Autor: AXMD **Verfasst am:** Sa 25.03.06 13:18

Titel: Kryptologie: Einfaches Feistel-Netzwerk

Kryptologie: Erläuterung und Implementation eines einfachen Feistel-Netzwerks

1. Einleitung

In diesem Tutorial wird es um die Erläuterung und Implementation eines einfachen Feistel-Netzwerks gehen - das Verständnis eines solchen Netzwerks ist grundlegend für das Verständnis von DES bzw. anderen Blockchiffren; wobei hier ein sehr einfaches Beispiel zur Implementation gewählt wird, um die Komplexität in Grenzen zu halten. Wer weiterführende Literatur sucht, ist bei Wikipedia (Feistelchiffre) bzw. Wikipedia (Blockchiffre) gut aufgehoben.

2. Grundlagen

2.1 XOR

Die XOR-Operation entspricht einer Addition ohne Übertrag, das heißt

Delphi-Quelltext
```
Zahl1 xor Zahl2
```

addiert die beiden Zahlen binär stellenweise, wobei die Addition von 1 und 1 einfach 0 ergibt und an der nächsten Stelle nicht berücksichtigt wird. Beispiel:

Quelltext
```
Zahl1: 1100
Zahl2: 1010
       ---- (xor)
Erg.:  0110
```

An der ersten (höherwertigsten) Stelle ist genau das passiert - die Addition von 1 und 1 hat 0 ergeben, der Übertrag wird allerdings nicht berücksichtigt.
Das besondere an der XOR-Operation ist die Tatsache, dass zweimaliges XOREn mit einer Zahl wieder den Ausgangswert ergibt. Um das Beispiel von oben fortzuführen:

Delphi-Quelltext
```
Zahl1 xor Zahl2
```

ergab 0110. Wird 0110 nun ein weiteres Mal xor Zahl2 gerechnet, erhalten wir wieder Zahl1:

Quelltext
```
Erg.:  0110
Zahl2: 1010
       ---- (xor)
       1100 (= Zahl1)
```

Diese Tatsache kann auch so erklärt werden: eine Zahl xor sich selbst ergibt immer 0 (da

Delphi-Quelltext
```
1 xor 1 = 0
```

und

Delphi-Quelltext

```
0 xor 0 = 0
```

), also ist

Delphi-Quelltext
```
x xor Zahl xor Zahl
```

gleich

Delphi-Quelltext
```
x xor (Zahl xor Zahl)
```

und da eine Zahl xor sich selbst 0 ergibt

Delphi-Quelltext
```
x xor 0
```

. Und eine Zahl xor 0 ergibt die Zahl selbst, da sie nichts verändert (1 xor 0 ergibt 1, ebenso wie 0 xor 0 0 ergibt).
Darauf aufbauend kann nun eine weitere Tatsache festgestellt werden: wird Zahl1 mit mehreren anderen Zahlen hintereinander geXORt und anschließend ein weiteres Mal, ist das Ergebnis wieder die ursprüngliche Zahl1 - wobei hier die Reihenfolge, in der die anderen Zahlen geXORt werden, keine Rolle spielt. Beispiel:

Quelltext
```
Zahl1: 1100             a) Erg.:  1001          b) Erg.:  1001
Zahl2: 1010                Zahl3: 1111             Zahl2: 1010
       ---- (xor)                 ---- (xor)             ---- (xor)
Erg.': 0110                       0110                    0011
Zahl3: 1111                Zahl2: 1010             Zahl3: 1111
       ---- (xor)                 ---- (xor)             ---- (xor)
Erg.:  1001                       1100                    1100
```

a) und b) liefern das gleiche Ergebnis. Weiters gilt folgendes: wenn

Delphi-Quelltext
```
Zahl1 xor Zahl2 = Zahl
```

ist

Delphi-Quelltext
```
Zahl1 xor Zahl = Zahl2
```

- das nur der Vollständigkeit halber.

2.2 Bitoperationen zum "Trennen" von Zahlen

In vielen Anwendungen ist es wichtig, Zahlen mit z.B. 8 Bit in zwei Zahlen à 4 Bit zu zerlegen - diese Werte wurden gleich in Anlehnung an das Beispiel (vgl. unten) gewählt.
Delphi stellt für eine 8 Bit große Zahl prinzipiell 2 Datentypen zur Verfügung: Char und Byte. Byte ist zum Rechnen besser geeignet, da es eine Zahl darstellt (Char stellt ein ASCII-Zeichen dar). Zur kurzen Wiederholung: mit Chr bzw. Ord kann der eine Datentyp in den anderen umgewandelt werden.
Das eigentliche Trennen der Zahl ist denkbar einfach: die rechte Hälfte kann ohne Veränderung beibehalten werden - es müssen nur die ersten 4 Bit auf 0 gesetzt werden - das ist am einfachsten mit einer AND-Operation möglich - man spricht bei diesem "Ausschneiden" auch von einer Maske.
Beispiel:

Quelltext
```
Zahl:   0101 1010
Rechts: 0000 1010
```

bzw. im Detail:

Quelltext
```
Zahl:   0101 1010
Maske:  0000 1111
        --------- (and)
Rechts: 0000 1010
```

Die Maske ist an all jenen Stellen 1 an denen die ursprüngliche Zahl (bzw. das ursprüngliche Bit) beibehalten werden soll, an all jenen Stellen, an denen im Ergebnis eine 0 stehen soll, 0. Der Grund ist relativ einfach: ein AND mit 0 ergibt immer 0, egal welchen Wert das andere Bit hat. Die angegebene Maske hat in Delphi entweder den Dezimalwert 15 oder alternativ den hexadezimalen Wert F. Letzterer wird für Masken dieser Art bevorzugt verwendet, da er deutlicher sichtbar macht, welcher Teil ausgeschnitten wird.
Der linke Teil muss offensichtlich erst um 4 Bit nach rechts verschoben werden, um verwendet werden zu können:

Quelltext
```
Zahl:  0101 1010
Links: 0000 0101
```

Praktisch ist hier, dass das Nach-Rechts-Schieben die 4 "offenen" Stellen automatisch mit Nullen auffüllt. Der komplette Code zum Zerlegen eines Bytes in 2 4-Bit-Werte (hier ebenfalls Bytes, da Delphi keinen Datentyp mit 4 Bit bereitstellt, mit dem es sich einfach rechnen ließe):

Delphi-Quelltext
```
procedure SeparateBlock(ABlock: Byte; var _Left, _Right: Byte);
begin
  _Left := ABlock shr 4;
  _Right := ABlock and $0F;
end;
```

Das Zusammenfügen von 2 4-Bit-Werten erfolgt prinzipiell umgekehrt zum Zerlegen: der rechte Teil kann prinzipiell bleiben wie er ist (er muss nur wieder mit der Maske $0F verUNDet werden, um eventuell gesetzte höhere Bits auszulöschen). Der linke Teil muss erst noch um 4 Stellen nach links verschoben werden. Anschließen müssen die beiden (verschobenen und maskierten) Teile zusammengefügt werden - das ist mit einer einfachen OR-Operation möglich. Beispiel:

Quelltext
```
Rechts: 0000 1010
Links:  0101 0000
        --------- (or)
Erg.:   0101 1010
```

Das OR entspricht hier einer Addition der beiden Teile. Der vollständige Code für das Zusammenfügen von 2 4-Bit-Werten zu einem 8-Bit-Wert lautet:

Delphi-Quelltext
```
function MergeBlocks(Block1, Block2: Byte): Byte;
begin
```

```
  Result := (Block1 shl 4) or (Block2 and $0F); //0F => Maske, um höherwertige Bits zu entfernen
end;
```

3. Feistel

3.1 Die Idee hinter Feistel

Die Idee, die hinter einem Feistel-Netzwerk steckt, ist prinzipiell sehr einfach: ein Block von 8 Bit wird in zwei Blöcke von je 4 Bit zerlegt (im folgenden linker bzw. rechter Block genannt). Danach werd die beiden Blöcke über n Runden (wobei n frei wählbar ist) modifiziert (Details siehe unten) und am Ende wieder zu einem 8-Bit-Block zusammengesetzt.
Beim Verschlüsseln sieht diese Modifikation wie folgt aus (L und R sind hier der linke bzw. rechte Block, L' und R' die neu berechneten linken und rechten Blöcke):

-

Quelltext
```
L' = R
```

[:82de2ac54f*:82de2ac54f]

Quelltext
```
R' = L xor f(R, K)
```

wobei K ein beliebig gewählter, 4 Bit langer Schlüssel ist und f eine beliebige Funktion ist, die aus R und dem Schlüssel einen Wert berechnet. Und genau an dieser Stelle die wohl wichtigste Tatsache: die Umkehrbarkeit.

3.2 Umkehrbarkeit

f kann eine beliebige Funktion sein - sie kann rein theoretisch auch immer 0 zurückgeben (unabhängig von K und R), sie muss also nicht umkehrbar, d.h. bijektiv, sein. Die Umkehrbarkeit wird allein durch die XOR-Operation gewährleistet (vgl. 2.1:

Delphi-Quelltext
```
Ausgangswert xor Schluessel xor Schluessel = Ausgangswert
```

). Zum Entschlüsseln wird also die exakt gleiche Funktion f verwendet, da das zweite XOR (mit dem gleichen Wert) beim Entschlüsseln dafür sorgt, dass wieder der Ausgangswert herauskommt.
Rein theoretisch kann in jeder Runde Runde ein anderer Schlüssel verwendet werden, in diesem einfachen Beispiel bleibt der Schlüssel allerdings immer gleich. Als Funktion wird (gewählt)

Quelltext
```
2 * R xor K
```

verwendet - wie bereits erwähnt, muss f nicht umkehrbar sein. Hier der Delphi-Code:

Delphi-Quelltext
```
function f(ABlock: Byte; AKey: Byte): Byte;
begin
  Result := (ABlock shl 1) xor AKey; //Links-Shift (Multiplikation mit 2); XOR Schlüssel
end;
```

Zur Information: eine Multiplikation entspricht einem einfachen Links-Shift, daher das

Delphi-Quelltext

```
shl 1
```

anstatt dem

Delphi-Quelltext
```
* 2
```

.

3.3 Blöcke

Da wir in unserem Beispiel mit 8 Bit großen Blöcken hantieren muss beim Verschlüsseln alles in 8-Bit-Blöcke zerlegt werden - eben weil ein ASCII-Zeichen genau 8 Bit groß ist, ist das Beispiel relativ einfach: 1 Zeichen entspricht einem Block. Um sich das eigentliche Zerlegen zu ersparen, wird der angegebene String zeichenweise durch das Feistel-Netzwerk geschickt und der Cipher-Text (in diesem Fall das Zeichen, das Feistel zurückliefert) an das Ergebnis angehängt.

3.4 Zahlenbeispiel

Abschließend soll das nun konstruierte Feistel-Netzwerk mit einem einfachen Zahlenbeispiel getestet werden. Es soll ein einziges Zeichen ver- und wieder ent-schlüsselt werden. Als Zeichen wird das große A (ASCII 65) gewählt, als Rundenanzahl wird 2 gewählt, als Schlüssel 5:

Quelltext
```
ASCII (binär): 0100 0001
Linker Teil:   0100
Rechter Teil:  0001
```

1. Runde:

Quelltext
```
Linker Teil:  0001 (ehemaliger rechter)
Rechter Teil: 0100 xor f(0001, 0101) = 0100 xor (0010 xor 0101) = 0100 xor 0111 = 0011
```

2. Runde:

Quelltext
```
Linker Teil:  0011 (ehemaliger rechter)
Rechter Teil: 0001 xor f(0011, 0101) = 0001 xor (0110 xor 0101) = 0001 xor 0011 = 0010
```

Zusammengesetzt ist der Cipher-Text dann

Quelltext
```
0011 0010
```

. Beim Entschlüsseln müssen nur der rechte und der linke Teil vertauscht werden:

Quelltext
```
ASCII (binär): 0011 0010
Linker Teil:   0010 (vertauscht!)
Rechter Teil:  0011 (vertauscht!)
```

1. Runde:

Quelltext
```
Linker Teil:  0011 (ehemaliger rechter)
Rechter Teil: 0010 xor f(0011, 0101) = 0010 xor (0110 xor 0101) = 0010 xor 0011 = 0001
```

2. Runde:

Quelltext
```
Linker Teil:  0001 (ehemaliger rechter)
Rechter Teil: 0011 xor f(0001, 0101) = 0011 xor (0010 xor 0101) = 0011 xor 0111 = 0100
```

Das Ergebnis ist wieder

Quelltext
```
0100 0001
```

(rechten und linken Teil wieder vertauschen!). Was hier sehr schön sichtbar wird, ist die aus 2.1 bekannte Tatsache, dass die Reihenfolge, in der die XOR-Operationen "rückgängig" gemacht werden, keine Rolle spielen.

3.5 Implementation in Delphi

Nun noch ein paar Worte zur Implementation des Feistel-Netzwerks in Delphi: jene Befehle, die sich in jeder Runde wiederholen (vgl. Aufzählung in 3.1), wurden in einer Methode Feistel zusammengefasst:

Delphi-Quelltext
```
procedure Feistel(var _Left, _Right: Byte; AKey: Byte);
var
  _LeftOld: Byte;
begin
  _LeftOld := _Left;
  _Left := _Right; //Links = Rechts
  _Right := _LeftOld xor f(_Right, AKey); //Rechts = Links (+) f(Rechts, Schlüssel)
end;
```

_LeftOld muss gesichert werden, da _Left ja durch _Right überschrieben, aber für die Berechnung des neuen _Right benötigt wird.
Die Ver- bzw. Entschlüsselung unterteilt den zu ver-/entschlüsselnden Text in 8-Bit-Blöcke (in diesem Fall in einer einfachen Schleife, die durch alle Zeichen des Strings iteriert), teilt diese Blöcke in zwei 4-Bit-Blöcke auf und ruft den Feistel-Algorithmus n mal auf (hier NumberOfPasses genannt, im Beispiel 2). Danach werden die beiden 4-Bit-Blöcke wieder zu einem 8-Bit-Block zusammengefügt und als Zeichen an der jeweiligen Stelle im Ergebnis (Result-String, Rückgabewert der Funktion) gespeichert. Die Funktion EnDecrypt kann ver- und entschlüsseln. Der letzte Parameter gibt an, ob ver- oder entschlüsselt werden soll - beim Entschlüsseln wird die Feistel-Funktion einfach mit vertauschten Parametern (links <-> rechts) aufgrufen.

Delphi-Quelltext
```
function EnDecrypt(AText: String; AKey: Byte; Encrypt: Boolean): String;
const
  NumberOfPasses = 2; //Anzahl Durchgänge
```

```
var
  i, j: Integer;
  _Left, _Right: Byte;
begin
  Result := AText;
  for i := 1 to Length(Result) do begin
    SeparateBlock(Ord(Result[i]), _Left, _Right);
    for j := 1 to NumberOfPasses do begin
      if Encrypt then
        Feistel(_Left, _Right, AKey)
      else
        Feistel(_Right, _Left, AKey); //Zum Entschlüsseln rechts und links vertauschen (invers)
      end;
    Result[i] := Chr(MergeBlocks(_Left, _Right));
    end;
end;
```

3.6 Vollständiger Code

Hier noch der vollständige Delphi-Code:

Delphi-Quelltext

```
//Implementation eines einfachen Feistel-Netzwerks; (c) by Dust Signs Andreas Unterweger 2006

procedure SeparateBlock(ABlock: Byte; var _Left, _Right: Byte);
begin
  _Left := ABlock shr 4;
  _Right := ABlock and $0F;
end;

function MergeBlocks(Block1, Block2: Byte): Byte;
begin
  Result := (Block1 shl 4) or (Block2 and $0F); //0F => Maske, um höherwertige Bits zu entfernen
end;

function f(ABlock: Byte; AKey: Byte): Byte;
begin
  Result := (ABlock shl 1) xor AKey; //Links-Shift (Multiplikation mit 2); XOR Schlüssel
end;

procedure Feistel(var _Left, _Right: Byte; AKey: Byte);
var
  _LeftOld: Byte;
begin
  _LeftOld := _Left;
  _Left := _Right; //Links = Rechts
  _Right := _LeftOld xor f(_Right, AKey); //Rechts = Links (+) f(Rechts, Schlüssel)
end;

function EnDecrypt(AText: String; AKey: Byte; Encrypt: Boolean): String;
const
  NumberOfPasses = 2; //Anzahl Durchgänge
var
  i, j: Integer;
  _Left, _Right: Byte;
begin
  Result := AText;
  for i := 1 to Length(Result) do begin
    SeparateBlock(Ord(Result[i]), _Left, _Right);
    for j := 1 to NumberOfPasses do begin
      if Encrypt then
        Feistel(_Left, _Right, AKey)
      else
        Feistel(_Right, _Left, AKey); //Zum Entschlüsseln rechts und links vertauschen (invers)
      end;
    Result[i] := Chr(MergeBlocks(_Left, _Right));
    end;
end;
```

AXMD

PS. C++-Quelltext auf Nachfrage erhältlich

Autor: AXMD **Verfasst am:** Do 26.07.07 19:15

Aufgrund der zahlreichen Nachfragen per Email hier der C++-Code im Anhang. Hab die Original-CPP leider nicht mehr, nur mehr ein formatiertes PDF.

AXMD

Anhänge:

feistel_cpp_full.pdf (38.24 KB)

C++-Code Feistel