

# Erzeugung von Zufallsbits und Stromchiffren

**Dozent:** Prof. Dr. Michael Eichberg

**Basierend auf:** *Cryptography and Network Security - Principles and Practice, 8th Edition, William Stallings*

**Version:** 2.0.1

---

**Folien:** [HTML] <https://delors.github.io/sec-stromchiffre/folien.de.rst.html>


[PDF] <https://delors.github.io/sec-stromchiffre/folien.de.rst.html.pdf>

**Fehler melden:**

<https://github.com/Delors/delors.github.io/issues>

# **1. ERZEUGUNG UND ZUFÄLLIGKEIT VON ZUFALLSZAHLEN**

# Zufallszahlen

- Eine Reihe von Sicherheitsalgorithmen und -protokollen, die auf Kryptographie basieren, verwenden binäre Zufallszahlen:
  - Schlüsselverteilung und reziproke ( *wechselseitige*) Authentifizierungsverfahren
  - Erzeugung von Sitzungsschlüsseln
  - Generierung von Schlüsseln für den RSA Public-Key-Verschlüsselungsalgorithmus
  - Generierung eines Bitstroms für die symmetrische Stromverschlüsselung

Es gibt zwei unterschiedliche Anforderungen an eine Folge von Zufallszahlen:

- Zufälligkeit
- Unvorhersehbarkeit

# Zufälligkeit

- Die Erzeugung einer Folge von angeblich zufälligen Zahlen, die in einem genau definierten statistischen Sinne zufällig sind, war ein Problem.
- Zwei Kriterien werden verwendet, um zu prüfen, ob eine Zahlenfolge zufällig ist:

## **Gleichmäßige Verteilung:**

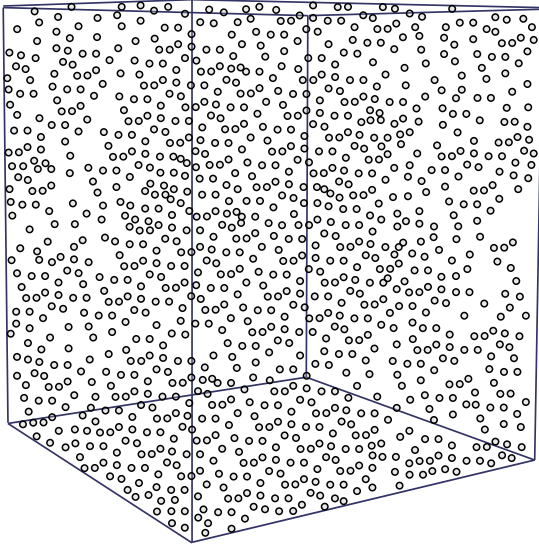
Die Häufigkeit des Auftretens von Einsen und Nullen sollte ungefähr gleich sein.

## **Unabhängigkeit:**

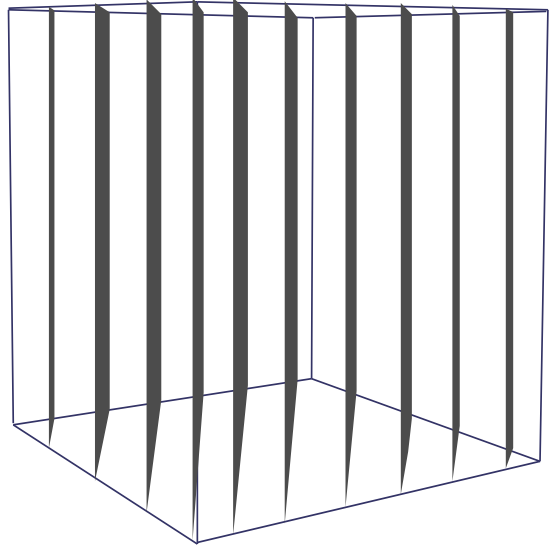
Keine Teilsequenz der Folge kann von den anderen abgeleitet werden.

# Visualisierung von Zufallszahlengeneratoren<sup>[1]</sup>

Erwartete Verteilung von Zufallswerten im 3D-Raum.



Verteilung von „zufälligen“ Werten eines schlechten RNGs im 3D-Raum.



[1] Zufallszahlengenerator  $\hat{=}$   *Random Number Generator (RNG)*

Bei diesem Experiment werden immer drei nacheinander auftretende Werte als Koordinate im 3D-Raum interpretiert. Die erwartete Verteilung ist eine gleichmäßige Verteilung im Raum. Die Verteilung der Werte eines schlechten RNGs ist nicht gleichmäßig und zeigt eine klare Struktur.

# Unvorhersehbarkeit

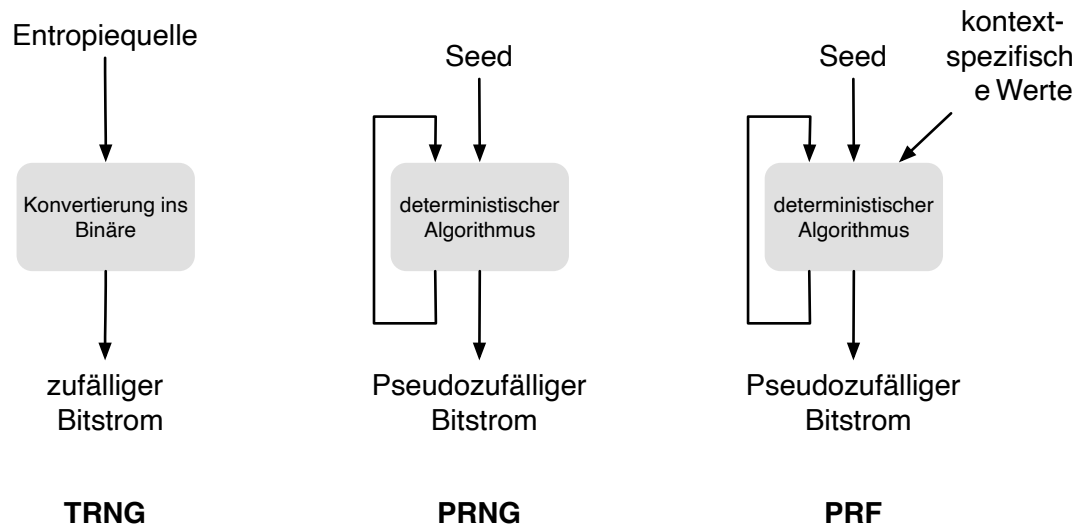
- Die Anforderung ist nicht nur, dass die Zahlenfolge statistisch zufällig ist, sondern auch, dass die *aufeinanderfolgenden Glieder der Folge unvorhersehbar* sind.
- Bei **echten** Zufallsfolgen ist jede Zahl statistisch unabhängig von den anderen Zahlen in der Folge und daher unvorhersehbar.
  - Echte Zufallszahlen(-generatoren) haben Grenzen, insbesondere die Ineffizienz, so dass es häufiger vorkommt, dass Algorithmen implementiert werden, die scheinbar zufällige Zahlenfolgen erzeugen.
  - Es muss darauf geachtet werden, dass ein Gegner nicht in der Lage ist, zukünftige Elemente der Folge auf der Grundlage früherer Elemente vorherzusagen.

# Pseudozufallszahlen

Bei kryptografischen Anwendungen werden in der Regel algorithmische Verfahren zur Erzeugung von Zufallszahlen verwendet.

- Diese Algorithmen sind deterministisch und erzeugen daher Zahlenfolgen, die nicht statistisch zufällig sind.
- Wenn der Algorithmus gut ist, bestehen die resultierenden Sequenzen viele Tests auf Zufälligkeit und werden als Pseudozufallszahlen bezeichnet.

# Zufalls- und Pseudozufallszahlengeneratoren



- TRNG:** Echter Zufallszahlengenerator (🇩🇪 *True Random Number Generator*)  
**PRNG:** Pseudozufallszahlengenerator (🇩🇪 *Pseudorandom Number Generator*)  
**PRF:** Pseudozufällige Funktion (🇩🇪 *Pseudorandom Function*)



# Echter Zufallszahlengenerator (TRNG)

- Nimmt als Eingabe eine Quelle, die effektiv zufällig ist.
- Die Quelle wird als Entropiequelle bezeichnet und stammt aus der physischen Umgebung des Computers:
  - Dazu gehören z. B. Zeitpunkte von Tastenanschlägen, elektrische Aktivität auf der Festplatte, Mausbewegungen und Momentanwerte der Systemuhr.
  - Die Quelle oder eine Kombination von Quellen dient als Eingabe für einen Algorithmus, der eine binäre Zufallsausgabe erzeugt.
- Der TRNG kann einfach die Umwandlung einer analogen Quelle in eine binäre Ausgabe beinhalten.
- Der TRNG kann zusätzliche Verarbeitungsschritte durchführen, um etwaige Verzerrungen in der Quelle auszugleichen.

# Pseudozufallszahlengenerator (PRNG) und Pseudozufallsfunktion (PRF)

## *Pseudozufallszahlengenerator*

- Ein Algorithmus, der zur Erzeugung einer nicht in der Länge beschränkten Bitfolge verwendet wird.
- Die Verwendung eines solchen Bitstroms als Eingabe für eine symmetrische Stromchiffre ist eine häufige Anwendung.

## *Pseudorandom function (PRF)*

- Wird verwendet, um eine pseudozufällige Bitfolge *mit einer bestimmten Länge* zu erzeugen.
- Beispiele sind symmetrische Verschlüsselungsschlüssel und Nonces.

- Nimmt als Eingabe einen festen Wert, den so genannten *Seed*, und erzeugt mithilfe eines deterministischen Algorithmus eine Folge von Ausgabebits.

Häufig wird der Seed von einem TRNG erzeugt.

- Der Ausgangsbitstrom wird ausschließlich durch den oder die Eingabewerte bestimmt, so dass ein Angreifer, der den Algorithmus und den Seed kennt, den gesamten Bitstrom reproduzieren kann.
- Abgesehen von der Anzahl der erzeugten Bits gibt es keinen Unterschied zwischen einem PRNG und einer PRF.

*Nonce (Number used Once)* ist ein Wert, der nur einmal verwendet wird. In der Kryptographie werden Nonces häufig verwendet, um die Sicherheit von Verschlüsselungsalgorithmen zu erhöhen bzw. überhaupt erst zu erhalten.

# PRNG-Anforderungen

- Die grundlegende Anforderung bei der Verwendung eines PRNG oder PRF für eine kryptografische Anwendung ist, dass **ein Gegner, der den Seed nicht kennt, nicht in der Lage ist, die pseudozufällige Zeichenfolge zu bestimmen.**
- Die Forderung nach Geheimhaltung der Ausgabe eines PRNG oder PRF führt zu spezifischen Anforderungen in den Bereichen:
  - Zufälligkeit
  - Unvorhersehbarkeit
  - Merkmale des Seeds

# Zufälligkeit

- Der erzeugte Bitstrom muss zufällig erscheinen, obwohl er deterministisch ist:
  - Es gibt keinen einzigen Test, mit dem festgestellt werden kann, ob ein PRNG Zahlen erzeugt, die die Eigenschaft der Zufälligkeit aufweisen
  - Wenn der PRNG auf der Grundlage mehrerer Tests Zufälligkeit aufweist, kann davon ausgegangen werden, dass er die Anforderung der Zufälligkeit erfüllt.

NIST SP 800-22 legt fest, dass die Tests auf drei Merkmale ausgerichtet sein sollten:

1. gleichmäßige Verteilung,
2. Skalierbarkeit,
3. Konsistenz

# Tests auf Zufälligkeit

SP 800-22 listet 15 verschiedene Zufallstests auf (Auszug):

## Häufigkeitstest:

- Der grundlegendste Test, der in jeder Testreihe enthalten sein muss.
- Es soll festgestellt werden, ob die Anzahl der Einsen und Nullen in einer Sequenz annähernd derjenigen entspricht, die bei einer echten Zufallssequenz zu erwarten wäre.

## Lauf längentest:

- Schwerpunkt dieses Tests ist die Zahl der Läufe (🚩 *runs*) in der Folge, wobei ein Lauf (🚩 *run*) eine ununterbrochene Folge identischer Bits ist, die vorher und nachher durch ein Bit des entgegengesetzten Werts begrenzt wird.
- Es soll festgestellt werden, ob die Anzahl der Läufe von Einsen und Nullen verschiedener Länge den Erwartungen für eine Zufallsfolge entspricht.

## Maurers universeller statistischer Test:

- Fokus ist die Anzahl der Bits zwischen übereinstimmenden Mustern.
- Ziel ist es, festzustellen, ob die Sequenz ohne Informationsverlust erheblich komprimiert werden kann oder nicht. Eine signifikant komprimierbare Sequenz wird als nicht zufällig betrachtet.

# Unvorhersehbarkeit

Ein Strom von Pseudozufallszahlen sollte zwei Formen der Unvorhersehbarkeit aufweisen:

## 1. Vorwärtsgerichtete Unvorhersehbarkeit

Wenn der Seed unbekannt ist, sollte das nächste erzeugte Bit in der Sequenz trotz Kenntnis der vorherigen Bits in der Sequenz unvorhersehbar sein.

## 2. Rückwärtsgerichtete Unvorhersehbarkeit

- Es sollte nicht möglich sein, den Seed aus der Kenntnis der erzeugten Werte zu bestimmen.
- Es sollte keine Korrelation zwischen einem Seed und einem aus diesem Seed generierten Wert erkennbar sein.
- Jedes Element der Sequenz sollte wie das Ergebnis eines unabhängigen Zufallsereignisses erscheinen, dessen Wahrscheinlichkeit  $1/2$  ist.

1

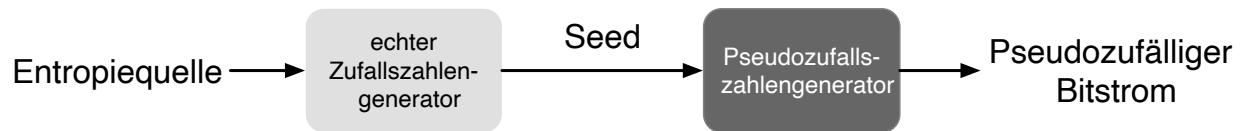
## Hinweis

Dieselbe Reihe von Tests für die Zufälligkeit liefert auch einen Test für die Unvorhersehbarkeit: Eine Zufallsfolge hat keine Korrelation mit einem festen Wert (dem Seed).

2

# Anforderungen an den Seed

- Der Seed, der als Eingabe für den PRNG dient, muss sicher und unvorhersehbar sein
- Der Seed selbst muss eine Zufalls- oder Pseudozufallszahl sein.
- Normalerweise wird der Seed von einem TRNG erzeugt.



# Algorithmus-Entwurf

Algorithmen lassen sich in zwei Kategorien einteilen:

1. Speziell entwickelte Verfahren.

Algorithmen, die speziell und ausschließlich für die Erzeugung pseudozufälliger Bitströme entwickelt wurden.

2. Algorithmen, die auf bestehenden kryptographischen Algorithmen basieren.

Sie bewirken eine Zufallsverteilung der Eingabedaten.

Kryptografische Algorithmen aus den folgenden drei Kategorien werden üblicherweise zur Erstellung von PRNGs verwendet:

- Symmetrische Blockchiffren
- Asymmetrische Verschlüsselungsalgorithmen
- Hash-Funktionen und Nachrichtenauthentifizierungscodes



# Lineare Kongruenzgeneratoren

Ein erstmals von Lehmer vorgeschlagener Algorithmus, der mit vier Zahlen parametrisiert ist:

$m$	der Modul	$m > 0$
$a$	der Multiplikator	$0 < a < m$
$c$	das Inkrement	$0 \leq c < m$
$X_0$	der Startwert, oder <i>Seed</i>	$0 \leq X_0 < m$

Die Folge von Zufallszahlen  $\{X_n\}$  erhält man durch die folgende iterative Gleichung:  
$$X_{n+1} = (aX_n + c) \bmod m$$

Wenn  $m$ ,  $a$ ,  $c$  und  $X_0$  ganze Zahlen sind, dann erzeugt diese Technik eine Folge von ganzen Zahlen, wobei jede ganze Zahl im Bereich  $0 \leq X_n < m$  liegt.

Die Auswahl der Werte für  $a$ ,  $c$  und  $m$  ist entscheidend für die Entwicklung eines brauchbaren Zufallszahlengenerators.

17

## Warnung

Lineare Kongruenzgeneratoren sind einfach zu implementieren und erfordern nur wenig Speicherplatz. Sie sind jedoch nicht für kryptografische Anwendungen geeignet, da sie eine viel zu kurze Periode haben und leicht zu brechen sind.

Im Bereich der Simulation können sie jedoch nützlich sein.

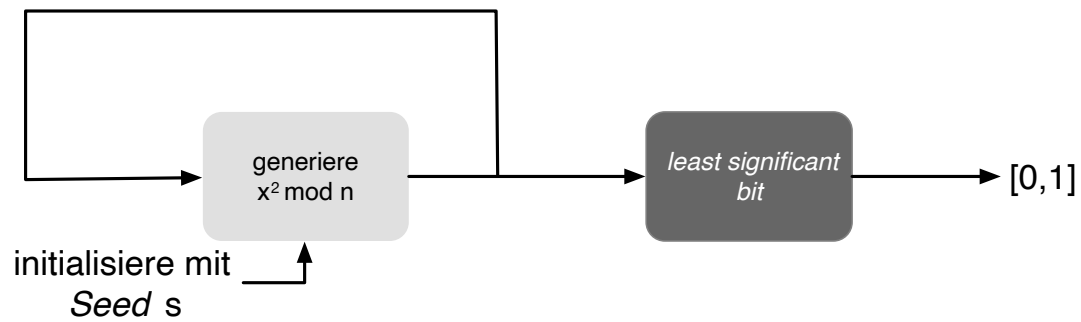
# Blum Blum Shub (BBS) Generator

- Hat vermutlich den stärksten öffentlichen Beweis für seine kryptografische Stärke von allen speziell entwickelten Algorithmen.
- Er wird als *kryptographisch sicherer Pseudozufallsbitgenerator (CSPRBG)* bezeichnet.

Ein CSPRBG ist definiert als ein Algorithmus, der den Next-Bit-Test besteht, wenn es keinen Polynomialzeit-Algorithmus gibt, der bei Eingabe der ersten  $k$  Bits einer Ausgabesequenz das  $(k + 1)$ -te Bit mit einer Wahrscheinlichkeit deutlich größer als  $1/2$  vorhersagen kann.

- Die Sicherheit von BBS beruht auf der Schwierigkeit der Faktorisierung von  $n$ .

# Blum Blum Shub Block Diagram



$n$  ist das Produkt von zwei (sehr großen) Primzahlen  $p$  und  $q$ :  $n = p \times q$ . Weiterhin muss gelten:  $p \equiv q \equiv 3 \pmod{4}$ .

Der Seed  $s$  sollte eine ganze Zahl sein, die zu  $n$  *coprime* ist (d. h.  $p$  und  $q$  sind keine Faktoren von  $s$ ) und nicht 1 oder 0.

## Beispiel - Blum Blum Shub (BBS) Generator

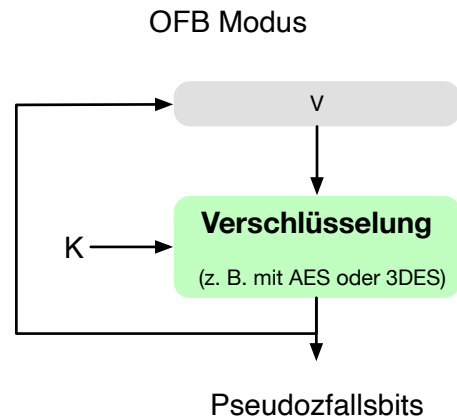
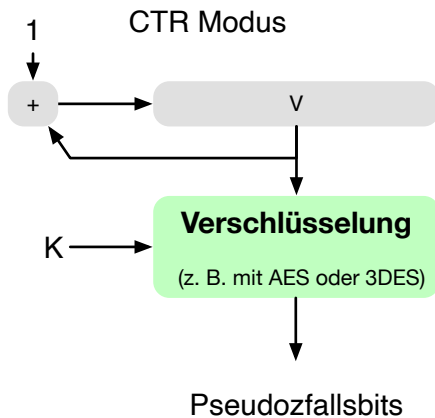
Sei  $p = 383$  und  $q = 503$ , dann ist  $n = 192649$ . Weiterhin sei der Seed  $s = 101355$ .

$i$	$x_i$	$B_i$
0	$101355^2 \bmod 192649 = 20749$	
1	143135	1
2	177671	1
3	97048	0
4	89992	0
5	174051	1
6	80649	1
7	45663	1
8	69442	0
9	186894	0
10	177046	0

# PRNG mit Hilfe der Betriebsmodi für Blockchiffren

Zwei Ansätze, die eine Blockchiffre zum Aufbau eines PRNG verwenden, haben weitgehend Akzeptanz erhalten:

- CTR Modus: Empfohlen in NIST SP 800-90, ANSI standard X.82, und RFC 4086
- OFB Modus: Empfohlen in X9.82 und RFC 4086



Die initialen Werte für V und K basieren auf dem Seed, der von einem TRNG erzeugt wird/erzeugt werden sollte. Zum Beispiel kann von einem 256-Bit-Zufallswert die ersten 128 Bit für V und die nächsten 128 Bit für K verwendet werden.

Die Blockchiffre wird verwendet, um den Seed zu verschlüsseln und den Schlüsselstrom zu erzeugen. Im CTR Mode wird der initiale Wert für V inkrementiert.

Gründe für die Verwendung von Blockchiffren ist die Einfachheit der Implementierung und die Tatsache, dass Blockchiffren bereits in vielen Anwendungen vorhanden sind und die kryptografischen Eigenschaften von Blockchiffren gut verstanden sind.

## Test auf Zufälligkeit

Test auf Zufälligkeit: Gegeben sei eine Bitfolge, die von einem RNG erzeugt wurde. Was ist das erwartete Ergebnis, wenn man gängige Komprimierungsprogramme (z. B. 7zip, gzip, rar, ...) verwendet, um die Datei zu komprimieren; d. h. welchen Kompressionsgrad erwarten Sie?

## Test auf Zufälligkeit

Test auf Zufälligkeit: Gegeben sei eine Bitfolge, die von einem RNG erzeugt wurde. Was ist das erwartete Ergebnis, wenn man gängige Komprimierungsprogramme (z. B. 7zip, gzip, rar, ...) verwendet, um die Datei zu komprimieren; d. h. welchen Kompressionsgrad erwarten Sie?

## Lineare Kongruenzgeneratoren

Implementiere einen linearen Kongruenzgenerator, um zu untersuchen, wie er sich verhält, wenn sich die Zahlenwerte von  $a$ ,  $c$  und  $m$  ändern. Versuchen Sie Werte zu finden, die eine vermeintlich zufällige Folge ergeben.

Testen Sie Ihren Zufallszahlengenerator unter anderem mit den folgenden Werten:

```
lcg(seed,a,c,m,number_of_random_values_to_generate)
```

```
lcg(1234,8,8,4096,100)
```

```
lcg(1234,4,8,4096,100)
```



## Lineare Kongruenzgeneratoren

Implementiere einen linearen Kongruenzgenerator, um zu untersuchen, wie er sich verhält, wenn sich die Zahlenwerte von  $a$ ,  $c$  und  $m$  ändern. Versuchen Sie Werte zu finden, die eine vermeintlich zufällige Folge ergeben.

Testen Sie Ihren Zufallszahlengenerator unter anderem mit den folgenden Werten:

```
lcg(seed, a, c, m, number_of_random_values_to_generate)
lcg(1234, 8, 8, 4096, 100)
lcg(1234, 4, 8, 4096, 100)
```

## Blum Blum Shub

Sei  $p = 83$  und  $q = 47$ .

Berechnen Sie die ersten 8 Bits der Folge, die von einem Blum Blum Shub Generator erzeugt wird, wenn der Seed  $s = 253$  ist. Nutzen Sie einen Taschenrechner oder schreiben Sie einfach ein Script in einer Sprache Ihrer Wahl.

## Blum Blum Shub

Sei  $p = 83$  und  $q = 47$ .

Berechnen Sie die ersten 8 Bits der Folge, die von einem Blum Blum Shub Generator erzeugt wird, wenn der Seed  $s = 253$  ist. Nutzen Sie einen Taschenrechner oder schreiben Sie einfach ein Script in einer Sprache Ihrer Wahl.

# Quellen der Entropie

- Ein echter Zufallszahlengenerator (TRNG) verwendet eine nicht-deterministische Quelle zur Erzeugung von Zufälligkeit.
- Die meisten funktionieren durch Messung unvorhersehbarer natürlicher Prozesse, wie z. B. Impulsdetektoren für ionisierende Strahlung, Gasentladungsröhren und undichte Kondensatoren.
- Intel hat einen kommerziell erhältlichen Chip entwickelt, der das thermische Rauschen durch Verstärkung der an nicht angesteuerten Widerständen gemessenen Spannung erfasst.

# Vergleich von PRNGs und TRNGs

	<b>Pseudozufallszahlengeneratoren</b>	<b>echte Zufallszahlengeneratoren</b>
<b>Effizienz</b>	sehr effizient	im Allgemeinen ineffizient
<b>Determinismus</b>	deterministisch	nicht Deterministisch
<b>Periodizität</b>	periodisch	aperiodisch

# Konditionierung

Ein TRNG kann eine Ausgabe erzeugen, die in irgendeiner Weise verzerrt ist (z. B. gibt es mehr Einsen als Nullen oder umgekehrt)

**Verzerrt:** NIST SP 800-90B definiert einen Zufallsprozess als verzerrt in Bezug auf einen angenommenen diskreten Satz möglicher Ergebnisse, wenn einige dieser Ergebnisse eine größere Wahrscheinlichkeit des Auftretens haben als andere.

## Entropierate:

NIST 800-90B definiert die Entropierate als die Rate, mit der eine digitalisierte Rauschquelle Entropie liefert.

- Ist ein Maß für die Zufälligkeit oder Unvorhersehbarkeit einer Bitfolge.
- Ein Wert zwischen 0 (keine Entropie) und 1 (volle Entropie).

1

## *Konditionierungsalgorithmen/Entzerrungsalgorithmen:*

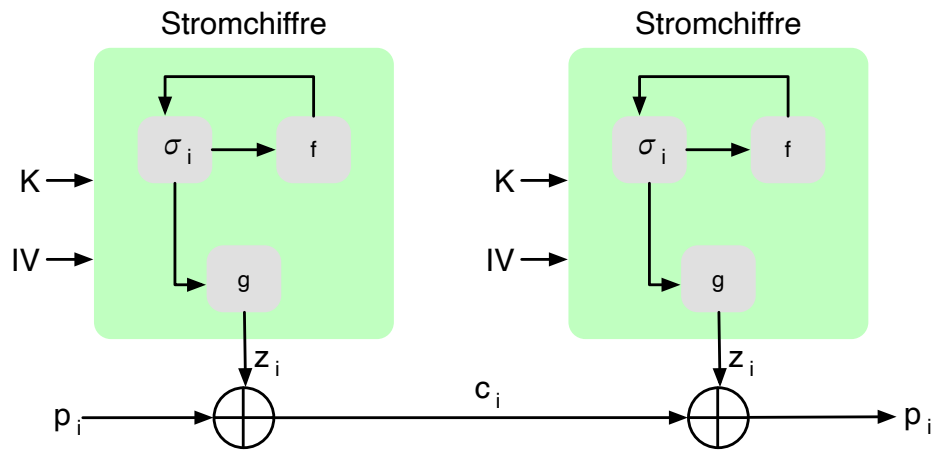
Verfahren zur Modifizierung eines Bitstroms zur weiteren Randomisierung der Bits.

- Die Konditionierung erfolgt in der Regel durch die Verwendung eines kryptografischen Algorithmus zur Verschlüsselung der Zufallsbits, um Verzerrungen zu vermeiden und die Entropie zu erhöhen.
- Die beiden gängigsten Ansätze sind die Verwendung einer Hash-Funktion oder einer symmetrischen Blockchiffre.

2

## 2. STROMCHIFFREN

# Allgemeine Struktur einer typischen Stromchiffre



Klartext  $p_i$

Chiffretext  $c_i$

Schlüsselstrom  $z_i$

Schlüssel  $K$

Initialisierungswert  $IV$

Zustand  $\sigma_i$

Funktion zur Berechnung des  
nächsten Zustands  $f$

Schlüsselstromfunktion  $g$



# Überlegungen zum Entwurf von Stromchiffren

## **Die Verschlüsselungssequenz sollte eine große Periode haben:**

Ein Pseudozufallszahlengenerator verwendet eine Funktion, die einen deterministischen Strom von Bits erzeugt, der sich schließlich wiederholt; je länger die Wiederholungsperiode, desto schwieriger wird die Kryptoanalyse.

## **Der Schlüsselstrom sollte die Eigenschaften eines echten Zufallszahlenstroms so gut wie möglich nachbilden:**

Es sollte eine ungefähr gleiche Anzahl von 1en und 0en geben.

Wenn der Schlüsselstrom als ein Strom von Bytes behandelt wird, sollten alle 256 möglichen Byte-Werte ungefähr gleich oft vorkommen.

## **Eine Schlüssellänge von mindestens 128 Bit ist wünschenswert:**

Die Ausgabe des Pseudo-Zufallszahlengenerators ist vom Wert des Eingabeschlüssels abhängig.

Es gelten die gleichen Überlegungen wie für Blockchiffren.

## **Mit einem richtig konzipierten Pseudozufallszahlengenerator kann eine Stromchiffre genauso sicher sein wie eine Blockchiffre mit vergleichbarer Schlüssellänge:**

Ein potenzieller Vorteil ist, dass Stromchiffren, die keine Blockchiffren als Baustein verwenden, in der Regel schneller sind und weit weniger Code benötigen als Blockchiffren.

# RC 4

- 1987 von Ron Rivest für RSA Security entwickelt.
- Stromchiffre mit variabler Schlüsselgröße und byteorientierten Operationen, die in Software sehr schnell ausgeführt werden können.
- Basiert auf der Verwendung einer zufälligen Permutation.

## Warnung

In der RC 4-Schlüsselableitungsfunktion wurde eine grundlegende Schwachstelle aufgedeckt, die den Aufwand für die Ermittlung des Schlüssels verringert.

Es wurde gezeigt, dass es möglich ist *wiederholt* verschlüsselte Klartexte wiederherzustellen.

Aufgrund der Schwachstellen hat die IETF RFC 7465 herausgegeben, der die Verwendung von RC4 in TLS verbietet. In seinen jüngsten TLS-Richtlinien verbietet das NIST ebenfalls die Verwendung von RC4 für Regierungszwecke.

# ChaCha20


- ChaCha20 ist eine Stromchiffre, die von Daniel J. Bernstein entwickelt wurde.
- ChaCha20 ist ein schneller Verschlüsselungsalgorithmus (ohne besondere Hardwareanforderungen).

Reine Softwareimplementierungen von ChaCha20 sind reinen Softwareimplementierungen von AES in Bezug auf die Geschwindigkeit überlegen.

- ChaCha20 ist im **RFC 8439** spezifiziert.
- ChaCha20 ist eine spezielle Form von ChaCha, die 20 Runden (oder *80 Quarter Rounds*) durchläuft.

## ChaCha20 Zustand - Matrix - Indizierung

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Die 16 Werte der Matrix sind vorzeichenlose 32-Bit Ganzzahlen ( *Integers*).

# ChaCha Quarter Round

Grundlegende Operation in ChaCha20.

1. `a += b; d ^= a; d <<<= 16;`
2. `c += d; b ^= c; b <<<= 12;`
3. `a += b; d ^= a; d <<<= 8;`
4. `c += d; b ^= c; b <<<= 7;`

`+`: ist die Addition modulo  $2^{32}$ .  
`^`: ist die XOR-Operation.  
`<<<`: ist die zyklische Linksverschiebung um die angegebene Anzahl von Stellen.

Gegeben seien die folgenden Werte:

```
a = 0x11111111
b = 0x01020304
c = 0x77777777
d = 0x01234567
```

Anwendung der vierten Formel:

```
c = c + d = 0x77777777 + 0x01234567
           = 0x789abcde

b = b ^ c = 0x01020304 ^ 0x789abcde
           = 0x7998bfda

b = b <<< 7 = 0x7998bfda <<< 7
           = 0xcc5fed3c
```

34

ChaCha20-Poly1305 - d. h. ChaCha20 mit zusätzlicher Authentifizierung (Poly 1305) - wird unter anderem von IPsec, SSH, TLS 1.2, DTLS 1.2, TLS 1.3, WireGuard, S/MIME 4.0, und OTRv4[22].

# Anwendung der *Quarter Round Operation*

- Die *Quarter Round Operation* operiert immer auf **vier der sechzehn Werte** des Zustands.
- **QUARTERROUND(x, y, z, w)** operiert auf den vier Werten identifiziert durch die Indizes: x, y, z und w.

1

Beispiel - *Column Round*:

Die **QUARTERROUND(1, 5, 9, 13)** operiert somit auf den Werten **a, b, c, d** der Matrix.

0	<b>a</b>	2	3
4	<b>b</b>	6	7
8	<b>c</b>	10	11
12	<b>d</b>	14	15

2

Beispiel - *Diagonal Round*:

Gegeben seien die Werte:

879531e0	c5ecf37d	516461b1	c9a62f8a
44c20ef3	3390af7f	d9fc690b	2a5f714c
53372767	b00a5631	974c541a	359e9963
5c971061	3d631689	2098d9d6	91dbd320

Ergebnis der **QUARTERROUND(2, 7, 8, 13)**:

879531e0	c5ecf37d	<b>bdb886dc</b>	c9a62f8a
44c20ef3	3390af7f	d9fc690b	<b>cfacafd2</b>
<b>e46bea80</b>	b00a5631	974c541a	359e9963
5c971061	<b>ccc07c79</b>	2098d9d6	91dbd320

3

35

# Die ChaCha20 Blockfunktion

Eingaben:

- Ein 256-Bit-Schlüssel ( $8 \times 32$ -Bit-Werte (little-endian))
- Eine 96-Bit-Nonce ( $3 \times 32$ -Bit-Werte (little-endian))
- Ein 32-Bit-Blockzähler

Ausgabe:

- 64 Byte (512 Bit) des Schlüsselstroms

1

Initialer Zustand:

**Werte 0-3:**

0x61707865, 0x3320646e, 0x79622d32, 0x6b206574

**Werte 4-11:**

Der 256-Bit Schlüssel ( $8 \times 32$ Bit)

**Wert 12:** der Blockzähler

Da ein Block 64 Byte lang ist, können max 256GiB Daten verschlüsselt werden.

**Wert 13-15:**

Die 96-Bit Nonce

2

Struktur des initialen Zustands:

```
cccccccc cccccccc cccccccc cccccccc
kkkkkkkk kkkkkkkk kkkkkkkk kkkkkkkk
kkkkkkkk kkkkkkkk kkkkkkkk kkkkkkkk
bbbbbbbb nnnnnnnn nnnnnnnn nnnnnnnn
```

**c:** Konstante  
**k:** Schlüssel  
**b:** Blockzähler  
**n:** Nonce

3

Es werden 20 Runden durchlaufen, wobei in jeder Runde vier *Quarter Rounds* ausgeführt werden.

```
// Column rounds: 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 Runde
{
    QUARTERROUND(0, 4, 8, 12)
    QUARTERROUND(1, 5, 9, 13)
    QUARTERROUND(2, 6, 10, 14)
    QUARTERROUND(3, 7, 11, 15)
}
// Diagonal rounds: 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 Runde
{
    QUARTERROUND(0, 5, 10, 15)
    QUARTERROUND(1, 6, 11, 12)
    QUARTERROUND(2, 7, 8, 13)
    QUARTERROUND(3, 4, 9, 14)
}
```

4

Nach den 20 Runden wird der Zustand mit dem initialen Zustand addiert. Auf diese Weise erhält man den Schlüsselstrom.

Dieser wird dann zum Verschlüsseln des Klartexts mittels XOR verwendet. Somit muss der Klartext somit kein vielfaches der Blockgröße sein.

5

36

Es gibt auch eine Variante von ChaCha20, die einen 64-Bit-Blockzähler und eine 64-Bit Nonce verwendet. Hier wird jedoch die IETF Variante diskutiert.

## Little-endian

Bei der Verwendung von *little-endian* (wörtlich etwa: „kleinendigen“) Format wird das niedrigstwertige Byte an der Anfangsadresse gespeichert.

D. h. die 32-Bit-Zahl 1 wird als:

(kleinste Adresse) 0xXX	0xXX+1	0xXX+2	(größte Adresse) 0xXX+3
01	00	00	00

gespeichert.