführt werden, wo diese nicht ins Gewicht fallen. Die Schlüsselgröße spielt außerdem eine große Rollen, wenn als Ziel andere mobile Geräte in Frage kommen, wie zum Beispiel Mobiltelefone, welche normalerweise wenig Speicher besitzen.

4.2 Kommunikation

Wie schon in 3.1 angesprochen wurde eine drahtlose Kommunikation gewählt, da diese in der Realität am wahrscheinlichsten ist und dem Gedanken des mobilen Nutzer entspricht. Bei der Verwendung der IrDA-kompatiblen (*Infrared Data Association*)(2.5)[14] Schnittstellen der PDA's stehen unter Windows CE folgende Nutzungsmöglichkeiten zu Verfügung:

(1) raw IR

(2) IrSock

(3) Ircomm

Der IrDA-Standard spezifiziert alles von der physikalischen Implementation, wie die benutzte Frequenz des Lichts, bis zum Handshaking zwischen den Geräten und wie Infrarotsysteme einander Finden und miteinander Reden. Im folgenden werden die oben genannten Kommunikationswege kurz erläutert.

4.2.1 raw IR

Bei der Benutzung von *raw IR*, folgt die Schnittstelle nicht dem IrDA-Standard, da dieser das passende Handshaking für die Verbindung fordert. Mit *raw IR* hat man die größte Kontrolle über die Verbindung, aber auch die meiste Verantwortung. Da Sender und Empfänger das gleiche Medium (die Luft) benutzen, treten Kollisionen auf, wenn man gleichzeitig von einem anderen Gerät Daten Senden und Empfangen will. Dieses Problem hat man bei einem seriellen Kabel nicht, da es dort getrennte Sende- und Empfangsleitungen gibt. Nach

[6] unterstützen wohl alle Windows CE-Getäte *raw IR*, doch das kann sich in der Zukunft jedoch ändern.

4.2.2 IrSock

Bei *IrSock* handelt sich um eine API, die auf dem IrDA-Stack aufsetzt und für IR Kommunikation benutzt wird. *IrSock* ist nur eine High-level Schnittstelle zum IrDA-Stack. Der große Unterschied zwischen den bekannten *IrSock* und *WinSock* ist, das *IrSock* keine verbindungslose Kommunikation (Datagram), keine Sicherheit unterstützt und eine vollständig andere Adressierung realisiert. *IrSock* stellt, die Möglichkeit die verfügbaren Geräte um die Infrarotschnittstelle abzufragen, Medienzugriff, Kollisionerkennung und Flusskontrolle zu Verfügung. Das einzig andere gegenüber den normalen Sockets, ist herauszufinden welche Geräte in Reichweite sind und wie deren Adressen sind. Die Adresse eines Gerätes ist eine 4-byte lange Zeichenkette. Desweiteren bindet man einem IrSocket nicht an eine Portnummer sondern an einen sogenannten Servicenamen, was in der aktuellen Version der af_irda.h [7] einer 25-byte langen Zeichenkette entspricht. Man kann also zusammenfassend sagen, der Rechnername (IP-Adresse) entspricht der 4-byte langen Device-ID und die Portnummer entspricht dem Servicenamen (ist jedocj nicht statisch). Da aber das IrDA-Protokoll serielle Kommunikation verschreibt können nicht 2 Sockets gleichzeitig Daten übertragen. Das wird jedoch durch die tieferliegenden Schichten verdeckt.

4.2.3 IrComm

Mit *IrComm* hat man schon weniger Probleme, da man mit dem Finden des Kommunikationspartners, Kollisionsvermeidung und Datenpufferung während die Kommikation mit dem Partner kurzzeitig unterbrochen ist nicht beschäftigen muß. Nachteil des Punkt-zu-Punkt Protokolls ist das nur jeweils 2 Geräte verbunden werden können.

Um Kompatiblität zu waren scheidet raw IR, da dies nicht dem IrDA-Standard folgt und somit auf verschiedenen Geräten unterschiedliche Funktionalität haben kann. Prinzipell sind alle IrDA-Kommunikationen Punkt zu Punkt Verbindungen, aber im Unterschied zu IrComm kann es bei IRSock mehere Socketverbindungen geben von denen aber nur jeweils eine Senden bzw. Empfangen kann. Ein weiterer Vorteil von IRSock ist die weitgehenden Kompatiblität zum Programmiermodell von Berkley-Sockets. Im Fall der WIN32-API wird die Integration von IrDA-Sockets mit Hilfe eine Headerdatei [7] gelöst, die die benötigten Konstanten und Strukturen enthält. Das heißt es wurden keine neuen API-Funktionen eingebaut, jediglich die Strukturen und Konstanten sind anders. Informationen über Geräte in Reichweite erhält man mit der Standardfunktion `getsockopt`, die es bei jeder Socketimplementation geben sollte. Deshalb ist es denkbar die Implementation einfach auf andere Socketanwendungen zu portieren[5].

[5] z.B. WLAN (*Wireless Local Area Network*)

4.3 Sicherheit

Transport

Da *IRSock* (4.2) in der WIN32-API keinerlei Sicherheitsfunktionen bietet, muß man die eigentliche Kommunikation anders sichern. Da naheliegendste Konzept ist die Verschlüsselung der Daten vor der Übertragung. Um an die Übertragungen Echtzeitanforderungen zu stellen, wird eine 3. Partei eingesetzt die einen zeitlich begrenzt gültigen Servicenamen für die jeweile IR-Socketverbindung vermittelt. So hat das eigentliche Sicherheitsprotokoll 2 Stufen:

1. Die erste Stufe ist ein NamingService mit konstanter Adresse, bei dem sich alle Klienten erst den aktuellen Servicenamen des gewünschten KeyManager-Sockets (Disperse oder Join) holen müssen. Dort kann überprüft werden ob sich ein Klient schon den aktuellen Servicenamen erfahren hat, denn eine eindeutige Identifizierung ist mit Hilfe der IRSockets nicht möglich, um so sicherzustellen das jedes Gerät diese Information nur einmal erhält.

2. Die zweite Stufe ist die Verschlüsselung der Kommunikation. Dies wird mit Hilfe eines reduzierten SSL-Protokolls gelöst. Beim der Verbindung eines Klienten an den Server, sendet der Server dem Klienten seinen Public-Key. Der Klient verschlüsselt damit Daten mit denen er anschließend einen symmetrischen Sitzungschlüssel erzeugt. Die verschlüsselte Information kann der Server mit seinem Private-Key entschlüsseln und seinerseits den symmetrischen Sitzungsschlüssel erzeugen. Da Klient und Server den symmetrischen Schlüssel mit den gleichen Datenmaterial erzeugt haben, können sie die Nachrichten mit diesem Schlüssel Ver- und Entschlüsseln.

Dieses Verfahren wird beim NamingService und beim eigentlichen Key-Transfer angewandt. Nun könnte man einwenden, das es mit einer *Man in the Middle* -Attacke [4] möglich ist einen oder sogar alle Schlüssel zu erlangen. Zum einen sollte es bei den Reichweiten von Infrarot nicht so einfach möglich sein unbemerkt eine Sitzung zu belauschen. Es muß dabei natürlich angenohmen werden, das eventuelle Kabelverbindungen zwischen Infrarotschnittstelle und Hostrechner ausreicht abgeschirmt sind, um nicht abgehört zu werden. Desweiteren kann beim Klienten mittels der IR-API überprüft werden wieviele Geräte in Reichweite sind. Wenn das mehr als 1 Gerät ist kann man von einem potentiellen Angriff ausgehen und erst garnicht eine Verbindung aufbauen, da man sowieso nicht weiß welches das richtige Gerät ist.
Dieses Verfahren läßt sich mit der vorhandenen Softwarebasis sowohl auf dem Hostsystem sowie auf den PDA's mittels Microsoft's Crypto-API realisieren. Der Ablauf des Protokolls ist ein Abbildung 4.2 dargestellt.
Da der asymmetrische Teil des Protokolls nicht in Funktion zu bringen war, wird dieser Schritt ausgelassen. Das Protokoll erzeugt gleich den symmetrischen Schlüssel aus beiden Seiten bekannten Daten z.B. der Device-ID des Klienten.

Authentifizierung

Bei der IR-Sockets gibt es keine Möglichkeit hat als Server die Identität des Clienten anhand des Sockets (DeviceId) zu überprüfen, da selbst die Device-ID nicht statisch ist. Sie wird zufällig vom IrLAP[14] (*Infrared Link Access Protocol*) erzeugt. Der Klient muß also

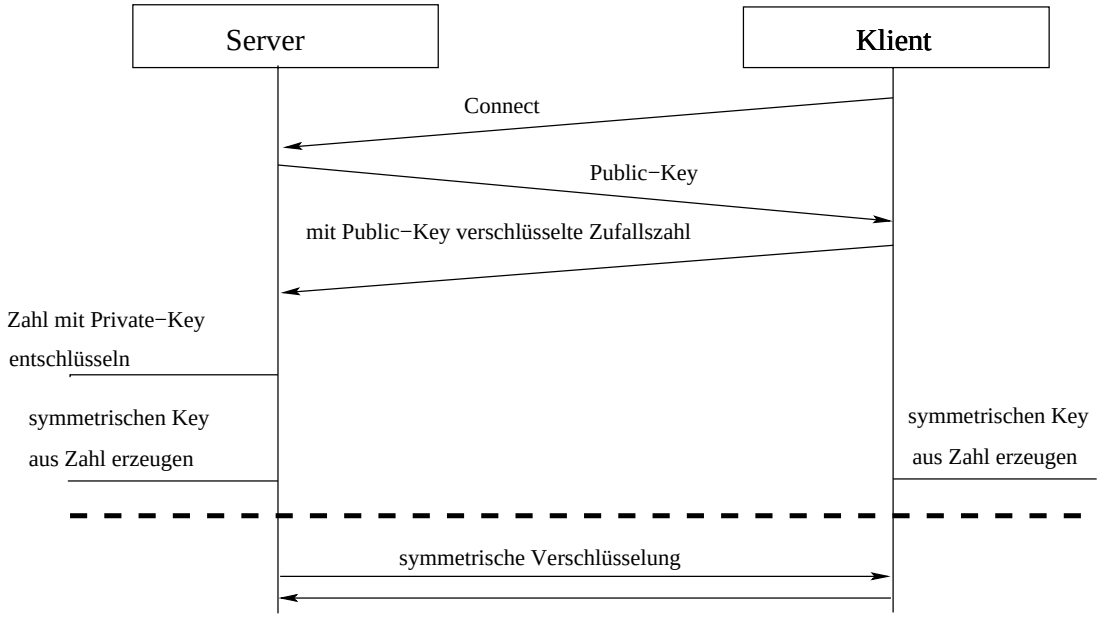

Abbildung 4.2: *Aushandeln der Verschlüsselung*

anders seine Identität beweisen. In der vorliegenden Implementation tut er dies indem er seine Identitätnummer überträgt. Der Server kann dann prüfen ob er diese Identität kennt. Diese Kennung soll eindeutig einem Gerät zugeordnet sein. Dazu würden sich Daten wie eine Seriennummer eignen. Im Rahmen dieser Arbeit wurde auf solch spezielle Feinheiten verzichten und sollen deshalb hier nur als Prinzip genannt werden.

5 Ausgewählte Implementationsdetails

Abbildung 5.1 zeigt eine schematischen Darstellung des entwickeltenten Systems, in Form von C++ Klassen. Der Klient (PDA) kommunziert mit dem KeyManager um sein Teil des zerlegten Schlssels zu erhalten und abzuliefern. Alle Parteien nutzen die Funktionen der Microsofts Crypto-API 2.0[9] in Form einer C++ Klasse die weitgehend in der jeweiligen IrSocket Klasse verborgen bleibt. Der KeyManager (NamingService,Disperse,Join) und die Klienten nutzen zur Ver- und Entschlüsselung der übertragenden Daten. Allein der Tresor ver- und entschlüsselt Dateien, welches somit den Inhalt des Tresors darstellt.Da die Klienten mobil sind und es bleiben sollen wird drahtlose Kommunikation benutzt. Alle gängigen PDA's haben heute eine IrDA-kompatible Infrarotschnittstelle. Andere drahtlosen Schnittstellen, wie z.B. WLAN (Wireless Local Area Network) muß man in der Regel extra bezahlen und erfordern zusätzlich Hardware. So lag die Verwendung der Infrarotschnittstelle mit Hilfe der WIN32-API nahe, da alle benutzten Plattformen Infrarotschnittstellen unter der WIN32-API unterstützen. Die Wahl fiehl auf die IrSock-Kommunikation, da diese mehrere Verbindungen zur gleichen Zeit unterstützt und Sockets ein wohlbekanntes Programmiermodell sind. Deshalb ist die IrSock-Programmiertechnik einfach zu verstehen. Abbildung 5.2 zeigt die IrDA-socket Struktur. Die `irdaDeviceID` Variable enthält die aktuelle Addrese des Geräted und die `irdaServiceName` Variable repräsentiert die Addresse der Socketverbindung, sowie Ports bei WinSockets. Der Wert in `irdaDeviceID` ist nicht

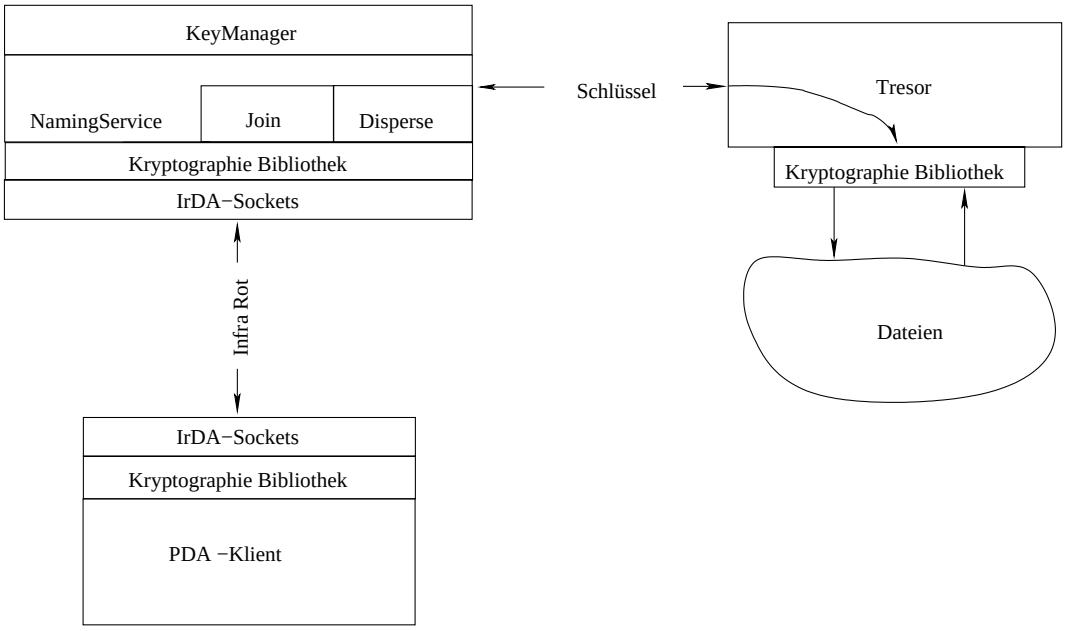

Abbildung 5.1: *Schematischer Aufbau des elektronischen Tresors*

```
typedef struct _SOCKADDR_IRDA{
  u_short irdaAddressFamily;
  u_char irdaDeviceID[4];
  char   irdaServiceName[25];
}SOCKADDR_IRDA, *PSOCKADDR_IRDA, FAR *LPSOCKADDR_IRDA;
```

Abbildung 5.2: *SOCKADDR_IRDA struct*

static, er wird dynamisch durch das IrLAP (*Infrared Link Access Protocol*)[14] erzeugt. Da IrSock nur eine aktive Verbindung erlaubt, kann für die Dispersal- und Joinoperation keine parallele Übertragung verwendet werden. Der KeyManager weiß dabei nicht wieviele Geräte kommunizieren wollen und wielange die Kommunikation dauern wird. Um an diese Information zu kommen, sind die benötigte Anzahl von Teilen zur Rekonstruktion in jedem Teil vermerkt. Abbildung 5.3 und 5.4 zeigen die Struktur der Teile nach dem Zerlegen. Es ist klar das alle Teile die gleiche Anzahl benötiger Teile und die gleiche Länge aufweisen müssen um erfolgreich zu Rekonstruieren.
Um sich vor Angreifern zu schützen, die ein Teil des Schlüssel stehlen wollen, ist der `irdaServiceName` zeitlich begrenzt gültig. Ein Klient muß sich erst mit dem NamingService-socket,welcher einen festen Servicenamen hat verbinden um den aktuellen Servicename des Dispersal- oder Join-socket zu erfahren. Zu diesem Zeitpunkt kann die Authentität

Felder:

benötigte Teile $m \leq n$	Nummer des Teils [1...n]	Daten ...

Abbildung 5.3: *Struktur eines Shamir-Teils*

Felder:	benötigte Teile $m \leq n$	*Vandermonde* Wert $[0 \ldots 255]$	Daten ...

Abbildung 5.4: *Struktur eines Rabin-Teils*

des Gerätes überprüft und die Verbindung abgebrochen werden wenn diese Gerät die Information schon erhalten hat. Denn jedes Gerät darf diese Information nur einmal erhalten. Durch Verschlüsselung der übertragenen Daten werden Angriffe durch Mithören erschwert. Man könnte auch so weit gehen und die letzen übertragenen Daten benutzen um einen neuen symmetrischen Schlüssel zu erzeugen. Dann hätte ein Angreifer nur eine Übertragungsperiode Zeit um eine Nachricht zu entschlüsseln. Als Verschlüsselungsverfahren wird ein RC4[4] *stream chipper* benutzt. Es wird ausschließlich symmetrische Verschlüsselung angewendet. Da PDA's unterschiedliche CPU-Leistungen und Betriebssystemfähigkeiten[6] haben, sollte das Verschlüsselungsverfahren nicht zu aufwendig sein. Um den gleichen symmetrischen Schlüssel zu erstellen müssen beide Parteien die gleichen Daten benutzen. Diesen Daten könnten beim Verbindungsaufbau übertragen werden, daß wäre eine unsichere Phase und ein potentieller Angriffspunkt. Deshalb werden Daten benutzt die beide Parteien ohne vorherigen Verbindungsaufbau kennen. Das ist zum Beispiel der Servicename, welche für die Disperse- und Joinsockets zeitbegrenzt sind und so für genau eine Verbindung gelten können. So kann mit Hilfe des Servicenamen für eine Verbindung genau ein symmetrischer Schlüssel existieren.
Das Kommunikationsprotokoll ist ein Request-Response Protokoll mit einigen Spezialfällen. Abbildung 5.5 zeigt das Protokoll für die Erfragung des DisperseServicenames. Zeile 1 ist

```
1  210 NamingService
2  ID 1
3  300 Accepted
4  DISPERSE
5  150 24
6  [ verschlüsselter Servicename ]
7  DONE
```

Abbildung 5.5: *DisperseServicename Request Protokoll*

die Begrüßungresponse des NamingService. Die ersten 3 Zeichen eines Responses bezeichnen den Status des vorangegangenen Request. In diesem Fall bedeutet 210: Verbindung aufgebaut und es wird symmetrische Verschlüsselung verwendet (200 bedeutet keine Verschlüsselung). In Zeile 2 schickt der Klient einen Request zur Authentifikation an den NamingService, welcher in diesem Fall verschlüsselt gesendet wird. Der Request in Zeile 4 ist eine Ausnahme vom Request-Response Protokoll, denn nach dem Response in Zeile 5 folgt gleich die angeforderte Information. Die Länge der zu empfangenen Daten wird als Parameter nach der Responsenummer 150 in Byte angegeben. Mit dem DONE Request in Zeile 7 meldet sich der Klient beim Service ab und beendet die Sitzung. Die Protokolle für das Empfangen und Senden laufen nach dem gleichen Schema ab.
Als Verteilungsalgorithmen wurden Shamir's Secret Sharing[2] und Rabin's Information Dispersal Algorithm[1] implementiert. Es ist denkbar den Algorithmus zum Wiederher-

[6] Es gibt keine einheitliche Ausgabe von Windows CE, wie bei Desktopbetriebssystemen. Jeder Hersteller kann und muß das Betriebssystem an seine Geräte anpassen.

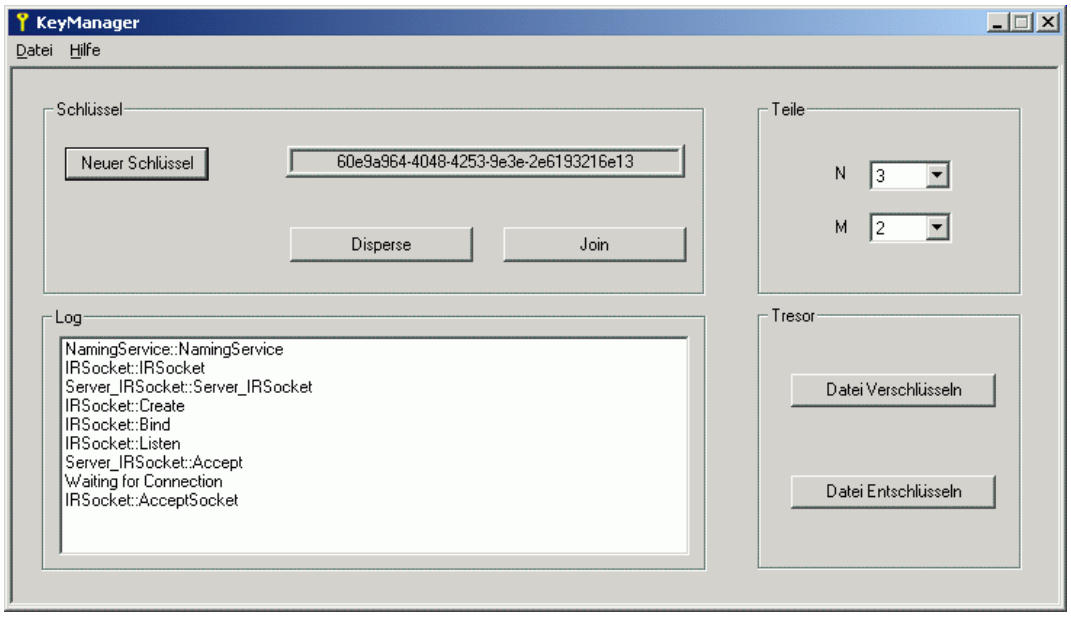

Abbildung 5.6: *KeyManager Dialog (Windows 2000)*

stellen dynamisch nach dem Empfang der Teile zu bestimmen. Dazu müsste man ein zusätzliches Feld in das Teil einbauen, welches den Algorithmustyp enthält. Es wurde in der Implementation nicht berücksichtigt, ist aber leicht änderbar. Für detailiertere Informationen zur Implementation sind wichtige Klassen der Software im Anhang zu finden. Die Abbildungen 5.6-5.9 zeigen Screenshoots der Software auf Server(Windows 2000)- und Klientseite(Windows CE).

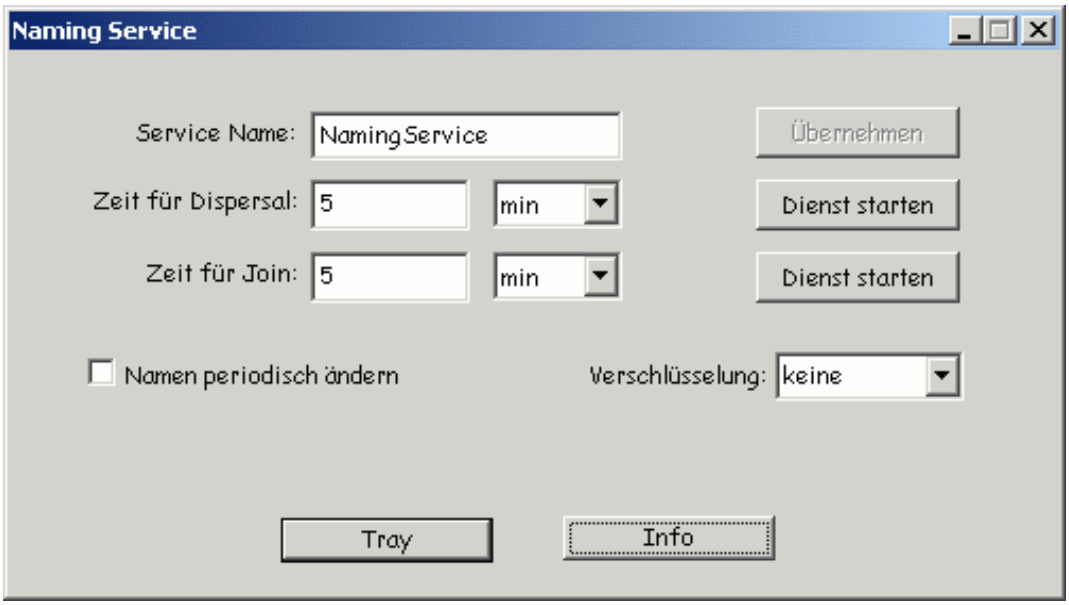

Abbildung 5.7: *NamingService Dialog (Windows 2000)*

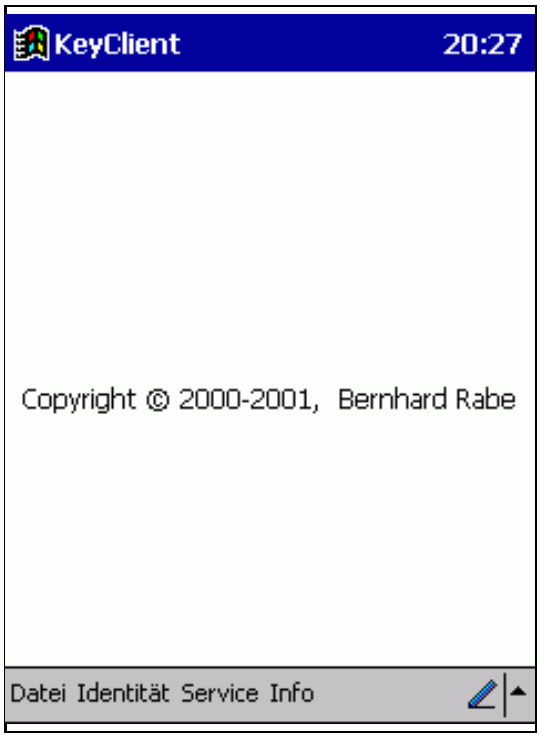

Abbildung 5.8: *KeyClient (Windows CE)*

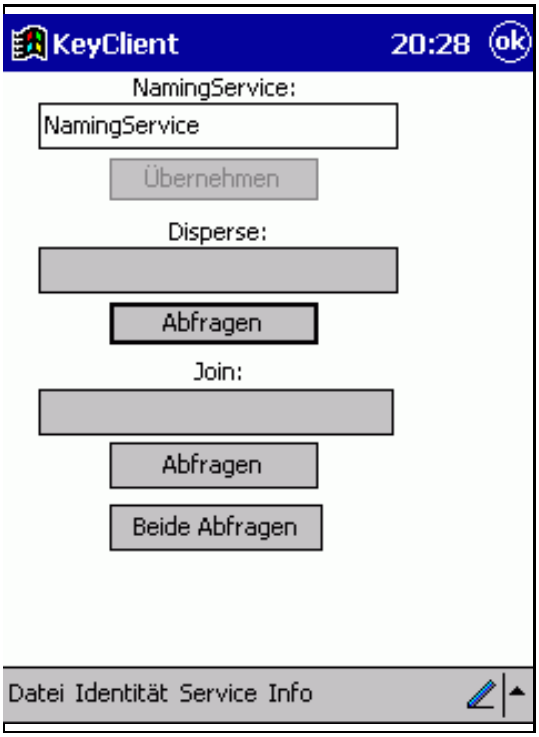

Abbildung 5.9: *Servicenamen für IrDA-Sockets (Windows CE)*

6 Ausblick/Schlußfolgerung

In dieser Arbeit wurde primär die sichere Verteilung und Aufbewahrung von Daten mittels mobiler Geräte untersucht. Dabei wurde als drahtloser Kommunikationsweg Infrarot gewählt und auf deren Eigenheiten basierend das Protokoll entwickelt. Es konnte gezeigt werden, das durch die Verteilung der Daten auf mobile Geräte die Sicherheit erhöht werden kann. Die Schwierigkeit liegt nun darin das einzelne mobile Geräte vor unbefugten Zugriff zu schützen. Doch die Sicherheit der einzelnen Geräte war nicht Ziel dieser Arbeit. Darum müssen sich die Hersteller solcher Geräte kümmern. Die Sicherheit dieses Szenarios liegt darin, das für einen Angreifer die zu stehlenden Daten nicht jederzeit sichtbar sind. Rechner in einem stationären Netzwerk sind immer verfügbar, wenn Sie nicht gerade abgeschaltet sind (dieser Fall ist aber nicht sinnvoll, da die Daten nicht zur Verfügung stehen). Es wurde gezeigt, das die Möglichkeiten der Sicherung von Infrarotkommunikation in der Programmierebene vorhanden ist, aber Einschränkungen bei der Realisierung verschiedener Szenarien gemacht werden müssen.
Bei anderen drahtlosen Kommunikationen[7] muß auch dort ein anderes Protokoll entwickelt werden. Ziel weiterführender Arbeiten kann es also sein, ein Weg zu finden das Kommunikationsprotokoll so zu verallgemeinern, das unterschiedliche drahtlose Kommunikationswege gleichzeitig benutzt werden können. Aspektorientierte Programmierung kann eine Lösung sein, indem die sichere Verteilung der Daten als Aspekt behandelt wird. Für unterschiedliche Kommunikationswege und Geräte existiert dann eine Beschreibung wie das Protokoll auf die spezielle Hardware und Kommunikation abgebildet wird.

[7] WLAN, Bluethoots, GSM...

7 Literaturverzeichnis

[1] Rabin M. O.: *Efficient dispersal of information for security, load balancing, and fault tolerance.* Journal of Association for Computing Machinery, 36(2): 335-348, April 1989

[2] Shamir A.: *How to share a secret?* Communication of the ACM, 22: 612-613, November 1979

[3] Lyuu Y.: *Information Dispersal and Parallel Computation.* Cambridge University Press: 1-22, 1992

[4] Menezes A., van Oorschot P., Vanstone S.: *Handbook of Applied Cryptography.* CRC Press `www.carc.math.uwaterloo.ca/hac`, 1996

[5] Plank James S.: *A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems.* Department of Computer Science - University of Tennessee, 19. Februar 1999

[6] Boling, D.: *Programming Microsoft Windows CE* Microsoft Press, 1998

[7] Microsoft: *af_irda.h IrDA 1.1* Windows CE SDK, 1999

[8] Hauf M.: *Automatische Generierung sicherer CORBA-Anwendungen* Humboldt-Universiät, 2000

[9] Microsoft: *Microsoft Developer Network (MSDN) 6.0a* Microsoft, 2000

[10] Bestavros A.: *AIDA-based Real-Time Fault-Tolerant Broadcast Disks* Computer Science Department Boston University, 1996

[11] Bestavros A.: *An adaptive information dispersal algorithm for time-critical reliable communication.* in Network Management and Control, Volume II. Plenum Publishing Corparation, New York, 1994

[12] Iyengar A., Cahn R., Jutla C., Garay J. A.: *Design and Implementation of a Secure Distributed Data Repository* IFIP (Chapman & Hall), 1998

[13] Garay J. A., Gennaro R., Julta C., Rabin T.: *Secure Distributed Storage and Retrieval* Proc. 11th International Workshop on Distributed Algorithms, 1997

[14] Infrared Data Association: *IrDA Serial Infrared Data Link Standard Specifications* Infrared Data Association `www.irda.org/standards/specifications.asp`, 1996

[15] Microsoft: *Windows CE Homepage* `www.microsoft.com/windowsce`

A Quelltextbeispiele

AbstractDispersal.h

```
/* AbstractDispersal.h
 *
 * Abstrakte Basisklasse fuer die Dispersalalgorithmen
 *
 * 2000-2001, Bernhard Rabe
 *
 * letzte Aenderung: 30.04.2001
*/

#ifndef __ABSTRACTDISPERSAL_H
#define __ABSTRACTDISPERSAL_H

#include "MyString.h"

class AbstractDispersal{
public:
  virtual ~AbstractDispersal() {}
  virtual int Encode(MyString,MyString*)=0;
  virtual int Decode(MyString*,MyString*,int)=0;
};
#endif
```

Rabin.h

```
#ifndef __RABIN_H
#define __RABIN_H
#include "Matrix.h"
#include "AbstractDispersal.h"

class Rabin : public AbstractDispersal
{
public:
      Rabin(int n=-1, int m=-1, int bits=8);
      int Encode(MyString message,MyString *output);
      int Decode(MyString *input,MyString *output,int z);
      int Get_n(void){return n;}
      int Get_m(void){return m;}
      ~Rabin();
protected:
      void generate_randoms(Byte *x);
      int n,m,bits;
};
#endif
```

Rabin.cpp

```
/*
 * Rabin.cpp
 *
 * Implementation des Rabin IDA Schemas ueber GF
 *
 * 2000-2001, Bernhard Rabe
 *
 * letzte Aenderung: 26.04.2001
 */
#ifdef __TEST
#pragma message("Test ist eingeschaltet in "__FILE__)
#endif

#include "Rabin.h"
#include <time.h>

////////////////////// Hilfsfunktionen
    ////////////////////////////////////
//Zufallszahlen im Bereich 0..255, keine Doppelung
    ////////////////////
void Rabin::generate_randoms(Byte *x){
      srand((unsigned) GetTickCount());
      Byte random,*belegt;

      belegt=new Byte[MAX];
      memset(belegt,0,MAX);

      for(int i=0;i<n;i++){
            do{
                  random=rand()%MAX;
            }while(belegt[random]);

            belegt[random]=1;
      x[i]=random;
      }
      delete[] belegt;
}

////////////////////// Kontruktor ////////////////////////////////

Rabin::Rabin(int n,int m,int bits): n(n), m(m), bits(bits){}

Rabin::~Rabin(){}
```

```
//////////////////////// Dispersal
    ////////////////////////////////////

int Rabin::Encode(MyString message,MyString *output){
   int lenmod,pieces,i,j,size;
   Byte b;
   Byte *x;
   Matrix A,C,encode;

   size=message.GetSize(); //Originale Groesse
   message.Insert(HIBYTE(size),0); //Highteil an den Anfang
   message.Insert(LOBYTE(size),0); //LowTeil an den Anfang

   lenmod=message.GetSize()%m; //wieviele Teile Fehlen

   pieces=(message.GetSize()+lenmod)/m; //Anzahl der Teile F1...F N/m

   encode=Matrix(m,pieces,bits);

   if(lenmod!=0){//Zeichen auffuellen
      for(i=0;i<lenmod;i++){
            message.Put(255); //Zeichen zum Auffuellen
      }
   }
   message.SetPos(0); //Auf Anfang

            /*Message in Matrix aufbauen (F1,F2 ...FN/m)*/
   for(i=0;i<encode.GetZSize();i++){
      for(j=0;j<encode.GetSSize();j++){
            if(message.GetByte(b)==-1) return -1;
                  encode.Set(i+1,j+1,b);
      }
   }

   x=new Byte[n];
   generate_randoms(x);
   A=Matrix(n,m);
   A.Vandermonde(x);

   delete[] x;

   C=A*encode;//Teile Berechnen

   for(i=0;i<n;i++){
      output[i].Put(m); //Anzahl der benoetigten Teile
      output[i].Put(A.Get(i+1,2)); //Zahl aus der Vandermonde
   }
```

```
90     for(i=0;i<C.GetZSize();i++){
          for(j=0;j<C.GetSSize();j++){
                output[i].Put(C.Get(i+1,j+1)); //W
          }
       }
95     return 0;
   }

   int Rabin::Decode(MyString *input,MyString *output,int z){
       int i,j;
100    Matrix Am,C,B;
       Byte b,*x;

       if(z!=m) return -1;

105    x=new Byte[z];

       try{
            Am=Matrix(z,z);
       }
110    catch(...){ return -99;}

       for(i=0;i<z;i++){
          input[i].SetPos(0);//Auf Anfang
          input[i].GetByte(b);
115       if(b!=z) return -2; //Falsche Anzahl angegeben

          if(input[0].GetSize()!=input[i].GetSize()) return -3; //falsche
              Laenge

          input[i].GetByte(b); //Nummer aus Vandermonde
120       x[i]=b;
       }

       Am.Vandermonde(x);

125    C=Matrix(z,input[0].GetSize()-2); //Anzahl der Teile=Zeilen Anzahl
           der Spalten=Laenge -2

       for(i=0;i<input[0].GetSize()-2;i++){ //alle Zeichen im String
                for(j=0;j<z;j++){ //Alle Strings
                      input[j].GetByte(b);
130                   C.Set(j+1,i+1,b);
                }
       }
```

```
   Am.GaussJordan(); //invertieren
   B=Am*C;      //Werten

   int size=MAKEWORD(B.Get(1,1),B.Get(1,2)); //Anzahl der Zeichen
   int count=0;
   i=1;//1. Zeile
   j=3;//3. Spalte
   while(count != size){
        b=B.Get(i,j);
        output->Put(b);
        count++;
        if(i!=B.GetZSize() && j==B.GetSSize())i++;

        if(j!=B.GetSSize())
              j++;
        else
              j=1;
   }
   delete[] x;
   return 0;
}
```

Shamir.h

```
#ifndef __SHAMIR_H
#define __SHAMIR_H

#include "AbstractDispersal.h"
#include "Gf.h"

class Shamir : public AbstractDispersal
{
public:
      Shamir(int k,int n,int p=8); //(k,n) threshold sheme
      ~Shamir();
      int Encode(MyString in,MyString *out);
      int Decode(MyString *in,MyString *out,int z);
private:
      void generate_ai(Byte *ai);
      int k,n,powerof2,prime;
      Gf *gf;
};
#endif
```

Shamir.cpp

```
/*
 * Shamir.cpp
```

```
 *
 * Implementation von Shamir's Secret Sharing ueber GF
 *
 * 2000-2001, Bernhard Rabe
 *
 * letzte Aenderung: 27.06.2001
 *
 */
#include <iostream>
#include <time.h>
#include <cmath>
#include "Shamir.h"

void Shamir::generate_ai(Byte *ai){
      srand((unsigned) GetTickCount() );
      for(int i=0;i<k-1;i++)
            ai[i]=rand()%255;
}

Shamir::Shamir(int k,int n,int p) : k(k), n(n), powerof2(p){
  prime = 1<<powerof2;
  gf=new Gf();
  srand((unsigned) GetTickCount());//Zufallsgenerator starten
}

Shamir::~Shamir(){
  delete gf;
}


/*Aufteilen ueber GF(2^powerof2)*/
int Shamir::Encode(MyString in,MyString *out){
      int i,j;
      byte message,prod;
      Byte *ai;   /*Vorzeichenlos da in GF 0...(2^x)-1 */

      ai=new Byte[k-1];

      for(i=0;i<n;i++){
            out[i].Put(k); //Anzahl der benoetigtigten Teile
            out[i].Put(i+1); //Nummer des Teils
      }

   while(in.GetByte(message)!=-1){
      generate_ai(ai);
            for(j=0;j<n;j++){
                  prod=message; //Byte
```

```
50                  for(i=1;i<k;i++){
                          prod=gf->Add(prod,gf->Mul(ai[i-1], gf->Pow(j
                              +1,i)));//zufaellig 0<a<257 ai*x^i
                    }
                    out[j].Put(prod); //Wert
              }
55        }
          delete[] ai;
          return 0;
    }

60  /*Ueber GF(2^powerof2) decodieren*/
    int Shamir::Decode(MyString *in,MyString *out,int z){
          byte b[100];
          int i,j;
          byte zaehler=0,nenner=0,D0=0;
65        byte c;

          /*Funktioniert nur mit z>=2*/
          for(i=0;i<z;i++){
          in[i].GetByte(c);
70              if(c>z)/*Falsche Anzahl benoetigter Schluessel*/
                    return -1;

          if(in[0].GetSize()!=in[i].GetSize())/*unterschiedliche öGre*/
                    return -2;
75        in[i].GetByte(c);
          b[i]=c; //Nummer des Teils
          }

          while(in[0].GetPos()<in[0].GetSize()){
80              for(j=0;j<z;j++){// xj
                    zaehler=0;
                    nenner=0;
                    for(i=0;i<z;i++){
                          if(j!=i){
85                              if(zaehler==0)
                                      zaehler=b[i];
                                else
                                      zaehler=gf->Mul(zaehler,b[i]); //(
                                          x-xi) i!=j x=0
                          }
90                        if(j!=i){
                                if(nenner==0)
                                      nenner=gf->Sub(b[j],b[i]); //(xj-
                                          xi)
                                else
```

```
                                        nenner=gf->Mul(nenner,gf->Sub(b[j
                                            ],b[i]));
95
                                }
                        }
                        in[j].GetByte(c);
                        D0 = gf->Add(D0,gf->Mul(gf->Div(zaehler,nenner),c)
                            );
100             }
                out->Put(D0);
                D0=0;
        }
        return 0;
105 }
```