Seminar: Innovative Technologien

Thema: Kryptologie – der RC 5 Algorithmus und das Bovine RC5-64 Projekt

Referenten: Patrick C. F. Ernzer
Thorsten Ferres

Was steht hinter der Bezeichnung RC5? ***3***

Wie funktioniert der RC5 – Algorithmus? ***3***

A) Die Schlüssel Expansion **4**

B) Verschlüsselung **5**

C) Entschlüsselung **6**

Was ist das Besondere am RC5? ***6***

Was ist das Bovine RC5-64 Projekt? ***7***

Wie funktioniert die Verteilung der Schlüssel? ***7***

Wie kann ein Client erkennen, ob er den Schlüssel gefunden hat? ***8***

Anhang (Beispielprogramm) ***9***

Was steht hinter der Bezeichnung RC5?

RC5 ist ein Verschlüsselungsalgorithmus, der von Professor Ronald Rivest vom Massachussets Institute of Technology (MIT) entwickelt und im Dezember 1994 zum ersten Mal veröffentlicht wurde. Die Abkürzung ‚RC‘ steht für ‚Ron's Code‘ bzw. ‚Rivest's Cipher‘, die Nummer stellt eine Art Versionsnummer dar, da Rivest zuvor schon die Algorithmen RC2 und RC4 entwickelt hat.

Seither hat dieser Algorithmus große Aufmerksamkeit unter den Kryptologen erlangt, da es herauszufinden gilt, welche Sicherheit er bietet, d. h. wie gut er gegen Angriffe von Kryptoanalytikern geschützt ist.

Die Erfahrung mit dem DES hat gezeigt, daß eine lange Untersuchung eines Verschlüsselungsalgorithmus bereits vor seiner Veröffentlichung dazu beitragen kann, daß er kryptoanalytischen Attacken für viele Jahre abwehren kann. Auf diese Weise können Schwachstellen schon sehr früh erkannt werden. Daher hofft RSA Data Security Inc., daß der RC5 diesen Prozeß erfolgreich durchlaufen wird und somit eventuell einen Nachfolger für den DES darstellt.

Wie funktioniert der RC5 – Algorithmus?

Der RC5-Algorithmus ist eine so genannte symmetrische Blockchiffre. Hierbei wird der zu verschlüsselnde Klartext in gleich große Blöcke eingeteilt, von denen jeder einzelne mit dem Schlüssel chiffriert wird. Der Chiffretext kann mit dem gleichen Schlüssel auch wieder dechiffriert werden.

Der RC5-Chiffre kann in verschiedenen Varianten angewandt werden, die unterschiedliche Parameter besitzen. Ein bestimmter RC5 Algorithmus wird folgendermaßen angegeben:

RC5 – *w* / *r* / *b*

Die Parameter bedeuten im Einzelnen:

w Die *Wortgröße*, angegeben in Bits. Der Standardwert ist 32; gültige Werte sind 16, 32 und 64. Der RC5-Algorithmus verschlüsselt aber jeweils zwei-Wort-Blöcke, so daß Klartext- und verschlüsselte Textblöcke jeweils 2w lang sind.

r Die Anzahl der Runden, d. h. wie oft die Verschlüsselung durchgeführt wird. Gültige Werte hierfür sind 0, 1, ..., 255.

b Die Anzahl von Bytes, die der geheime Schlüssel *K*. enthält. Gültige Werte sind wiederum 0, 1, ..., 255.

Der RC5 besteht im Wesentlichen aus drei Komponenten: einem Schlüssel-Expansions-Algorithmus, einem Verschlüsselungs- und einem Entschlüsselungsalgorithmus.

Diese Algorithmen basieren alle auf den folgenden drei sehr einfachen Operationen (und ihren entsprechenden Umkehroperationen):

1. Die Addition von Worten modulo 2^w, im Folgenden lediglich mit „+“ bezeichnet.
2. Bitweises exklusives Oder von Worten, bezeichnet mit $\oplus$.
3. Bitrotationen: Die Rotation von x nach links um y Bits wird im Folgenden mit $x <<< y$ bezeichnet. Achtung: Eigentlich haben nur die $\log_2(w)$ unteren Bits von y Einfluß auf diese Rotation.

A) Die Schlüssel Expansion

Hier wird aus dem vom Benutzer des Algorithmus vorgegebenen Schlüssel K eine Schlüsseltabelle S erstellt. Diese Schlüsseltabelle besteht aus einem Feld von $t = 2(r+1)$ binären „Zufallsworten", die mit Hilfe des Schlüssels K erstellt wurden. Der Algorithmus macht sich zwei „magische Konstanten" zu Nutze und besteht aus drei einfachen Teilen.

Die zwei jeweils ein-Wort-großen magischen Konstanten sehen für eine festgelegte Wortgröße w wie folgt aus:

$$P_w = Odd((e-2)\cdot 2^w)$$
$$Q_w = Odd((\varphi-1)\cdot 2^w)$$

mit

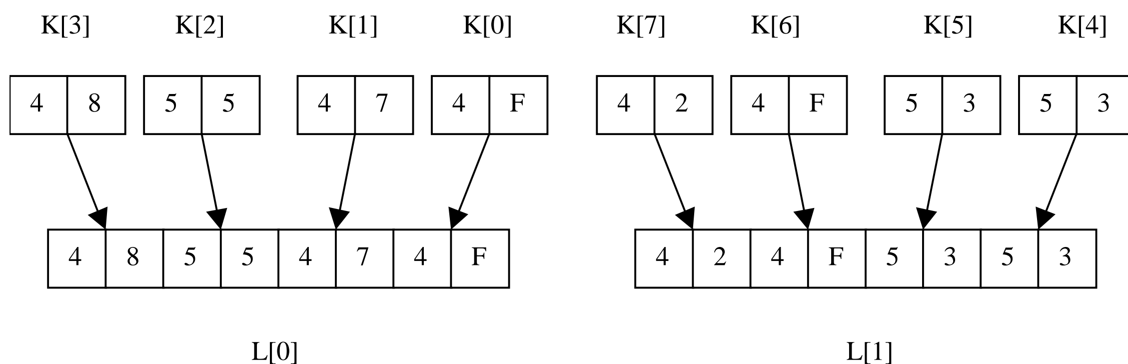
e = 2,718281828459... (Basis des natürlichen Logarithmus, Eulerzahl e)
φ = 1,618033988749... (Goldenes Verhältnis / Goldener Schnitt).

Odd(x) ist diejenige ungerade Ganzzahl, die am nächsten an x liegt. Sollte x eine gerade Ganzzahl sein, so wird sie aufgerundet. Dies kann aber hier normalerweise nicht passieren, da sowohl e als auch φ irrationale Zahlen sind.

Bevor mit dem Aufbau der Schlüsseltabelle begonnen werden kann, müssen die Schlüsselbytes jeweils so zusammengefaßt werden, daß sie Wortgröße ergeben, d. h. die neuen Schlüssel sind jeweils so groß wie ein Wort. Dazu werden die geheimen Schlüsselbytes K[0, ..., b-1] in ein Feld L[0, ..., c-1] kopiert, wobei c = [b/u] ist, und u = w/8 der Anzahl der Bytes pro Wort entspricht. Dies geschieht ganz natürlich, indem einfach u aufeinanderfolgende Schlüsselbytes in ein Wort L eingetragen werden, zunächst die niederwertigen, dann die höherwertigen Bytes. Jede unaufgefüllte Byte-Position in L wird vernullt. Für den Fall, daß b = c = 0 gilt, wird c auf 1 gesetzt und L[0] auf Null.

Beispiel:

Eintragung der 8 Schlüsselbytes K[0] bis K[7] in die beiden Wort-Felder L[0] und L[1] bei einer Schlüssellänge von 64 Bit (= 8 Byte = b) und einer Wortlänge w von 32 Bit (= 4 Byte).

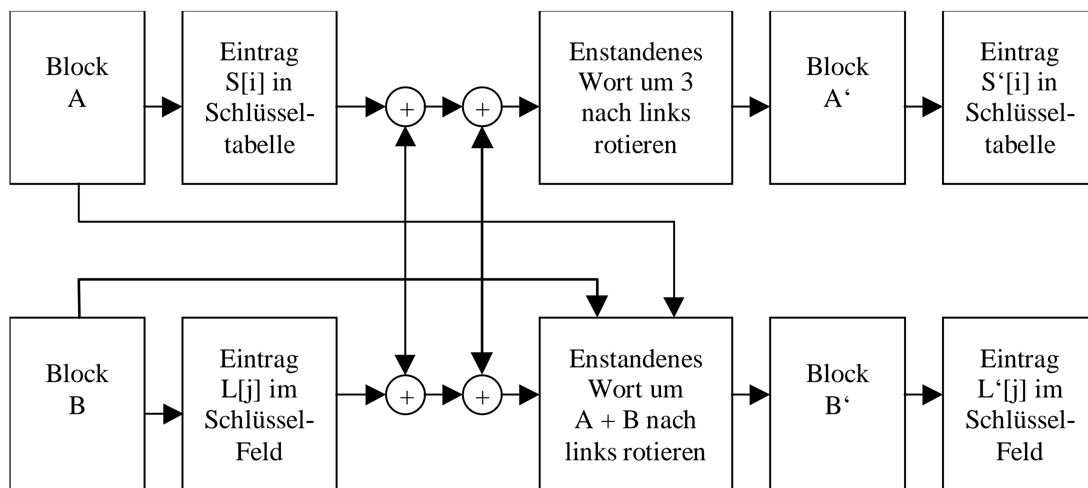


Der nächste Schritt der Schlüsselexpansion initialisiert das Schlüsselfeld S auf von der Wortlänge abhängige, vom Schlüssel jedoch unabhängige, Pseudo-Zufallswerte. Hierbei werden jeweils die magischen Konstanten P_w und Q_w addiert, anschließend wird ein Modulo 2^w durchgeführt, damit der Wert nicht größer als eine Wortlänge werden kann. Da die beiden Werte ungerade sind, hat dieser Prozeß eine Periode von 2^w.

```
S[0]=P_w;
for i=1 to t-1 do
      S[i] = S[i-1] + Q_w;
```

Der dritte Schritt des Algorithmus bringt nun den geheimen Schlüssel des Benutzers in Spiel. Hier werden nämlich die beiden Felder S und L miteinander vermischt, sogar dreimal. Genauer gesagt: Das größere der beiden Felder wird dreimal neu beschrieben, das andere sogar noch öfter.

```
i = j = 0;
A = B = 0;
do 3 * max(t,c) times:
      A = S[i] = (S[i] + A + B) <<< 3;
      B = L[j] = (L[j] + A + B) <<< (A + B);
      i = (i + 1) mod t;
      j = (j + 1) mod c;
```



Die Schlüsseltabelle muß dreimal neu beschrieben werden, da sonst der geheime Schlüssel keinen Einfluß auf alle Elemente dieser Tabelle haben kann. Das erste Element S[0] der Tabelle wird beim ersten Durchlauf lediglich um drei Bits nach links rotiert. Erst beim zweiten Durchlauf kommt durch den Block B der geheime Schlüssel ins Spiel.

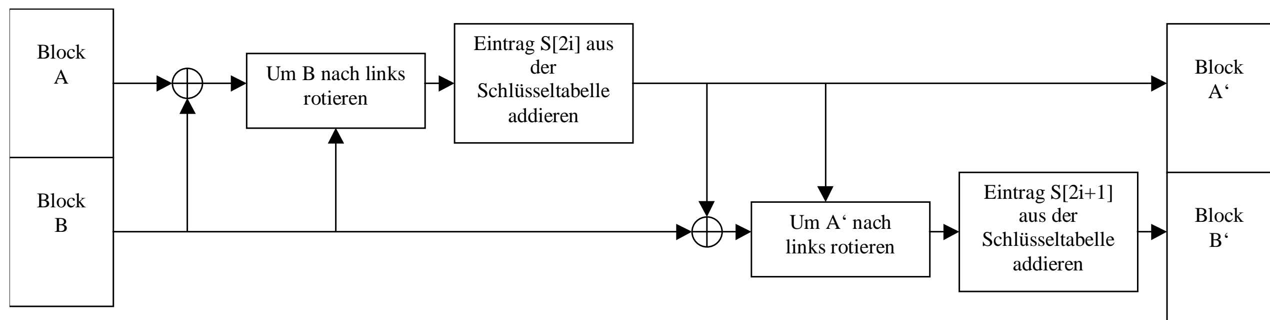
Wie man sieht, ist die Schlüsselexpansion in gewisser Weise eine „Einwegfunktion“, da sich aus der Kenntnis von S nicht so einfach auf den Schlüssel K schließen läßt.

B) Verschlüsselung

Der folgende Pseudocode soll den Verschlüsselungsalgorithmus darstellen. Hierbei wird angenommen, daß der zu verschlüsselnde Eingabeblock in den beiden w-Bit-Registern A und B übergeben wird. Der Ausgabeblock wird ebenfalls in diesen beiden Registern abgelegt.

```
A = A + S[0]
B = B + S[1]
for i = 1 to r do
      A = ((A ⊕ B) <<< B) + S[2i]
      B = ((B ⊕ A) <<< A) + S[2i+1]
```

oder grafisch:



Diese Schritte werden in jeder Runde durchgeführt, wobei die Ausgabe einer Runde als Eingabe der nächsten Runde verwendet wird. Die einzelnen Blöcke werden als mehrfach chiffriert.

C) Entschlüsselung

Die Entschlüsselung erfolgt genauso wie die Verschlüsselung, nur werden jeweils die Umkehrfunktionen verwendet. Außerdem werden diese in umgekehrter Reihenfolge durchgeführt:

```
for i = r downto 1 do
     A = ((A - S[2i]) >>> B) ⊕ B
     B = ((B - S[2i+1]) >>> A) ⊕ A
A = A - S[0]
B = B - S[1]
```

Was ist das Besondere am RC5?

Der RC5-Algorithmus ist ein schneller Blockchiffre, der entwickelt wurde, um ihn sowohl leicht per Software, als auch per Hardware implementieren zu können. Da er durch verschiedene Parameter gesteuert wird - sowohl Blockgröße, als auch die Anzahl der Runden und die Schlüsselgröße sind variabel – bietet er ein hohes Maß an Flexibilität. Es kann somit genau bestimmt werden, welche Durchführungsgeschwindigkeit und welches Maß an Sicherheit die gewünschte Chiffre bieten soll.

Eine weitere Besonderheit, die den RC5 auszeichnet, ist, daß der Algorithmus auf nur drei einfachen Operationen basiert: Addition, Exklusives-Oder und Bitrotation. Dieses Kennzeichen ermöglicht sowohl eine einfache Implementation, als auch einen einfachen Zugang für die Analyse des Algorithmus. Diese beiden Tatsachen gehörten sicherlich zu den Zielen von Ron Rivest, als er den RC5-Algorithmus entwickelte.

Der RC5 verwendet häufig datenabhängige Bitrotationen, was sehr nützlich ist, um sowohl differentielle, als auch lineare Kryptoanalyse zu verhindern. Bei der differentiellen Kryptoanalyse werden zwei Klartexte verschlüsselt, die sich nur durch ein P‘ unterscheiden. Der Unterschied zwischen den verschlüsselten Texten soll mit höherer Wahrscheinlichkeit als normal auftreten. Er wird mit P‘ bezeichnet. Ein solches Paar (C‘, P‘) wird als charakteristisch bezeichnet. Aufgrund dieser Beziehungen zwischen den beiden Klartexten und den beiden verschlüsselten Texten sowie den Unterschieden C‘ und P‘ ist es bei verschiedenen Blockchiffren möglich, einzelne Bits des Schlüssels zu bestimmen. Auch bei der linearen Analyse wird nach statistischen Beziehungen zwischen Klartext, verschlüsseltem Text und Schlüssel gesucht.

Dies wird auch durch die Inkompatibilität zwischen den verwendeten Operationen erreicht. Inkompatibel heißt, daß die Reihenfolge, in der die Operationen durchgeführt werden, nicht verändert werden kann, ohne daß das Resultat geändert wird.

Was ist das Bovine RC5-64 Projekt?

Das Bovine RC5-64 Projekt ist ein Projekt, das im März 1997 von distributet.net ins Leben gerufen wurde. Es beschäftigt sich mit dem von RSA Data Security Inc. ausgerufenen Wettbewerb. Dieser Wettbewerb stellt die Herausforderung an seine Teilnehmer, den Schlüssel einer mit Hilfe des RC5 verschlüsselten Nachricht herauszufinden. Hierzu kann man sich verschiedener kryptoanalytischer Lösungsansätze bedienen.

distributet.net versucht es mit einem sogenannten ‚Brute-force‘ Angriff, d. h. es werden alle möglichen Schlüssel ausprobiert. Im Falle des RC5-64 Algorithmus sind dies insgesamt 2^{64} = 18.446.744.073.709.551.616 (~18,5 Trillionen) verschiedene Schlüssel. Diese Menge ist so unvorstellbar groß, daß man hierfür ca. 584.942.417.355 (~585 Milliarden) Jahre benötigen wurde, wenn jeder Versuch eine Sekunde in Anspruch nehmen würde. Auch wenn heutige Rechner in der Lage sind, eine Entschlüsselung in Bruchteilen von Sekunden durchzuführen, führt dieser Versuch in einem vertretbaren Zeitraum nicht zum Erfolg.

Deshalb werden nicht alle Schlüssel von einem einzigen Computer durchprobiert, sondern mehrere Computer probieren jeweils einen Teil der Schüssel aus. distributet.net möchte hierfür die Rechenzeit der Computer nutzen, die mit dem Internet verbunden sind.

Wie funktioniert die Verteilung der Schlüssel?

Die Verteilung der Schlüssel erfolgt über eine flache Hierarchie von Servern:

An der Spitze steht der Master-Keyserver. Er überwacht die Blöcke, die er schon versendet hat, damit sie geprüft werden, die Blöcke, die schon geprüft und zurückgeschickt wurden und welche Blöcke noch geprüft werden müssen.

Unterhalb des Master-Keyservers stehen die Proxy-Keyserver. Sie dienen als Bindeglied zwischen dem Master-Keyserver und den Clients. Die Proxy-Keyserver erhalten jeweils große Schlüsselblöcke vom Master-Keyserver (sog. Superblocks), die in sehr viel kleinere Blöcke zerteilt werden. Diese kleineren Schlüsselblöcke, sogenannte Standard-Keyblocks, bestehen pro Block aus 2^{28} Schlüsseln. Sie werden letztendlich an die Clients geschickt, die sie überprüfen sollen.

Die Clients schicken die geprüften Schlüssel an die Proxy-Keyserver zurück, die sie wiederum an den Master-Keyserver schicken, der sie in seine Datenbank aufnimmt. Er führt Buch über die schon geprüften und noch zu prüfenden Schlüssel. Somit wird sichergestellt, daß kein Schlüssel mehrmals geprüft wird.

Die Nutzung mehrerer Proxy-Keyserver ist deshalb sinnvoll, da es vorkommen kann, daß einer dieser Server ausfällt. Deshalb arbeiten sie mit einem sogenannte Round-Robin DNS Proxy Zuteilung, die dafür Sorge trägt, daß ein Client automatisch mit einen anderen Proxy-Keyserver verbunden wird, wenn ein Keyserver ausfällt bzw. wenn der Proxy-Keyserver, mit dem der Client sich verbinden möchte, bereits ausgelastet ist. So wird eine ständige Versorgung der Clients mit Proxy-Keyservern sichergestellt.

Optional kann zwischen Proxy-Keyserver und Client noch ein Personal Proxy-Keyserver (Pproxy) geschaltet werden. Sie erhalten Standard-Schlüsselblöcke von den Bovine Proxy-Keyservern und geben sie an die Clients weiter. Sie ermöglichen zum Beispiel der Verteilung von Schlüsseln an Clients hinter einer Firewall. Außerdem werden die Pproxy gerne von Teams genutzt, die am Bovine Projekt teilnehmen. Alle Teammitglieder sind mit dem Personal Proxy-Keyserver verbunden, der wiederum mit einem Bovine Proxy-Keyserver verbunden wird. So können Teams eigene Statistiken über die bereits geprüften Schlüssel aufbauen und entlasten somit den Bovine Statistik Server. Es läßt dem Team auch die Freiheit, in die Statistik nur bestimmte Informationen aufzunehmen.

Wie kann ein Client erkennen, ob er den Schlüssel gefunden hat?

Die zu entschlüsselnde Nachricht enthält am Anfang den Satz: „The unknown message is:“. Konnte ein Client diesen Satz entschlüsseln, so sendet er den Schlüssel an das Bovine Projekt zurück. Dort wird der gefundene Schlüssel an RSA Data Security Inc. weitergegeben, wo die Nachricht nochmals mit diesem Schlüssel dechiffriert wird. Erst jetzt kann geprüft werden, ob es sich wirklich um den gesuchten Schlüssel handelt, da der Schlüssel per Zufall erzeugt wurde und die verschlüsselte Nachricht – bis auf den oben erwähnten Satz – nur einem Mitarbeiter von RSA bekannt war.

Der Client gibt also keine Informationen über den eventuell gefundenen Schlüssel an den Benutzer weiter, da er nicht wirklich sicherstellen kann, ob der gefundene wirklich der gesuchte Schlüssel ist. Außerdem gibt es viele Benutzer, die nicht möchten, daß ihr Rechner sie z. B. durch das Spielen der ‚Ode an die Freude‘ darauf aufmerksam macht, daß er den Schlüssel gefunden hat. Viele lassen den Client auch auf Rechnern in ihrem Büro laufen, wo eine derartige Unterbrechung der Arbeit nur stören würde.

Anhang (Beispielprogramm)

```
// RC5 Test-Programm
//
// V1.0
// Autor: Thorsten Ferres
// Datum: November/Dezember 1998
//
// Dieses Programm stellt eine Implementation des RC5-Verschlüsselungs-
// algorithmus dar. Zwei Blöcke werden verschlüsselt und anschließend
// wieder entschlüsselt, um zu testen, ob Ver- und Entschlüsselungs-
// algorithmus korrekt zusammenarbeiten.

// Libraries einbinden

#include <stdio.h>      // Standard-Ein-Ausgabe
#include <conio.h>      // Weitere Funktionen für Ein-Ausgabe

// Konstantendeklarationen

#define keysize 8       // Schlüsselgröße (in Bytes)
#define wordsize 32     // Blockgröße (in Bits)
#define rounds 12       // Anzahl der Verschlüsselungsrunden
#define t 2*(rounds+1)  // Größe der Verschlüsselungstabelle

// Zwei irrationale Zahlen zur Berechnung von Pseudo-Zufallszahlen
#define eul 2.718281828459    // Eulersche Zahl e
#define phi 1.618033988749   // Goldenes Verhältnis Phi (Goldener Schnitt)

typedef unsigned char key[keysize];       // Typ für den Schlüssel
typedef unsigned long keytable[t];        // Typ für die Schlüsseltabelle
typedef unsigned char wort[wordsize/8+1]; // Typ für einen Block

unsigned long exponent(unsigned int basis,unsigned int expo)
// Funktion, die den Wert (basis hoch expo) berechnet
{
  unsigned int zaehler;       // Schleifenzaehler
  unsigned long wert=1;       // Anfangswert zum Rechnen

  // Diese Schleifen multipliziert wert expo-mal mit basis
  for(zaehler=0;zaehler<expo;zaehler++)
  {
    wert = wert * basis;
  }

  return(wert);   // Berechneten Wert zurückliefern
}

//*************************************************************************
//*                                                                       *
//*   Funktion zum Aufbau der Schlüsseltabelle                            *
//*                                                                       *
//*   Parameter:                                                          *
//*                                                                       *
//*   key used_key      :     Schlüssel, der zur Verschlüsselung benutzt*
//*                           wird                                        *
//*   keytable s  :     Dieses Feld nimmt die berechnete Schlüssel-     *
//*                     tabelle auf                                       *
//*                                                                       *
//*************************************************************************

void key_expansion(key used_key, keytable s)
```

```
{
  unsigned int u,c;     // Zwei Integerwert zur Berechnung von Feldgrößen

  unsigned int zaehler1, zaehler2, zaehler3;    // Drei Schleifenzähler

  unsigned int maxi;    // Integervariable, die einen Maximalwert aufnimmt

  unsigned long L[keysize/(wordsize/8)]; // Schlüsselfeld, daß Schlüssel
                            aufnimmt, die genauso groß sind wie ein Block

  unsigned long Pw, Qw, A, B; // Werte zum Berechnen der Schlüsseltabelle

  unsigned long zwischen;// Variable zum Speichern von Zwischenergebnissen

  u=wordsize/8;   // Anzahl der Bytes pro Block

  c=keysize/u;    // Anzahl der Schlüsselblocks, die genauso groß sind wie
                  ein zu verschlüsselnder Block

  // Diese Schleife baut die einzelnen Bytes des Schlüssels zu
  // Schlüsselblocks zusammen, die genauso groß sind wie ein zu
  // verschlüsselnder Block

  for(zaehler1=0; zaehler1<c; zaehler1++)// Anzahl der Schlüsselblocks-mal
                                          die Schleife durchlaufen
  {
    L[zaehler1]=0;
    zaehler3=0;
    for(zaehler2=0; zaehler2<u; zaehler2++)// Anzahl der Bytes pro
                                             Schlüsselblock-mal
                                             durchlaufen
    {
      L[zaehler1]=L[zaehler1]+(used_key[zaehler1*u+zaehler2] << zaehler3);
      zaehler3=zaehler3+8;
      // zum Schlüsselblock Schlüssel addieren, nachdem dieser um 0,8,16,24
      nach links geschoben wurde
    }
  }

  // Pseudozufallszahl Pw erzeugen
      Pw = (int)((eul-2)*exponent(2,wordsize-1) + (eul-
      2)*exponent(2,wordsize-1));
  // Wenn Pw gerade ist, dann die nächste ungerade Zahl nehmen
  if ((Pw % 2)==0) Pw++;

  // Pseudozufallszahl Qw erzeugen
      Qw = (int)((phi-1)*exponent(2,wordsize-1) + (phi-
      1)*exponent(2,wordsize- 1));
  // Wenn Qw gerade ist, dann die nächste ungerade Zahl nehmen
  if ((Qw % 2)==0) Qw++;

  s[0] = Pw;      // erster Eintrag der Schlüsseltabelle sei zunächst Pw
  for(zaehler1=1;zaehler1<t;zaehler1++)
  {
      // Alle weiteren Einträge seien zunächst um Qw (modulo 2^wordsize)
      größer als der Vorhergehende
    s[zaehler1]=(s[zaehler1-1] + Qw) /*% exponent(2,wordsize)*/;
  }

  zaehler1=zaehler2=0;  // Zähler initialisieren
  A=B=0;          // Werte zum Rechnen initialisieren
```

```
  if(t<c) maxi=3*c; else maxi=3*t;  // Anzahl der Schleifendurchläufe
                                       bestimmen

  for(zaehler3=0; zaehler3<maxi; zaehler3++)
  {
     zwischen = (s[zaehler1] + A + B);
     s[zaehler1] = ((zwischen) << 3) + ((zwischen) >> (wordsize-3));
     A = s[zaehler1];
     zwischen = (L[zaehler2] + A + B);
     L[zaehler2] = ((zwischen) << ((A+B) % wordsize)) + ((zwischen) >>
(wordsize-((A+B) % wordsize)));
     B = L[zaehler2];

     zaehler1 = (zaehler1 + 1) % t;
     zaehler2 = (zaehler2 + 1) % c;
  }

}

//***************************************************************************
//*                                                                         *
//*   Funktion zum Verschlüsseln                                            *
//*                                                                         *
//*   Parameter:                                                            *
//*                                                                         *
//*   unsigned long A, B      :     Zu verschlüsselnde Textblöcke á 32      *
//*                                 Bit                                     *
//*   unsigned long *CA, *CB  :     Zeiger auf die Adressen für die         *
//*                                 Rückgabewerte (verschlüsselter Text)   *
//*     keytable s            :     Die vorher berechnete Schlüssel-        *
//*                                 tabelle                                 *
//*                                                                         *
//***************************************************************************

void encryption(unsigned long A,unsigned  long B,unsigned long *CA,unsigned
long *CB, keytable s)
{
  unsigned int zaehler;       // Ein Zähler zum Durchzählen der
                              Verschlüsselungsrunden
  unsigned long zwischen1, zwischen2, zwischen3;// Variablen zur Aufnahme
  von Zwischenergebnissen

  *CA = (A + s[0]) /*% exponent(2, wordsize)*/;
  *CB = (B + s[1]) /*% exponent(2, wordsize)*/;

  for(zaehler=1; zaehler<=rounds; zaehler++) // Die einzelnen
  Verschlüsselungsrunden durchlaufen
  {
    zwischen1 = *CA ^ *CB;          // Vorberechnungen: 1. Block
    zwischen2 = *CB % wordsize;
    zwischen3 = wordsize - zwischen2;
    // Verschlüsselung des 1. Blocks
      *CA = ((zwischen1 << zwischen2) + (zwischen1 >> zwischen3)) +
      s[2*zaehler];

    zwischen1 = *CB ^ *CA;          // Vorberechnungen: 2. Block
    zwischen2 = *CA % wordsize;
    zwischen3 = wordsize - zwischen2;
    // Verschlüsselung des 2. Blocks
      *CB = ((zwischen1 << zwischen2) + (zwischen1 >> zwischen3)) +
      s[2*zaehler+1];
  }
```

```
}

//***********************************************************************
//*                                                                     *
//*   Funktion zum Entschlüsseln                                        *
//*                                                                     *
//*   Parameter:                                                        *
//*                                                                     *
//*   unsigned long A, B      :       Verschlüsselte Textblöcke á 32 Bit  *
//*                                                                     *
//*   unsigned long *CA, *CB  :     Zeiger auf die Adressen für die     *
//*                                 Rückgabewerte (entschlüsselter Text)  *
//*     keytable s            :     Die vorher berechnete Schlüssel-    *
//*                                 tabelle                             *
//*                                                                     *
//***********************************************************************

void decryption(unsigned long A,unsigned  long B,unsigned long *DA,unsigned
long *DB, keytable s)
{
      unsigned int zaehler;                    // Ein Zähler zum
      Durchzählen der Verschlüsselungsrunden
      unsigned long zwischen1, zwischen2, zwischen3;// Variablen zur
      Aufnahme von Zwischenergebnissen

  *DA=A;    // Werte zunächst in Rückgabewerte kopieren
  *DB=B;

      for(zaehler=rounds; zaehler>=1; zaehler--) // Die einzelnen
      Verschlüsselungsrunden rückwärts durchlaufen
      {

      zwischen1 = *DB - s[2*zaehler+1];   // Vorberechnungen 2. Block
      zwischen2 = *DA % wordsize;
      zwischen3 = wordsize - zwischen2;
      // Entschlüsselung des 2. Blocks
     *DB = ((zwischen1) >> zwischen2)+(zwischen1 << zwischen3)^*DA;

      zwischen1 = *DA - s[2*zaehler];     // Vorberechnungen 1. Block
      zwischen2 = *DB % wordsize;
      zwischen3 = wordsize - zwischen2;
      // Entschlüsselung des 1. Blocks
      *DA = ((zwischen1) >> zwischen2)+(zwischen1 << zwischen3)^*DB;
      }

  // ersten Schritt der Verschlüsselung rückgängig machen
  *DA = (*DA - s[0]) /*% exponent(2,wordsize)*/;
  *DB = (*DB - s[1]) /*% exponent(2,wordsize)*/;

}

//***********************************************************************
//*                                                                     *
//*   Funktion zum Umwandeln eines Wortes mit vier Buchstaben in eine   *
//*   32-Bit-Zahl                                                       *
//*                                                                     *
//*   Parameter:                                                        *
//*                                                                     *
//*   wort block  :       Umzuwandelndes Wort                           *
//*                                                                     *
//*   Rückgabewert                                                      *
//*   unsigned long     :     Die 32-Bit-Zahl                           *
//*                                                                     *
```

```
//**********************************************************************

unsigned long block_to_long(wort block)
{
  unsigned int zaehler;       // Zaehler
  unsigned long returnwert=0; // Rückgabewert (mit 0 initialisieren)

  // Schleife wird Buchstabe für Buchstabe durchlaufen (Byteweise)
  for(zaehler=0; zaehler<(wordsize/8);zaehler++)
  {
      // alten Wert um ein Byte (*256) nach links schieben und neuen Wert
      addieren
      returnwert = returnwert * 256 + block[zaehler];
  }

  return(returnwert);
}

//**********************************************************************
//*                                                                    *
//*   Funktion zum Umwandeln einer 32-Bit-Zahl in ein Wort mit vier Buch- *
//*   staben                                                           *
//*                                                                    *
//*   Parameter:                                                       *
//*                                                                    *
//*   unsigned long chiffre   :       Umzuwandelnde 32-Bit-Zahl        *
//*                                                                    *
//*   wort returnvalue  :     Das umgewandelte Wort (mit 4 Buch-       *
//*                           staben)                                  *
//**********************************************************************

void long_to_block(unsigned long chiffre, wort returnvalue)
{

  int zaehler;                              // Zaehler
  unsigned long zwischen=chiffre;           // Zwischenvariable zum Rechnen

  returnvalue[wordsize/8]='\0';
  for(zaehler=(wordsize/8-1);zaehler>=0;zaehler--)
  {
     returnvalue[zaehler]=zwischen % 256; // Die untersten acht Bits jeweils
in ein Feld eintragen
     zwischen=zwischen / 256;                  // und diese Bits entfernen
(/256)
  }
}

//----------------------------Hauptprogramm------------------------------
//
// Das Hauptprogramm dient lediglich zum Testen der oben geschriebenen
// Verschlüsselungs- und Entschlüsselungsroutinen. Es fordert den
// Benutzer auf, zwei Worte mit vier Buchstaben und einen Schlüssel mit
// acht Buchstaben einzugeben, verschlüsselt diese dann und zeigt den ver-
// schlüsselten Text an. Anschließend entschlüsselt es diesen Text wieder
// und zeigt somit wieder den eingegebenen Text an
//

void main(void)
{

      wort erster_block, zweiter_block;   // Erster und zweiter
      Klartextblock als String
```

```
    unsigned long block1,block2;       // Erster und zweiter
    Klartextblock als Integer
    wort erste_chiffre, zweite_chiffre;      // Erster und zweiter
    Chiffreblock als String
    unsigned long chiffre1, chiffre2;  // Erster und zweiter Chiffreblock
    als Integer
    wort erstes_ergebnis, zweites_ergebnis; // Erster und zweiter
    entschlüsselter Chiffre als String
    unsigned long ergebnis1, ergebnis2;     // Erster und zweiter
    entschlüsselter Chiffre als Integer
    key benutzter_schluessel;            // Benutzer Schlüssel (als Byte-
    Feld, da Char=Byte in C)
    unsigned char schloss[keysize+1];    // Der gleiche Schlüssel als
    String (ein Feld mehr für abschließende '\0')
    int zaehler;                          // Zaehler der Schleife, die
    den Schlüssel umwandelt
    keytable schluesseltabelle;

clrscr();      // Bildschirm löschen

// Zwei Klartextblöcke á vier Buchstaben eingeben

printf("\n Bitte geben Sie ein Wort mit vier Buchstaben ein:");
scanf("%4s",erster_block);

printf("\n Bitte geben Sie noch ein Wort mit vier Buchstaben ein:");
scanf("%4s",zweiter_block);

// Blöcke in Integer umwandeln

block1=block_to_long(erster_block);
block2=block_to_long(zweiter_block);

// Einen Schlüssel mit acht Buchstaben eingeben

printf("\n\n Bitte geben Sie einen Schlüssel mit acht Buchstaben ein:");
scanf("%8s",schloss);

// Schlüssel umwandeln (in anderes Feld kopieren)

for(zaehler=0;zaehler<keysize;zaehler++)
{
  benutzter_schluessel[zaehler]=schloss[zaehler];
}

// Routine zur Schlüsselexpansion aufrufen

key_expansion(benutzter_schluessel,schluesseltabelle);

// Die eingegebenen Blöcke verschlüsseln (verschlüsselte
// Integerwerte in chiffre1 und chiffre2)

encryption(block1,block2,&chiffre1,&chiffre2,schluesseltabelle);

// Verschlüsselte Integerwerte wieder in Strings umwandeln

long_to_block(chiffre1,erste_chiffre);
long_to_block(chiffre2,zweite_chiffre);

// Verschlüsselte Blöcke ausgeben

printf("\n\n Das erste Chiffre lautet: %s",erste_chiffre);
printf("\n Das zweite Chiffre lautet: %s",zweite_chiffre);
```

```
  // Die verschlüsselten Blöcke wieder dechiffrieren

  decryption(chiffre1,chiffre2,&ergebnis1,&ergebnis2,schluesseltabelle);

  // Ergebnisse wieder in Strings umwandeln

  long_to_block(ergebnis1,erstes_ergebnis);
  long_to_block(ergebnis2,zweites_ergebnis);

  // Ergebnisse (dechiffrierte Blöcke) ausgeben

  printf("\n\n Das erste Ergebnis lautet: %s",erstes_ergebnis);
  printf("\n Das zweite Ergebnis lautet: %s",zweites_ergebnis);

  getch();  // Warten auf Tastendruck
}
```