Stromchiffren (3)

Krypto II

VL 17

29.11.2018

Buch des Tages

Titel: Cryptography Made Simple

Autor(en): Nigel P. Smart

Verlag: Springer, 2016

Umfang: ca. 480 Seiten

Hinweise zum Inhalt:

Das Buch gibt einen umfassenden Überblick zu mathematischen Grundlagen der Kryptographie und behandelt auch weniger gebräuchliche Themen wie Secure Multi-Party Computation, Zero-Knowledge Proofs, oder Commitments and Oblivious Transfer.

Symmetrische Verschlüsselung wird hauptsächlich anhand historischer Chiffren besprochen. Blockchiffren und Stromchiffren werden in vergleichsweise kurzen Kapiteln abgehandelt.

Die Darstellung ist gelungen, im Vergleich zu anderen Einführungen aber vergleichsweise formal und mathematisch. Insofern ist der Buchtitel eher irreführend.

RC4 (1)

- Die Stromchiffre RC4 wurde bereits 1987 entwickelt, von Ron Rivest und der Firma RSA Security.
- RC4 wurde seinerzeit zwar breitflächig genutzt, der Algorithmus selbst jedoch wurde nicht veröffentlicht.
- 1994 wurde der Algorithmus anonym ins Internet gepostet und in der Folge von zahlreichen Experten analysiert.
- RC4 ist eine der am einfachsten zu beschreibenden Chiffren überhaupt. Gemessen an seiner simplen Struktur hat sich RC4 als erstaunlich robust erwiesen gegenüber kryptoanalytischen Angriffen.
- Zwischenzeitlich sind jedoch mehrere Schwächen bekannt geworden und RC4 wird nicht mehr als Verschlüsselungsalgorithmus empfohlen.
- Dennoch wurde RC4 bis in die jüngste Vergangenheit weiter verwendet und findet sich auch heute noch vereinzelt in Anwendungen.
- RC4 wurde verwendet u.a. in WPA, SSH, SSL, Kerberos, u.v.m.

RC4 (2)

- Die Schlüssellänge ist bei RV4 variabel und liegt zwischen 1 – 256 Bytes (bzw. 8 - 2048 Bit).
- Der State besteht aus einem 256-Byte langen Vektor S = S[0] ... S[255].
- Zu Beginn wird der State initialisiert mittels

  S [0] = 0, S [1] = 1, ..., S [255] = 255.

  Die einzelnen Bytes werden also der Reihe nach gefüllt mit den Binärdarstellungen der Zahlen 0 bis 255.
- In der Folge werden die 256 Bytes des State Vektors immer wieder untereinander permutiert. Zu jedem Zeitpunkt kommt jeder Wert genau einmal vor als eines der Bytes im State Vektor.
- Zur Berechnung der Permutationen von S wird ein zweiter 256 Bytes langer, temporärer Vektor T benötigt: T = T[0] ... T[255].

RC4 (3)

- Der Key K wird einfach bei T[0] beginnend in T kopiert. Falls K aus weniger als 256 Bytes besteht, so wird K einfach so oft hintereinander in T kopiert, bis T gefüllt ist.
- Nun werden die Bytes des State Vektors S wie folgt permutiert:

```
j = 0
for i = 0 to 255 do
      j = (j + S[i] + T[i]) mod 256
      swap (S[i] , S[j])
```

Die initiale Permutation besteht also aus 256 Transpositionen (Transposition = Vertauschung zweier Elemente).

RC4 (4)

Nun wird Byte-weise Key Stream erzeugt. Hierbei wird nach jedem Output erneut eine Permutation auf den Bytes des State Vektors ausgeführt wie folgt:

```
i, j = 0
while GeneratingKeyStream
        i = (i + 1) mod 256
        j = (j + S[i]) mod 256
        swap(S[i] , S[j])
        t = (S[i] + S[j]) mod 256
        output S[t]
```

Der Output wird sodann bitweise XOR-verknüpft mit dem Plaintext.

Reduktion Modulo 256

RC4 ist extrem effizient zu berechnen.

Dies gilt auch für die in RC4 mehrfach enthaltene “Modulo-Berechnung”:

Die Reduktion modulo 256 führt man einfach durch, indem man von der Binärdarstellung der betreffenden Zahl nur die letzten 8 least significant Bits (bzw. das low-order Byte) behält.

Diese Operation entspricht einem bitweisen AND mit dem Wert 255.

RC4 in WEP

- Das über viele Jahre weltweit verbreitete Wireless Encryption Protocol WEP machte Gebrauch von RC4.
- Ein Implementierungsfehler in der WEP Protokollspezifikation, zusammen mit den zwischenzeitlich bekannt gewordenen Schwächen des RC4, führten dazu, dass WEP geknackt werden konnte.
- Details zu den verschiedenen Angriffsmethoden finden sich beispielsweise in Wikipedia.
- Aufgrund der genannten Schwächen wurde WEP abgelöst durch WPA. Dieses wiederum musste bald danach ersetzt werden durch WPA2...

A5/1 (1)

- Im Mobilfunk der 2. Generation (2G) wurde zur Sprachverschlüsselung eine Stromchiffre namens ***A5*** verwendet.
- In unserer Region wurde die Variante ***A5/1*** eingesetzt, in manchen anderen Ländern kam die bewusst abgeschwächte Variante ***A5/2*** zum Einsatz.
- A5/1 setzt sich zusammen aus drei linearen Feedback Shift Registern. Diese haben die Registerlängen 19, 22 und 23 Bit.
- Die Outputs dieser drei LFSR werden per XOR verknüpft und bilden auf diese Weise den Output der Stromchiffre.

A5/1 (2)

- Hierbei werden nicht nach jeder Iteration alle drei LFSR fortbewegt und deren Outputs XOR-verknüpft. Stattdessen erfolgt ein sogenanntes ***Clocking***:
  - in jedem der drei LFSR ist jeweils ein Bit festgelegt als ***Clocking Bit***. Dessen Zustand bestimmt darüber, ob das zugehörige LFSRim aktuellen Schritt weitergeschaltet wird oder nicht.
  - dabei werden zunächst die drei Outputbits der LFSRs betrachtet: je nach dem, ob das Bit Null oder Eins mehrheitlich auftritt, wird dieses Mehrheitsbit gewählt.
  - Dann wird in jedem LFSR nachgeschaut, ob dessen Clocking Bit übereinstimmt mit dem Mehrheitsbit oder nicht. Bei Übereinstimmung wird das LFSR weitergeschaltet. Stimmt das Clocking Bit nicht mit dem Mehrheitsbit überein, so wird der State des zugehörigen LFSR nicht verändert. Es wird also nicht nur kein neues Bit ausgegeben, es wird auch nicht geshiftet.

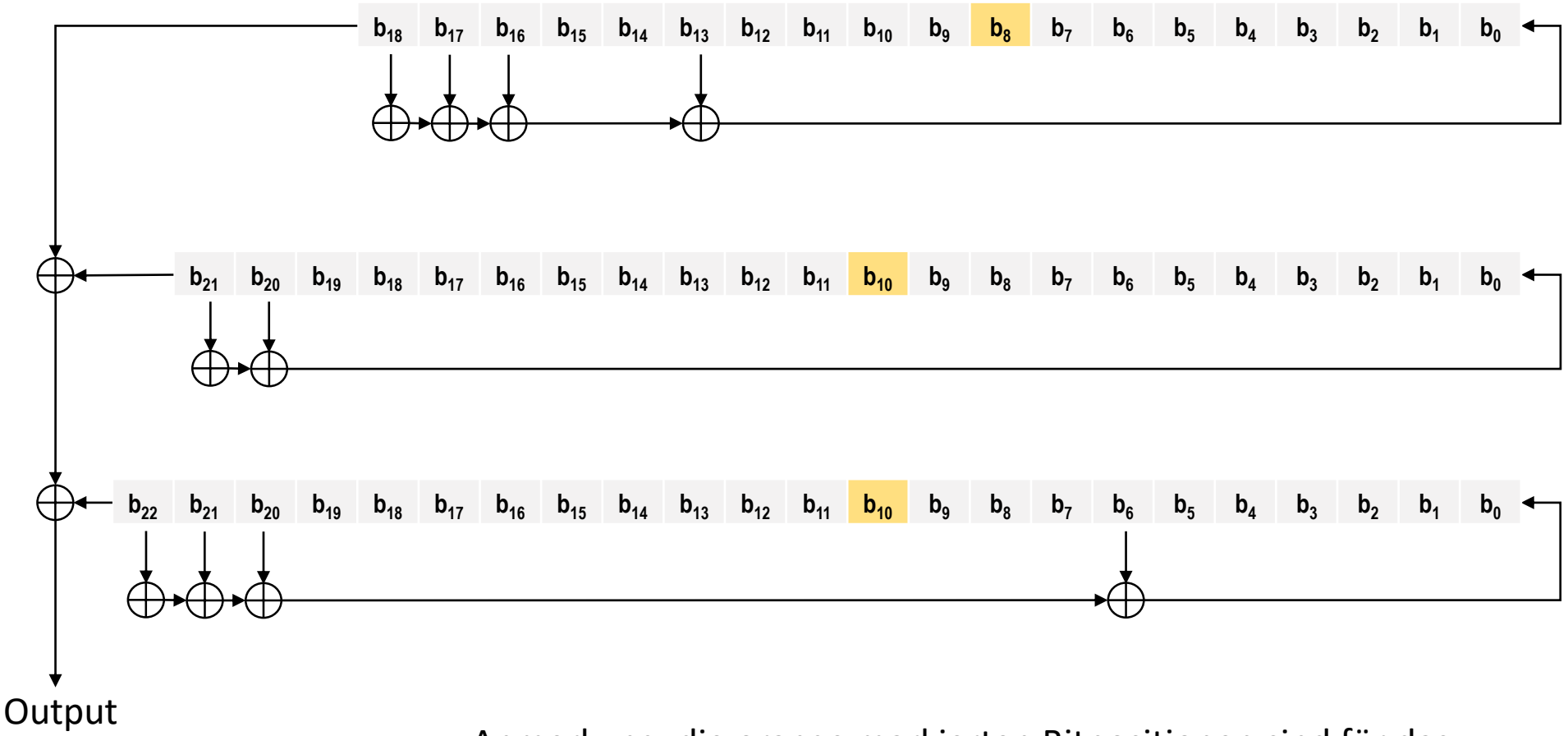
A5/1 (3)

- Durch diese Clocking-Regel wird sichergestellt, dass in jedem Iterationsschritt der Stromchiffre mindestens zwei der drei LFSR weitergeschaltet werden.
- Jedes der drei LFSR wird pro Iterationsschritt mit einer Wahrscheinlichkeit von $\frac{3}{4}$ weitergeschaltet:

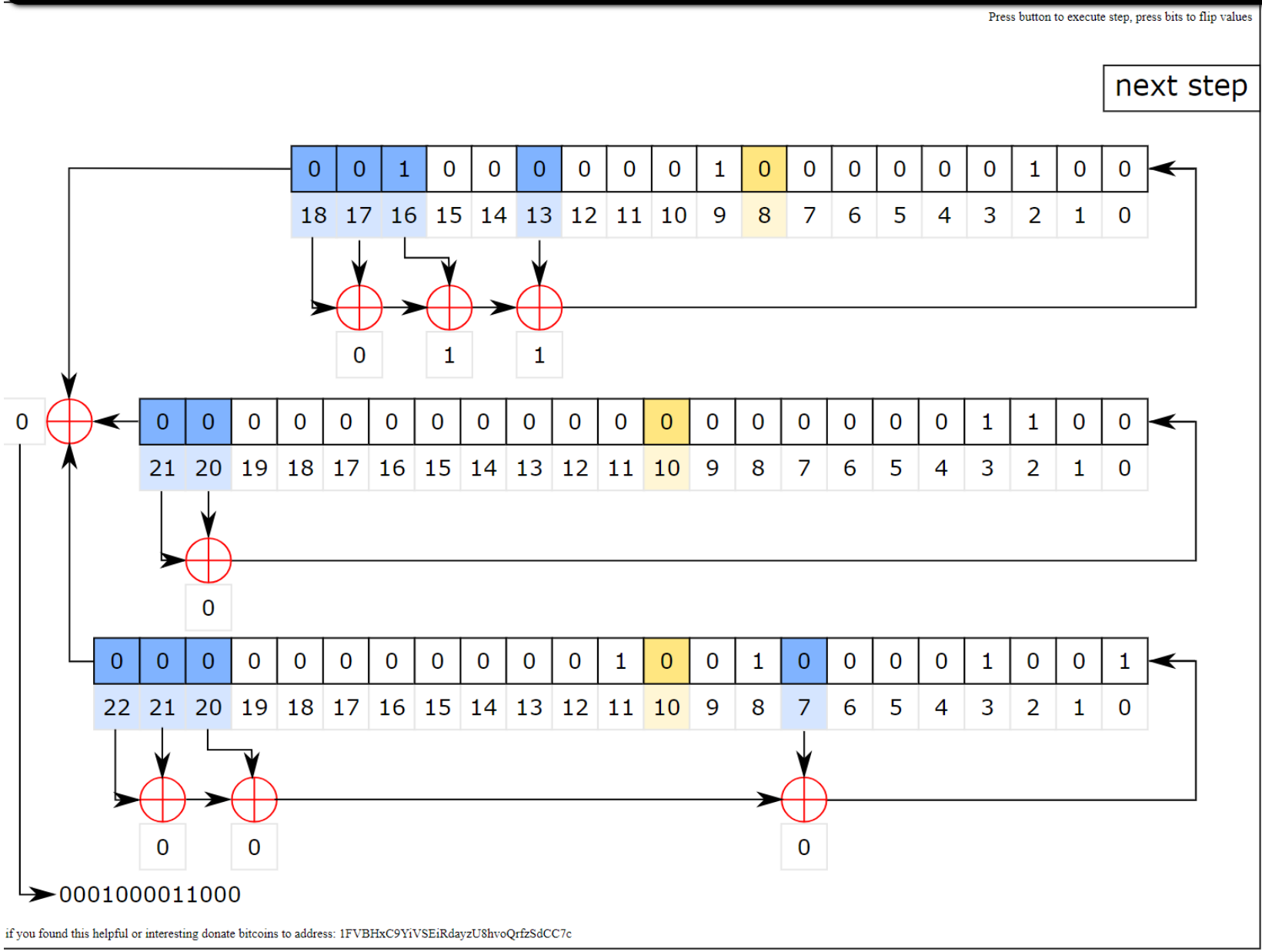
Clocking Bit of LFSR 1	Clocking Bit of LFSR 2	LFSR 3 clocked if its clocking bit is 0 ?	LFSR 3 clocked if its clocking bit is 1 ?
0	0	yes	no
0	1	yes	yes
1	0	yes	yes
1	1	no	yes

- Eine schöne Animation des A5/1 findet sich unter folgendem Link: https://web.archive.org/web/20120326211404/http://l-system.net.pl/crypto/A5_1_stream_cipher.svg

A5/1 (4)



Anmerkung: die orange markierten Bitpositionen sind für das Clocking verantwortlich.



Anmerkung: Diese animierte Darstellung des A5/1 fand sich auf einer Webseite, die nur noch in archive.org erreichbar ist. Die Animation funktioniert aber noch.

Grain und Grain-128a (1)

- Die Stromchiffre Grain war einer der Kandidaten der eStream Competition. Da Grain Schwächen aufwies, haben die Autoren nachgebessert und nach Beendigung von eStream eine modifizierte Version namens Grain-128a veröffentlicht.
- Grain-128a besteht aus der Kombination eines linearen (LFSR) und eines nichtlinearen (NFSR) Feedback Shift Registers.
- Das NFSR hat die Aufgabe, die kryptographische Stärke zu gewährleisten, da lineare Funktionen zu leicht knackbar sind.
- Das LFSR stellt sicher, dass die Periode der Stromchiffre nicht zu kurz ist. Die Konstruktion der Stromchiffre sorgt dafür, dass die Periode der Stromchiffre übereinstimmt mit der Periode des enthaltenen LFSR. Im vorliegenden Fall kann gezeigt werden, dass die Periode maximale Länge hat, und zwar $2^{128} – 1$.

Grain und Grain-128a (2)

- Die Länge des Encryption Keys, sowie die Registerlängen des LFSR und des NFSR betragen jeweils 128 Bit.
- Die Länge der verwendeten Nonce beträgt 96 Bit.
- Zur Initialisierung wird der Encryption Key in das NFSR kopiert, und die Nonce wird in das LFSR kopiert. Die verbleibenden 32 Bit des LFSR werden aufgefüllt mit 31 Einsen und einer Null ganz am Ende.
- Von den 128 Bit des LFSR werden nur 6 Bit als Input für die lineare Feedback Funktion f() verwendet: Bitpositionen 128, 121, 90, 58, 47, 32. (Bem.: diese Bits eines LFSR werden auch bezeichnet als tapped bits). Die korrespondierende Polynomdarstellung liefert ein primitives Polynom und somit eine maximale Periodenlänge.

Grain und Grain-128a (3)

Die nichtlineare Funktion g() hat folgende Form g() =

$b_{32} + b_{72} + b_{102} + b_{128} +$

$+ b_{44}b_{60} + b_{61}b_{125} + b_{63}b_{67} + b_{69}b_{101} + b_{80}b_{88} + b_{110}b_{111} + b_{115}b_{117} +$

$+ b_{46}b_{50}b_{58} + + b_{103}b_{104}b_{106} +$

$+ b_{33}b_{35}b_{36}b_{40}$

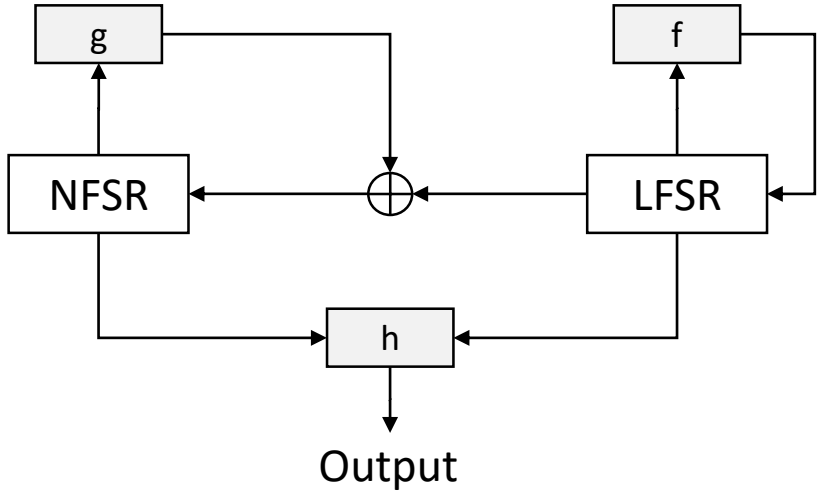
Der Term mit den meisten Variablen ist $b_{33}b_{35}b_{36}b_{40}$, weshalb die Funktion g() den algebraischen Grad 4 hat.

g() wurde so gewählt, dass die Funktion “highly nonlinear” ist und nicht mithilfe linearer Funktionen approximiert werden kann.

Grain und Grain-128a (3)

Vereinfachte Darstellung der Stromchiffre Grain-128a.

h() ist eine weitere, nichtlineare Funktion und bezieht ihren Input aus beiden Feedback Shift Registern.



Salsa20 (1)

- Salsa20 ist eine für Software-Implementierungen ausgelegte Stromchiffre, die 2005 im Rahmen der eStream Competition vorgestellt wurde. Autor ist Dan Bernstein.
- Salsa20 ist lizenzkostenfrei und wird in zahlreichen Anwendungen eingesetzt.
- Salsa20 gehört in die Kategorie der Counter-based Stream Ciphers.
- Salsa20 operiert auf einem Register der Länge 512 Bit.
- Die Schlüssellänge beträgt 256 Bit.
- Hinzu kommen eine Nonce und ein Counter mit je 64 Bit Länge.

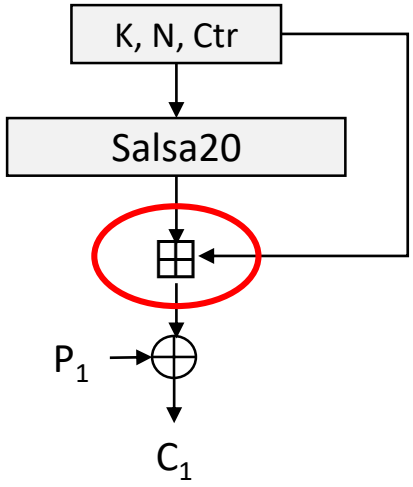
Salsa20 (2)

- Ein Vorteil von Salsa20 besteht darin, dass man an beliebiger Stelle im Schlüsselstrom mit der Entschlüsselung beginnen kann, da keine Abhängigkeit vom vorherigen State besteht.
- Salsa20 ist ein Beispiel einer ARX-Chiffre (siehe Vortragsfolien zu Blockchiffren). Verwendet werden ausschliesslich Addition mod 2^{32}, Rotation, und XOR.
- Ähnlich wie beim AES wird der Inputvektor aufgeteilt in Teilblöcke. Die so entstehenden 16 Teilblöcke mit je 32 Bit werden angeordnet in Form einer 4x4 Matrix.
- Die Operationen werden sodann auf den Teilblöcken (“Words”) der Länge 32 Bit ausgeführt.

Salsa20 (3)

- Ein Designmerkmal von Salsa20 besteht darin, dass der 128 Bit Inputvektor, bestehend aus Key, Nonce und Counter, nicht nur als Input für den Salsa20 Algorithmus dient, sondern danach nochmals auf das Ergebnis aufaddiert wird (siehe Grafik).
- Die Addition $\boxplus$ (nicht mit XOR bzw. $\oplus$ zu verwechseln) der 128 Bit Vektoren erfolgt hierbei, nachdem Teilblöcke (Words) der Länge 32 Bit gebildet wurden. Diese werden dann modulo 2^{32} aufeinander addiert.

- Ohne diese Operation würde der Inputvektor "nur" permutiert und man könnte den Output der Salsa20-Funktion mittels Invertierung wieder auf den Input, bestehend aus Key, Nonce und Counter, zurückrechnen. Daher zerstört man die Invertierbarkeit durch Hinzufügen oben genannter Addition.



Salsa20 (3)

constant	**key**	**key**	**key**
key	**constant**	**nonce**	**nonce**
counter	**counter**	**constant**	**key**
key	**key**	**key**	**constant**

- Zur Initialisierung werden die 512 Bits wie oben dargestellt in 16 Words aufgeteilt und in einer Matrix angeordnet.
- 8 Words werden mit den 128 Bits des Keys gefüllt, 2 Words mit der Nonce und 2 Words mit dem Counter.
- Die verbleibenden 4 Words (in der Diagonale) werden mit konstanten Werten ausgefüllt.

Salsa20 Constants

expa	key	key	key
key	**nd 3**	nonce	nonce
counter	counter	**2-by**	key
key	key	key	**te k**

Die vier Konstanten (Words) mit je 32 Bit innerhalb der 4x4 Matrix von Salsa20 werden bei allen Operationen verwendet und bleiben unverändert. Der Autor des Algorithmus hat nicht irgendwelche Zufallswerte eingesetzt, sondern Werte (auch "***Nothing up my sleeve number***" genannt), deren Inhalt glaubhaft machen soll, dass sie nicht zur Realisierung unerwünschter Mermale wie Backdoors o.ä. dienen. Hierzu wurde der Begriff "expand 32-byte k" auf vier Worte aufgeteilt, wie oben dargestellt.

Salsa20 Column Round

Salsa20 operiert abwechselnd auf Spalten (Columns) oder Zeilen (Rows) der 4x4 Matrix.
Die auszuführende Transformation heisst ***Quarter-Round Function QR*** und bewirkt eine Permutation des Inputs, bestehend aus vier 32-Bit Words.
QR wird also viermal ausgeführt, wahlweise auf vier Spalten oder vier Zeilen.
Die ***Column Round*** operiert auf folgenden Inputs:

$(w_0 , w_4 , w_8 , w_{12})$
$(w_1 , w_5 , w_9 , w_{13})$
$(w_2 , w_6 , w_{10} , w_{14})$
$(w_3 , w_7 , w_{11} , w_{15})$

w_0	w_1	w_2	w_3
w_4	w_5	w_6	w_7
w_8	w_9	w_{10}	w_{11}
w_{12}	w_{13}	w_{14}	w_{15}

Salsa20 Row Round

Die ***Row Round*** operiert, wie der Name suggeriert, auf den Zeilen der 4x4 Matrix. Vor Anwendung der Transformation QR werden die Words der Zeilen jedoch noch um unterschiedlich viele Positionen rotiert:

- die erste Zeile bleibt unverändert
- die zweite Zeile wird um eine Position nach links rotiert
- die dritte Zeile wird um zwei Positionen nach links rotiert
- die vierte Zeile wird um drei Positionen nach links rotiert

Im Ergebnis erhalten wir folgende Zeileninhalte, auf die dann QR angewandt wird:

w_0	w_1	w_2	w_3
w_5	w_6	w_7	w_4
w_{10}	w_{11}	w_8	w_9
w_{15}	w_{12}	w_{13}	w_{14}

Salsa20 Quarter Round

Die ***Quarter-round function QR*** führt auf ihrem aus vier 32-Bit Vektoren bestehenden Input (a, b, c, d) die folgende Transformation durch:

$b = b \oplus [\,(d + a) <<< 7\,]$

$c = c \oplus [\,(a + b) <<< 9\,]$

$d = d \oplus [\,(b + c) <<< 13\,]$

$a = a \oplus [\,(c + d) <<< 18\,]$

Anmerkungen:

+ bezeichnet Addition zweier 32-Bit Vektoren modulo 2^{32}

<<< k bezeichnet Linksrotation eines 32-Bit Vektors um k Positionen

Die vier Zeilen werden der Reihe nach ausgeführt, es werden also die jeweils bereits modifizierten Variableninhalte zugrunde gelegt.

Salsa20 Round

- In Salsa20 wechseln Column Round und Row Round immer ab:
- Als erstes wird die 4x4 Matrix initialisiert.
- Dann beginnt man mit einer Column Round, d.h. Anwendung der Quarter Round auf die vier Spalten der Matrix.
- Anschließend folgt eine Row Round, d.h. Anwendung der Quarter Round auf die vier Zeilen der (wie vorstehend beschrieben modifizierten) Matrix.
- Die Kombination einer Column Round und einer Row Round wird in Salsa20 als ***Double Round*** bezeichnet.
- Insgesamt besteht Salsa20 aus zehn Double Rounds bzw. 20 Column- und Row Rounds. Daher der Name Salsa20.
- Es existieren Varianten, die nur 8 bzw. 12 Rounden ausführen. Diese werden bezeichnet mit Salsa20/8 bzw. Salsa20/12.