Facharbeit Informatik: Kryptographische Verfahren

Arne Bochem

10. August 2003

Inhaltsverzeichnis

1 Einführung

„There are two kinds of cryptography in this world: cryptography that will stop your kid sister from reading your files, and cryptography that will stop major governments from reading your files.“ schreibt Bruce Schneier im Vorwort seines Buches „Applied Cryptography“[1]. Was aber ist Kryptographie eigentlich? Auf diese und andere Fragen wird sich im Folgenden eine Antwort finden.

1.1 Was ist Kryptographie?

Kryptographie ist die Kunst des Kodierens von Daten auf eine Art, welche es nur bestimmten Personen ermöglicht, sie wieder zu dekodieren. Gewöhnlich geschieht dies, indem unter Verwendung eines Algorithmus der Klartext und Schlüssel auf eine bestimmte Weise kombiniert werden, um den Chiffretext zu erzeugen. Dieser Chiffretext sollte nur von Personen wieder zurück in den Klartext verwandelt werden können, welche sich im Besitz des Schlüssels befinden.[2]

1.2 Rahmen dieses Textes

Dieser Text wird einen Überblick über die verschiedenen Typen von Verfahren geben, welche in der Kryptographie verwendet werden, als da wären Substitutionschiffre, Stromchiffre, Blockchiffre, symmetrische Verfahren, asymmetrische Verfahren und Quantenkryptographie. Außerdem wird eine Beschreibung der zwei Algorithmen RC4 (als CipherSaber2) und Blowfish (im ECB/Electronic Codebook Modus) geliefert und die Performance einer Implementation der beiden Verfahren verglichen werden.

Zu Quellcode-Teilen, welche in diesem Text auftauchen, ist zu sagen, dass die Kommentare aus Platzgründen entfernt wurden, jedoch im Quellcode auf der CD zu finden sind. Auch wurden manche Formatierungen verändert.

2 Verschiedene Verfahren

In diesem Teil des Textes werden verschiedene Arten von Chiffren, die Bedeutung von symmetrischen und asymmetrischen Verfahren sowie die Quantenkryptographie näher betrachtet.

[1] [AC] S. xix
[2] Nach [FOD]

2.1 Substitutionschiffre

Diese Art der Verschlüsselung ist wohl die gemeinhin bekannteste und simpelste. Beim Einsatz eines Substitutionschiffre werden ganz einfach bestimmte Buchstaben durch ein oder mehrere Symbole bzw. Buchstaben ersetzt, wodurch die Nachricht unleserlich gemacht wird.

Hier muss als Beispiel mindestens das einfache Rotationsverfahren (Alphabetischer Cäsar)[3] genannt werden. Hierbei werden die Buchstaben des Alphabets einfach um eine bestimmte Anzahl weitergedreht. Wählt man bei diesem Verfahren als Schlüssel eine Rotation von 1, so wird z.B. „a“ zu „b“, „b“ zu „c“, etc. und „z“ zu „a“, aus dem Wort „Kryptographie“ würde also „Lszquphsbqijf“. Da es hier nur 26 verschiedene Schlüssel gibt (für Bytes 256), ist dieses Verfahren extrem unsicher. Verwendet wird es zum Beispiel in Form von Rot13 (Rotation um 13 Buchstaben), bei welchem Ver- und Entschlüsselung exakt gleich verlaufen, um potentiell beleidigende Aussagen nur für Personen sichtbar zu machen, welche auch bereit sind, diese zu lesen.

2.2 Blockchiffre

Bei der Blockchiffrierung (Blockcipher)[4] von Daten, wird der zu verschlüsselnde Klartext in Blöcken behandelt. Oft wird als Blockgröße 64 Bit ausgewählt. Der Algorithmus arbeitet dann mit dem gesamten Blockinhalt gleichzeitig. Um diese Blöcke nun zusammenzufügen, gibt es verschiedene Möglichkeiten.

Am offensichtlichsten ist hier der ECB - Electronic Codebook Modus - in welchem die Blöcke ganz einfach nacheinander verschlüsselt und „aneinander geklebt“ werden. Dieser Modus hat den Nachteil, dass Muster im Klartext nicht verborgen werden. Außerdem kann hierbei der Klartext durch Änderungen am Chiffretext einfach manipuliert werden. Die einzigen Vorteile sind hierbei Geschwindigkeit und Simplizität.

Eine sicherere Möglichkeit ist der CBC, der Cipher Block Chaining Modus. Hierbei wird der Klartext, bevor er verschlüsselt wird, mit dem letzten Block geXORt[5]. Dies verhindert ein Erkennen von Mustern im Klartext. Um den ersten Block zu verschlüsseln, wird ein Initialisierungsvektor (IV) benötigt, um den ersten Block damit zu XOREn. Dieser IV ist nicht geheim und kann einfach im Klartext mit der Nachricht mitgeschickt werden. Als IV geeignet sind z.B. Zufallsdaten oder ein Datum. Ein IV hat einen weiteren Vor-

[3] [CAE]

[4] [AC] S. 189, folgende

[5] XOR: Exklusiv-Oder. Wert1 XOR Wert2 ist wahr, wenn entweder Wert1 oder Wert2 wahr ist, nicht aber beide. Beispiele: 1 xor 0 = 1; 1 xor 1 = 0; 0 xor 1 = 1; 0 xor 0 = 0; 11010010 xor 01101010 = 10111000

teil. Auch wenn zweimal dieselbe Nachricht verschlüsselt wird, wird sie nicht denselben Chiffretext erzeugen, da der IV verschieden ist. Die Entschlüsselung verläuft wie die Verschlüsselung, nur in umgekehrter Reihenfolge.

Falls man kleinere Blöcke, als der Blockchiffre zulässt, verschlüsseln möchte, bietet sich der CFB/Cipher Feedback Modus an. Hier wird wie beim CBC ein IV von der Größe der Blöcke des Algorithmus benutzt. Dieser wird verschlüsselt und dann werden die linken n (n ist die Anzahl der Bits, welche auf einmal verschlüsselt werden sollen) Bits des verschlüsselten IV mit den n Bits des Klartextes geXORt, welche nun verschickt werden können. Im IV werden nun alle Bits um n Stellen nach links verschoben und die nun freien n Bits rechts mit denen des Chiffretextes gefüllt. Ist n gleich der Anzahl der Bits, welche einen Block des Chiffres ausmachen, so funktioniert der CFB Modus praktisch genauso wie der CBC Modus. Beim CFB muss allerdings der IV einzigartig sein, welches beim CBC zwar auch gut ist, jedoch nicht für die Sicherheit essentiell.

Es gibt noch einige weitere Möglichkeiten, zum Beispiel:

- OFB: Output Feedback Modus
- Counter (Zähler) Modus
- BC: Block Chaining Modus
- PCBC: Propagating Cipher Block Chaining Modus
- CBCC: Cipher Block Chaining with Checksum Modus und einige andere.

Zu den Blockchiffren gehören zum Beispiel die Algorithmen:

- IDEA
- RC2
- Blowfish
- RC5
- LOKI

2.3 Stromchiffre

Im Gegensatz zum Blockchiffre wird beim Stromchiffre (Streamcipher) nicht mit einem ganzen Block an Daten gearbeitet, sondern der Klartext wird (für gewöhnlich) Bit für Bit oder Byte für Byte abgearbeitet.[6] In den meisten Fällen wird hier mit einem Schlüsselstromgenerator (als Pseudozufallszahlengenerator) ein Strom von Bits (oder Bytes) erzeugt, die mit denen des Klartextes geXORt oder modulo[7] den Maximalwert addiert werden. Hierbei hängt die Sicherheit des Chiffres davon ab, wie „zufällig“ die Ausgabe des Schlüsselstromgenerators ist. Eine geringe Zufälligkeit schwächt das Verfahren und führt dazu, dass es leichter ist, unbefugt eine Nachricht zu entschlüsseln. Damit nicht jede Person, die eine Verschlüsselung mit einem Stromchiffre durchführen möchte, einen eigenen Algorithmus benötigt, was ja relativ schlecht durchführbar wäre, werden Schlüssel verwendet, welche die Ausgabe des Schlüsselstromgenerators bestimmen.

Zu den Stromchiffren gehören unter anderem die Algorithmen:

- RC4
- SEAL
- PKZIP (welcher der ziemlich schwache Verschlüsselungsalgorithmus ist, mit dem Zip-Dateien mit Passwörtern geschützt werden.)

2.4 Symmetrische Verfahren

Ob ein Verfahren symmetrisch ist, sagt uns die Tatsache, ob Verschlüsselung und Entschlüsselung mit denselben Schlüsseln geschehen. Ist dies der Fall, ist das Verfahren symmetrisch. Allgemein lässt sich sagen, dass für diese Verfahren in den meisten Fällen kürzere Schlüssel verwendet werden als bei asymmetrischen Verfahren. Die bereits oben genannten Algorithmen gehören zu dieser Kategorie.

2.5 Asymmetrische Verfahren

Ein asymmetrisches Verfahren benutzt zum Ver- und Entschlüsseln verschiedene Schlüssel. Dies macht es einerseits einfacher, einen Schlüsselaustausch mit jemandem durchzuführen, da man nur den Schlüssel zum Verschlüsseln weitergibt, welchen in den meisten Fällen wohl jeder haben darf. Andererseits

[6] [AC] S. 197

[7] Modulo: Wert1 modulo Wert2 ist gleich dem Rest von Wert1 div Wert2. Beispiele: 2 mod 10 = 2; 13 mod 10 = 3; 10 mod 10 = 0.

werden hier jedoch bei vielen Verfahren längere Schlüssel benötigt, um eine vergleichbare Sicherheit wie ein symmetrisches Verfahren mit einem kürzeren Schlüssel zu erreichen, da bei asymmetrischen Verfahren oft Wege vorhanden sind, einen Schlüssel anhand des anderen zu berechnen. Je nach Schlüssellänge und Verfahren dauert diese Berechnung des einen Schlüssels aus dem anderen so lange, dass diese praktisch nicht durchführbar ist.

Solche Algorithmen sind zum Beispiel:

- RSA (nach dessen Erfindern Ron Rivest, Adi Shamir und Leonard Adleman benannt), welcher seine Sicherheit aus dem Problem der Primfaktorisierung großer Zahlen bezieht.
- ElGamal, dessen Basis die Schwierigkeit der Berechnung diskreter Logarithmen innerhalb eines endlichen Feldes ist.

Mit in diesen Bereich gehören auch die digitalen Signaturalgorithmen, die zumeist auf asymmetrischen Public-Key (Öffentlicher Schlüssel) Algorithmen basieren bzw. Teil von ihnen sind. Als Beispiel sei hier DSA (Digital Signature Algorithm) genannt, welcher sich sowohl mit ElGamal als auch RSA verwenden lässt.

2.6 Quantenkryptographie

Durch die Ausnutzung physikalischer Gesetze wird es selbst mit unbegrenzter Rechenkraft unmöglich, etwas mit Quantenkryptographie[8] Verschlüsseltes zu knacken, ohne dabei die Daten (des Schlüssels) zu verändern. Der Quantenmechanik zufolge kann man nicht mehrere Eigenschaften eines Partikels gleichzeitig messen. Wenn eine Eigenschaft gemessen wird, folgt daraus automatisch, dass es keine Möglichkeit gibt, um eine andere Eigenschaft zu messen. Damit besteht stets eine gewisse Unsicherheit.

Diese kann ausgenutzt werden, um einen kryptographischen Schlüssel oder sogar einen Einwegblock (One-Time-Pad) zu erzeugen. Eine gute Möglichkeit dazu ergibt sich bei polarisierten Photonen. Angenommen, sie sind in einem bestimmten Winkel polarisiert, werden alle einen Polarisationsfilter mit gleicher Ausrichtung durchdringen. Je stärker die Ausrichtungen von Polarisationsfilter und polarisierten Photonen differieren, desto weniger Photonen durchdringen den Filter. Bei einer Differenz von 90° wird so zum Beispiel kein einziges Photon den Filter durchdringen, während bei einer 45° Differenz immer noch eine 50%ige Chance besteht, dass ein Photon den Filter durchdringt.

[8] [AC] S. 554

Gesetzt den Fall, wir haben zwei Messbasen, eine horizontal-und-vertikal, d.h. sie lässt 0° und 90° (180°, 270°) durch. Die zweite ist eine diagonale Messbasis, welche 45° und 135° (225°, 315°) durchlässt, so haben wir jeweils zwei Möglichkeiten, ein Photon zu messen. Je nach Ausrichtung des Photons liefert beim Messen nun die eine Basis ein zufälliges Ergebnis, während die andere ein korrektes liefert und ein Messen mit beiden nicht möglich ist.

Nun ein Beispiel zur Anwendung: Alice und Bob wollen über einen sicheren Kanal kommunizieren. Hierzu schickt nun Alice eine Reihe von Photonenimpulsen, wobei jeder Impuls zufällig in einem der vier Winkel (0°, 90°, 45°, 135°) ausgerichtet ist. Bob wählt nun zufällig die Messbasen für einen Polarisationsdetektor aus. Nach der Messung teilt Bob nun über einen unsicheren Kanal mit, welche Einstellungen er verwendet hat, woraufhin sie ihm sagt, welche davon korrekt waren. Die Messungen mit nicht richtigen Einstellungen werden verworfen und die anderen nach einem Vereinbarten System in Bits umgewandelt. Auf diese Weise können so viele Bits generiert werden wie gewünscht. Bob wird durchschnittlich in 50% der Fälle die richtige Einstellung wählen, das heißt Alice muss ca. $2n$ (wobei n die Anzahl der gewünschten Bits ist) Photonenimpulse abschicken.

Im Falle, dass jemand versucht, die Photonenimpulse abzuhören, wird das eine Änderung der Bits, welche Alice und Bob messen, zur Folge haben. Um festzustellen, ob jemand versucht sie abzuhören, können sie einfach einige der gemessenen Bits vergleichen. Falls es Unterschiede gibt, wissen sie, dass sie abgehört werden. Falls es keine gibt, werden die verglichenen Bits einfach verworfen und die übrigen verwendet.

3 Auswahl zweier Algorithmen

Im folgenden wird die Performance zweier kryptographischer Algorithmen verglichen, nämlich RC4 (in Form von CipherSaber2[9]) als Stromchiffre und Blowfish (im ECB) als Blockchiffre. Aufgrund der teilweise ziemlich radikalen Unterschiede zwischen den einzelnen Algorithmen der beiden Gattungen kann ein solcher Vergleich natürlich nicht repräsentativ für Stromchiffren und Blockchiffren sein. Auch kann es durch Eigenheiten bei der Umsetzung der Algorithmen zu Verzerrungen des Ergebnisses kommen. Dies soll jedoch nicht bedeuten, dass der Vergleich völlig ohne Aussagekraft ist. Insbesondere bei großen Unterschieden lässt sich vermutlich doch etwas über die Geschwindigkeit der beiden Algorithmen aussagen.

[9] [CS]

3.1 Implementation von RC4 (CipherSaber2)

Der RC4 Algorithmus ist ein Stromchiffre, welcher 1987 von Ron Rivest für RSA Data Security Inc. entwickelt wurde. Als CipherSaber2 implementiert, besteht der Schlüssel aus einem 10 Bytes langen Initialisierungsvektor und bis zu 246 weiteren Bytes. Da mit einigen Millionen abgefangenen RC4-Nachrichten - bei schwachen Schlüsseln reichen ca. 20.000 aus, welche denselben Schlüssel verwenden, eine Möglichkeit besteht, den Schlüssel zu brechen, wird in CS2 die StateArray-Misch-Schleife n Mal wiederholt, wobei n eine Zahl ist, auf die sich Absender und Empfänger geeinigt haben. Dies sollte das Problem zu einem gewissen Grad beheben. Mit $n = 1$ ist CS2 dasselbe wie CipherSaber1.

Eine CipherSaber Datei beginnt mit dem 10 Bytes langen Initialisierungsvektor, der unverschlüsselt, also im Klartext, gespeichert wird, da er nicht geheim ist. Darauf folgen die verschlüsselten Daten. Um eine optimale Vermischung von IV und Schlüssel zu garantieren, sollte die Schlüssellänge nicht über 54 Bytes steigen. Die gilt allerdings nur für CipherSaber1, für CipherSaber2 mit $n \geq 2$ darf der Schlüssel die möglichen 246 Bytes lang sein.

In RC4 gibt es ein StateArray (256 Bytes lang), das den Zustand des Pseudozufallszahlengenerators speichert. Dieses wird anhand des KeyArrays (bis zu 256 Bytes lang), welches dem Schlüssel-String entspricht, gemischt. Das DataArray ist einfach ein Array, in dem die Bytes der zu verschlüsselnden Daten enthalten sind.

Die erste Phase von RC4 besteht darin, das StateArray vorzubereiten. Zunächst wird hier das StateArray mit den Werten von 0 bis 255 gefüllt, so dass das 0-te Element 0 ist, das 1-te Element 1 und so weiter. Danach wird gemischt. Hierzu wird mit einer Schleife der Zähler i von 0 bis 255 hochgezählt. Bei jedem dieser Schritte wird zu der Variable j (Startwert: 0) der Inhalt des i-ten Elementes des StateArray und des n-ten Elementes des KeyArray addiert (alle Additionen in diesem Algorithmus werden modulo 256 durchgeführt), wobei n gleich i modulo die Schlüssellänge ist. Als nächstes werden das i-te und j-te Element des StateArray vertauscht. Für CipherSaber2 wird diese Schleife n Mal wiederholt.

Die Ver- oder Entschlüsselung an sich findet in der zweiten Phase von RC4 statt. Zu Beginn dieser Phase werden wieder 2 Variablen, i und j, mit 0 initialisiert. Nun wird für jedes Byte der zu verschlüsselnden Daten i um eins erhöht und der Inhalt des i-ten Elementes des StateArray zu j addiert. Das i-te und j-te Element des StateArray werden vertauscht und zu einem neuen Wert „n“ addiert. Das n-te Element des StateArray wird mit dem aktuellen Daten-Byte geXORt. Das geXORte Byte kann nun geschrieben werden. Es ist verschlüsselt.

In dieser Umsetzung wird eine Klasse TCipherSaber implementiert. Die für das benutzende Programm wichtigsten Funktionen sind der Constructor ApplyCS2, welcher als Argumente den Schlüssel, die Daten und einen booleschen Wert annimmt, der bestimmt, ob ver- oder entschlüsselt werden soll. Auch das StateArray wird im Constructor initialisiert. Auch wichtig ist die Prozedur ApplyCipher, die die echte Ver- oder Entschlüsselung durchführt. Die Funktion GetData, liefert nun die ver- oder entschlüsselten Daten als String zurück.

Im Constructor geschieht nun, nachdem die Schlüssellänge festgestellt wurde und in einer booleschen Variable gespeichert wurde, ob ver- oder entschlüsselt wird, folgendes:

```
for i := 0 to 245 do
  if i < (keylength - 10) then
    begin
      KeyArray[i] := ord(Key[i + 1]);
      Key[i + 1] := chr(0);
    end;
```

Hier werden die Bytes des Schlüssels in das KeyArray übertragen. Als nächstes wird bei Verschlüsselung ein Initialisierungsvektor aus Zufallszahlen an den Schlüssel gehängt und zusätzlich in einem eigenen Array gespeichert:

```
SetLength(InitVector, 10);
j := 0;
for i := (keylength - 10) to keylength do
  begin
    KeyArray[i] := random(256);
    InitVector[j] := KeyArray[i];
    j := j + 1;
  end;
```

Falls entschlüsselt wird, wird der IV nicht in einem eigenen Array gespeichert, sondern nur aus dem Data-String in das KeyArray kopiert und danach aus dem Data-String gelöscht.

```
for i := (keylength - 10) to keylength - 1 do
  begin
    KeyArray[i] := ord(Data[i - (keylength - 11)]);
    Data[i - (keylength - 11)] := chr(0);
  end;
Delete(Data, 1, 10);
```

Im letzten dokumentierenswerten Teil des Constructors wird der Data-String zum DataArray (TByteArray) umgewandelt:

```
for i:=0 to (datlength-1) do
  DataArray[i]:=ord(Data[i+1]);
```

In der Prozedur SetupStateArray wird zunächst, wie in der Definition von RC4 beschrieben, das StateArray mit den Werten 0-255 gefüllt. Nachdem dies erledigt ist, wird das Array „gemischt“. Die äußere for-Schleife ist Teil von CipherSaber2:

```
j := 0;
for k := 1 to cs2_n do
  for i := 0 to 255 do
    begin
      n := i mod keylength;
      j := (j + StateArray[i] + KeyArray[n]) mod 256;
      SwapElements(StateArray, i, j);
    end;
```

Die (De-)Chiffrierung selbst, aus ApplyCipher, ist ziemlich simpel. Das StateArray wird hierbei weiter gemischt und dann ein Element des DataArray mit einem des StateArrays geXORt:

```
i := 0;
j := 0;
for c := 0 to (datlength - 1) do begin
    i := (i + 1) mod 256;
    j := (j + StateArray[i]) mod 256;
    SwapElements(StateArray, i, j);
    n := (StateArray[i] + StateArray[j]) mod 256;
    DataArray[c] := StateArray[n] xor DataArray[c];
  end;
```

SwapElements ist ein simpler Ringtausch (Wert1 → Temp, Wert2 → Wert1, Temp → Wert2) und eigentlich keine genauere Erklärung wert. Damit sind wir auch schon bei der letzten Funktion der RC4 Implementation, nämlich GetData. Hier wird das DataArray wieder zum String umgebaut. Im Falle einer Verschlüsselung wird noch der Inhalt des Arrays, in welchem der Initialisierungsvektor gespeichert wurde, vorne an den String gehängt. Gleichzeitig werden die Elemente der beiden Arrays auf Null gesetzt. Im folgenden Stück wird der IV vorne an die Ausgabe gehängt.

```
if ciphermode then for i := 0 to 9 do
  begin
    datastring := datastring + chr(InitVector[i]);
    InitVector[i] := 0;
  end;
```

Mit diesem Code wird das DataArray wieder in einen String umgewandelt und die einzelnen Elemente auf Null gesetzt:

```
for i := 0 to (datlength - 1) do
  begin
    datastring := datastring + chr(DataArray[i]);
    DataArray[i] := 0;
  end;
```

3.2 Implementation von Blowfish (ECB)

Der Blockchiffre Algorithmus Blowfish wurde 1993 von Bruce Schneier[10] entwickelt. Gegen Blowfish selbst gibt es keine erfolgreichen Methoden, welche für eine Implementation mit 16 Runden schneller sind als Brute Force[11]. Es gibt zwar schwache Schlüssel, jedoch wurde bisher noch keine Möglichkeit entdeckt, diese gegen ein 16 Runden Blowfish auszunutzen. Ein Problem, was die Sicherheit anbelangt, ist hier jedoch der aus Performancegründen (für den Geschwindigkeitsvergleich) gewählte ECB Modus. Dieser ermöglicht es theoretisch, die Daten relativ einfach zu modifizieren oder ein Muster zu entdecken (siehe 2.2).

Erst einmal soll nun der Algorithmus an sich beschrieben werden. Blowfish ist ein 64 Bit Blockchiffre mit einer Schlüssellänge von bis zu 448 Bits. Dieser Schlüssel wird durch eine Schlüsselexpansionsroutine zu mehreren Subkeyarrays mit einer Gesamtgröße von 4168 Bytes konvertiert. Diese Subkeys müssen berechnet werden, bevor eine Ver- oder Entschlüsselung stattfinden kann.

Da Blowfish ein Feistel Network[12] ist, verlaufen Ver- und Entschlüsselung gleich, mit dem Unterschied, dass bei der Entschlüsselung P_1, P_2, …, P_{18}

[10] [CP] /blowfish.html

[11] Brute Force: Das Ausprobieren aller möglichen Schlüssel oder Passworte.

[12] Feistel Network: Ein aus den frühen der 70er Jahren des 20. Jhdts stammendes Prinzip, bei dem ein Block der Länge n in zwei Hälften, L und R, von der Länge $n/2$ gespalten wird, wobei n eine gerade Zahl sein muss. Die Ausgabe der i-ten Runde wird durch die Ausgabe der vorherigen Runde bestimmt. Da XOR benutzt wird um die linke und rechte Hälfte in den Runden zu kombinieren, ist garantiert, dass der Vorgang umkehrbar ist. Wenn K_i der Subkey in der i-ten Runde ist und f eine frei bestimmbare Funktion für diese Runde ist, gilt automatisch

in der umgekehrten Reihenfolge verwendet werden. P_1 bis P_{18} sind die 32 Bit Subkeys und Elemente des P-Array. Außer dem P-Array gibt es noch 4 S-Boxen, welche jeweils 256 Elemente von je 32 Bits Länge enthalten. Um diese Subkeys zu berechnen wird folgendes Verfahren angewendet:

Zunächst werden das P-Array und die vier S-Boxen - in dieser Reihenfolge - mit einem konstanten String initialisiert, welcher aus den hexadezimalen Stellen von π besteht. Nun wird P_1 mit den ersten 32 Bits des Schlüssels geXORt, P_2 mit den zweiten 32 Bits des Schlüssels und so weiter, bis alle Elemente des P-Array mit Schlüssel-Bits geXORt werden. Wenn keine neuen Schlüssel-Bits mehr zu finden sind, werden wieder die ersten Bits verwendet.

Nachdem dieser Schritt getan ist, wird ein 64 Bit langer Block aus Bytes mit dem Wert Null verschlüsselt. Die linke Hälfte der 64 Bit wird P_1, die rechte Hälfte wird P_2 zugewiesen. Sodann wird der verschlüsselte Wert erneut verschlüsselt. Diesmal wird die linke Hälfte P_3 und die rechte Hälfte P_4 zugewiesen. Auf diese Weise wird auch mit den anderen Elementen des P-Array verfahren und danach mit den vier S-Boxen, in der Reihenfolge.

Im Ver- und Entschlüsselungsteil wird eine Funktion $F(L)$ verwendet. Hier wird zunächst der 32 Bit Wert L in 4 Teile - a, b, c, d, jeder 8 Bits - aufgeteilt. Definiert ist die Funktion als ($S1$ ist die erste S-Box, $S2$ die zweite S-Box und so weiter):

$$((((S_{1,a} + S_{2,b}) \text{ modulo } 2^{32}) \text{ xor } S_{3,c}) + S_{4,d}) \text{ modulo } 2^{32}$$

Eine Verschlüsselung mit dem Blowfish Algorithmus verläuft folgendermaßen: Zunächst wird ein 64 Bit Block in zwei 32 Bit Hälften aufgeteilt. Nun wird ein Zähler „i“ von 1 bis 16 hochgezählt. Bei jedem Hochzählen wird die linke Hälfte mit dem i-ten Element des P-Array geXORt und die rechte Hälfte mit dem Ergebnis einer Funktion $F(L)$ geXORt, wobei L die linke Hälfte ist. Außerdem werden die linke und rechte Hälfte vertauscht. Nach dem Durchlaufen der Schleife werden die linke und rechte Hälfte erneut vertauscht, um den letzten Tausch rückgängig zu machen. Die rechte Hälfte wird nun mit P_{17} geXORt und die linke Hälfte mit P_{18}. Nun können die beiden Hälften wieder zu einem 64 Bit Block zusammengefügt werden. Damit ist die Verschlüsselung abgeschlossen. Zum Entschlüsseln wird das ganze Verfahren genauso verwendet, nur werden $P_1, P_2, \ldots, P_{18}$ in der umgekehrten Reihenfolge verwendet.

$L_i = R_{i-1} \wedge R_i = L_{i-1}\ XOR\ f(R_{i-1}, K_i)$
$\Rightarrow L_{i-1}\ XOR\ f(R_{i-1}, K_i)\ XOR\ f(R_{i-1}, K_i) = L_{i-1}$

solange der Input für f in jeder Runde rekonstruiert werden kann. Das bedeutet, ein Algorithmus ist ausreichend für Ver- und Entschlüsselung.

Für ein Programm, welches die TBlowfish Klasse im ECB Modus verwenden will, sind die einzig wichtigen Zugriffspunkte der Constructor Blowfish_Init, der den Schlüssel als Argument hat und dafür sorgt, dass die Subkeys berechnet werden, und die Prozeduren Decipher und Encipher, welche als Argument einen Referenzparameter auf einen String erhält, welcher die zu ver- oder entschlüsselnden Daten enthält.

Zunächst soll nun der Constructor näher betrachtet werden. Dieser kopiert zunächst die π-Stellen aus einigen Konstanten in die Arrays für P-Array und S-Boxen. In einer for-Schleife werden dann per Typecasting die Bytes des Schlüssels in einem Longword (32 Bits lang) untergebracht, welches dann mit dem 0-ten bis 17-ten Element des P-Array geXORt wird. An dieser Stelle ist es angebracht, das später mehrfach wiederholte Umbauen eines Strings in ein Longword durch Typecasting kurz als Beispiel darzustellen. TByteArray ist als „array [0..3] of Byte“ definiert, was es ermöglicht, die einzelnen Bytes direkt als Zahlenwerte anzusteuern:

```
TByteArray(temp)[3]:=ord(key[j + 1]);
TByteArray(temp)[2]:=ord(key[((j+1) mod keylength) + 1]);
TByteArray(temp)[1]:=ord(key[((j+2) mod keylength) + 1]);
TByteArray(temp)[0]:=ord(key[((j+3) mod keylength) + 1]);
```

Die Variable j wird bei jedem Durchgang um 4 erhöht (modulo keylength). Als nächstes wird, mit dem Verschlüsseln eines leeren Strings beginnend, damit angefangen, den Elementen von P-Array und S-Boxen verschlüsselte Werte zuzuweisen. Sobald dies durchgeführt ist, ist der Constructor auch bereits zuende.

In der Prozedur Encipher, welche die Verschlüsselung durchführt, wird zunächst geprüft, ob die Datenmenge sauber in 64 Bit Blöcke spaltbar ist. Sodann wird damit begonnen, jeden 64 Bit Block mit der von oben bekannten Typecasting-Methode in zwei 32 Bit Longwords zu spalten. Die beiden Hälften werden als Referenzparameter an die Encode Prozedur übergeben. Danach wird, wieder per Typecasting, nur umgekehrt, der Inhalt der Hälften wieder in den String geschrieben. Sollte der letzte Block nicht 64 Bits lang sein, wird er mit 0-Bytes aufgefüllt. Vor dem Ende der Prozedur wird noch an den Anfang des Strings die Originallänge der Daten als Longword geschrieben.

Die Entschlüsselungsprozedur Decipher arbeitet praktisch genauso, nur liest sie vorher die ersten 4 Bytes, um die Originallänge in Erfahrung zu bringen. Außerdem ruft sie Decode anstelle von Encode auf. Nach der Entschlüsselung wird der String auf die Originallänge gesetzt.

In der Encode Prozedur wird zuerst die linke Hälfte mit dem 0-ten Element geXORt. Danach wird die Zeile

```
Round(xR, xL, 1);  Round(xL, xR, 2);
```

mit den Werten 3, 5, . . . , 15 und den Werten 4, 6, . . . , 16 wiederholt. Dann wird die rechte Hälfte mit dem 17-ten Element geXORt und ein Ringtausch durchgeführt.

Die Entschlüsselung verläuft praktisch genauso, nur werden die Werte (0 bis 17) in umkehrter Reihenfolge verwendet. Die Prozedur Round nimmt den ersten Parameter als Referenzparameter mit der Bezeichnung a. Inhalt der Prozeduren Round, sowie der Rückgabewert der Funktionen F und S folgen:

Round `a := (a xor f(b)) xor p[n];`

F
```
((((S(x,0) + S(x,1)) mod pow32)
    xor S(x,2)) + S(x,3)) mod pow32;
```

S `sboxes[i][TByteArray(x)[3-i]];`

Die Funktion OutputLength stellt fest, ob die Datenlänge durch 8 teilbar ist. Falls ja gibt sie diese zurück, ansonsten die nächste durch 8 teilbare Größe.

3.3 Vergleich bei großen Datenmengen

Nachdem nun beide Algorithmen implementiert sind, ist es an der Zeit, sie auf ihre Effizienz zu untersuchen. Hierzu wird eine große Datei mit jedem der Verfahren ver- und entschlüsselt und die Zeit gemessen, die dafür benötigt wird.

3.3.1 Angaben zum Testverfahren

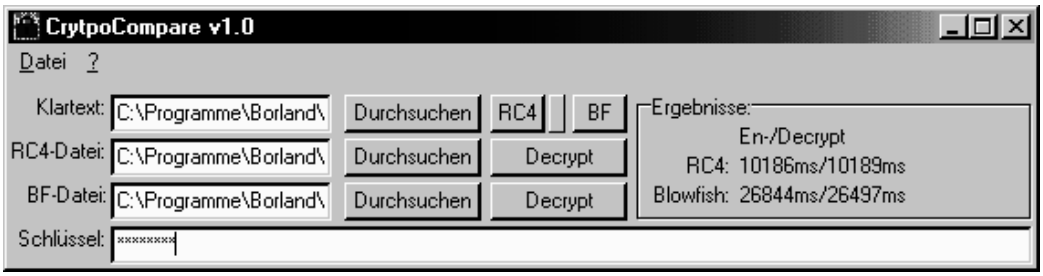

Abbildung 1: Screenshot des Programmes

Zunächst wurde eine 20MB (= 20.971.520 Bytes) große Datei angelegt, die mit Zufallsdaten gefüllt ist. Dies geschah mit Hilfe des Linux-Tools dd, welches folgendermaßen aufgerufen wurde:

```
dd if=/dev/urandom of=hugefile.dat bs=1024k count=20
```

Damit wurden 20 Blöcke von je 1024KB aus /dev/urandom (Kernel-Zufallszahlen-Quellgerät) gelesen und nach hugefile.dat geschrieben. Diese Datei, hugefile.dat, wurde nun in jedem Durchgang zunächst von der Festplatte gelesen und in einem String gespeichert. Dieser String wurde dann jeweils mit RC4 und Blowfish einmal verschlüsselt und wieder entschlüsselt, so dass er für den nächsten Algorithmus wieder in der ursprünglichen Form vorliegt. Gemessen wurde die Zeit vom Aufruf des Constructors des entsprechenden Objektes bis nach dem Aufruf des Destructors. Die verschlüsselten Daten werden hierbei nicht gespeichert. Insgesamt werden die Zeiten der vier Aktionen in zehn Durchgängen gemessen.

3.3.2 Testergebnisse

Durchgang	RC4 Encode	RC4 Decode	BF Encode	BF Decode
1	10051	10171	26857	26466
2	10143	10172	26882	26470
3	10085	10144	26806	26431
4	10071	10248	26822	26408
5	10070	10251	26807	26430
6	10143	10142	26832	26407
7	10130	10412	26807	26424
8	10186	10189	26844	26497
9	10096	10425	26823	26427
10	10177	10177	26837	26406
Durchschnitt	10115,2	10233,1	26831,7	26436,6

Werte in Millisekunden

Abbildung 2: Testergebnisse der 10 Durchgänge

3.3.3 Fazit aus Ergebnissen ziehen

Die Ergebnisse sind ziemlich eindeutig. Blowfish im ECB Modus benötigt in dieser Implementation ungefähr 2.6 Mal so lange wie RC4 für dieselben Daten. Von der Performance her ist demnach in dieser Form eher RC4 (als

CipherSaber2 mit $n = 20$ oder mehr) zu empfehlen. Außerdem gibt es Sicherheitsbedenken beim Einsatz von Blowfish im ECB Modus, was ebenfalls für RC4 spricht. Im CBC Modus hingegen wäre die Sicherheit von Blowfish der von RC4 möglicherweise überlegen, da es keine Möglichkeit gibt, aus ca. 20.000 (bis mehreren Millionen) Nachrichten den Schlüssel zu bestimmen, wobei natürlich zu bedenken ist, dass ein solches Problem auch in Blowfish irgendwann gefunden werden könnte. Nachteilig am Einsatz des CBC Modus wäre allerdings, dass dieser die Performance von Blowfish noch ein Stuck heruntersetzen würde.

4 Allgemeines

Zunächst muss natürlich gesagt werden, dass in einer 15 Seiten langen Facharbeit keine erschöpfende Behandlung aller möglichen kryptographischen Verfahren möglich ist. Doch auch dieser relativ kurze Überblick beginnt schon ein Bild zu vermitteln. So lässt sich aus dem bisherigen ableiten, dass ein schneller Algorithmus nicht unbedingt sicherer sein muss, als ein langsamerer - aber auch das Gegenteil nicht zutreffen muss. Auch kann man ersehen, dass manche Verfahren relativ simpel sind, während andere im Vergleich sehr komplex erscheinen.

Wenn man sich für ein Verfahren entscheiden möchte, so muss man sich zunächst sehr sicher sein, was genau man möchte. Bei einer großen Vielfalt der Möglichkeiten fällt eine Entscheidung nämlich nicht sonderlich leicht, wenn man kein genaues Ziel hat. So muss man zum Beispiel, wenn man sich für ein Block-Verbindungsverfahren entscheiden möchte, abwägen, was einem wichtiger ist - Sicherheit oder Geschwindigkeit. Auch der Zweck sollte mit in die Überlegungen einbezogen werden.

5 Einsatzmöglichkeiten

Die Einsatzmöglichkeiten für kryptographische Verfahren sind vielfältig. So könnte zum Beispiel das Signieren von Nachrichten mit asymmetrischen Verfahren dazu verwendet werden, um eine Geldtransferanweisung zu authentifizieren. Onlineverbindungen zu Banken oder Onlineshops profitieren auch von Verschlüsselung. Ohne diese könnte jeder, der an einem der Router zwischen dem Kunden und dem Server sitzt, die Daten mitlesen oder sogar manipulieren. Verschlüsselung von Daten verhindert dieses Mitlesen und eine digitale Signatur erschwert das Manipulieren erheblich. Noch eine Einsätzmöglichkeit wäre folgende: Eine Firma, welche an einem neuen Produkt arbeitet, kann

diese Informationen verschlüsseln, so dass sie nicht durch Betriebsspionage in die Hände der Konkurrenz fallen kann. Selbiges gilt natürlich auch für militärische Zwecke. Verschlüsselt bringen durch Spionage erlangte Daten nichts. Aber auch für Privatpersonen ist Kryptographie durchaus sinnvoll. Private Daten (z.B. von Alice und Bob vgl. S.6) können verschlüsselt nicht von Fremden gelesen werden, was im Zeitalter der Emailwürmer, welche zufällige Dateien verschicken, gut ist.

A Anhang: Literaturangaben

- [AC] Bruce Schneier, „Applied Cryptography“, Second Edition, John Wiley & Sons, Inc., ISBN 0-471-12845-7
- [CP] http://www.counterpane.com/
- [CS] http://ciphersaber.gurus.com/
- [FOD] The Free On-line Dictionary of Computing (09 FEB 02) (http://www.dict.org/)
- [CAE] http://www.sias.de/kryptologie-caesar.html
- http://www.gnupg.org/
- http://www.goingware.com/encryption/
- http://www.pgpi.org/
- http://world.std.com/∼reinhold/diceware.html
- http://theory.lcs.mit.edu/∼rivest/crypto-security.html Linksammlung
- http://www.arneb.de/ Homepage des Autors