KRYPTOLOGIE UND SYSTEMSICHERHEIT

Sommersemester 2020

Technische Hochschule Mittelhessen

Andre Rein

– Moderne Kryptographie: Stromchiffren –

MODERNE KRYPTOGRAPHIE: SYMMETRISCHE KRYPTOGRAPHIE

– Stromchiffren –

ÜBERLEGUNG ZUR VIGENÈRE-CHIFFRE

Wie wir bei der Vigenère-Chiffre gesehen haben, war es bei der Kryptoanalyse notwendig Kombinationen der Elemente des Schlüsselalphabets (hier: $\{A, B, C, \ldots, Z\}$ bzw. $\mathbb{Z}_{26}$) je nach Länge des Schlüssel auszuprobieren um dann eine Häufigkeitsanalyse auszuführen, die den statistisch besten entschlüsselten Klartext bestimmt. Ein Lösung war es z.B. auf Basis von Bigrammen zu arbeiten.

✍ Machen Sie nun folgende Überlegung: Was würde passieren, wenn der gewählte Schlüssel genau so lang wie die zu verschlüsselnde Nachricht wäre?

Ist der Schlüssel (k) genau so lang wie der zu verschlüsselnde Text (m), wobei gilt $|m| = |k|$, dann existieren $26^{|k|}$ mögliche Kombinationen.

Gegeben sei $|k| = 2 \rightarrow 26^2 = 626$, d.h.
$\{(A, A), (A, B), \ldots (A, Z), (B, A), (B, B), \ldots (B, Z), \ldots (Z, Z)\}$.

|k|	$26^{\|k\|}$	**Kombinationen**
2	26^2	626
3	26^3	17576
4	26^4	456976
5	26^5	11881376
10	26^{10}	$1,411670957 \times 10^{14}$
100	26^{100}	$3,142930642 \times 10^{141}$

Das ist auch ein Grund, warum ein Bruteforce-Angriff, der alle Kombinationen ausprobiert und dann eine Häufigkeitsanalyse ausführt, alleine gegen Vigenère nur bis zu einer gewissen Schlüssellänge verwendet werden kann.

ONE TIME PAD

Das sog. **One Time Pad**, verallgemeinert nun das Prinzip der Vigenère-Chiffre wobei die Schlüssellänge mindestens gleich der Länge des zu verschlüsselnden Textes entspricht!

ONE TIME PAD

Definiert sei eine Chiffre für (K, M, C) als ein paar "effizienter" Algorithmen (E, D) für die gilt:

$$\forall m \in M, c \in C, k \in K : D_k(E_k(m)) = m$$

- One Time Pad (OTP, 1917 Vernam):

$$M = C = K = \{0,1\}^n$$
$$E_k(m) = k \oplus m, D_k(c) = k \oplus c$$

ONE TIME PAD: EIGENSCHAFTEN

- Sehr schnelle Verschlüsselung und Entschlüsselung
- Bietet sog. **perfekte Sicherheit**, d.h.:
  - Ein Ciphertext erlaubt **keinerlei** Rückschlüsse auf den Klartext, außer dessen Länge
  - Es besteht keinerlei statistische Abhängigkeit zwischen Ciphertext und Klartext
  - Egal wie viel Rechenleistung zur Verfügung steht → Das erraten des Klartextes anhand des Ciphertext ist nicht möglich
- Pro Bit beträgt die Wahrscheinlichkeit richtig oder falsch zu liegen **genau** 50%
  - **Somit ist das Problem nicht entscheidbar!**

ONE TIME PAD: EIGENSCHAFTEN

!!! ABER !!!

- Perfekte Sicherheit kann nur erreicht werden:
  - wenn der Schlüssel **tatsächlich zufällig** (truely random) ist
  - wenn gilt $|K| \geq |M|$ → Schlüssellänge $\geq$ Nachricht
- Daraus folgt das die Länge des Schlüssels mindestens so lang sein muss wie die Nachricht (Klartext) selbst
  - Klartext, Schlüssel und Ciphertext haben also die gleiche Länge

	Wegen der Längenanforderung ist die Verwendung des One Time Pads in der Praxis nicht umsetzbar!

ONE TIME PAD: BEISPIEL

Wir verwenden hier als Operation die modulare Addition in $\mathbb{Z}_{26}$, anstelle von XOR ($\oplus$), ansonsten wären die Ciphertexte nicht vernünftig darstellbar.

```
ONETIMPADICHBINEINPLAINTEXTONETIMEPAD <- m
      (modulare Addition Z26)
ANKASJHPQLKJASLKDNMNRMNCAJHGIUOYSDOUW <- k
------------------------------------
OAOTAVWPTTMQBAYOLABYRUAVEGAUVYHGEHDUZ <- c
      (modulare Addition Z26)
ANKASJHPQLKJASLKDNMNRMNCAJHGIUOYSDOUW <- k
------------------------------------
ONETIMPADICHBINEINPLAINTEXTONETIMEPAD <- m
```

Der Schlüssel k hier ist nicht (truely random), warum?

ONE TIME PAD (AUFGABE 5.1)✍️

AUFGABE: VERSCHLÜSSELN EINER NACHRICHT

Die Nachricht `attack at dawn` ist mit einem One Time Pad verschlüsselt worden. Der Ciphertext lautet: `09e1c5f70a65ac519458e7e53f36` (in hex). Wie lautet der ciphertext der Nachricht `attack at dusk` unter dem gleichen Schlüssel?

	Hier bitte `XOR` verwenden!

⏱️ Bearbeitung: 15 Minuten → Dann besprechen

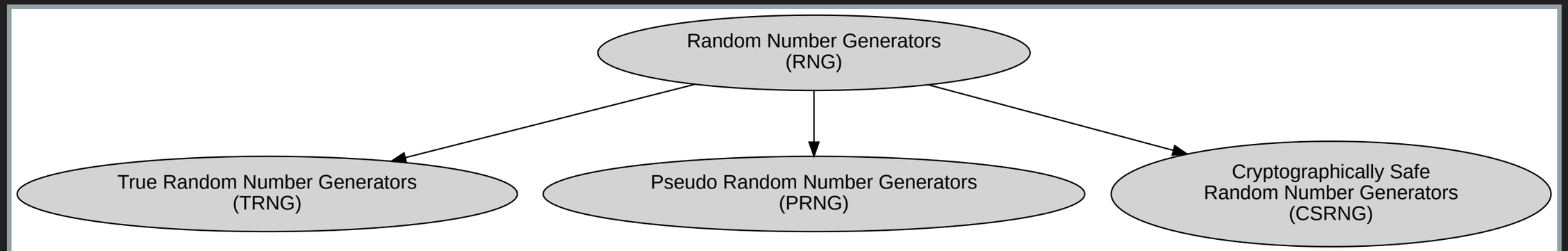
ONE TIME PAD

- Die Grundidee des One Time Pads ist eigentlich ganz gut
- Das einzigen beiden Probleme in der Praxis sind:
  - Sehr **lange** und **zufällige** Schlüssel zu generieren

Kennen Sie ein Verfahren um so einen zufälligen und langen Bitstrom zu erzeugen?

PSEUDO RANDOM NUMBER GENERATORS UND STROMCHIFFREN

RANDOM NUMBER GENERATORS (RNG)



TRUE RANDOM NUMBER GENERATORS (TRNG)

- Echte Zufallszahlengeneratoren (TRNG) basieren auf physikalischen Zufallsprozessen:
  - Münzwurf, Würfeln, radioaktiver Zerfall, Leitungsrauschen
- Das Ausgabesignal s_i des TRNG sollte "gute" Statistische Eigenschaften haben
  - $Pr(s_i = 0) = Pr(s_i = 1) = 50$ (Pr = Probability)
  - Das kann durch Nachbearbeitung des Ausgabesignals erreicht werden
- Die Ausgabe ist weder **vorhersagbar** noch **reproduzierbar**

Verwendung: Schlüsselerzeugung, Noncen (Numbers used only-once) und einige weitere Anwendungen

PSEUDO RANDOM NUMBER GENERATORS (PRNG)

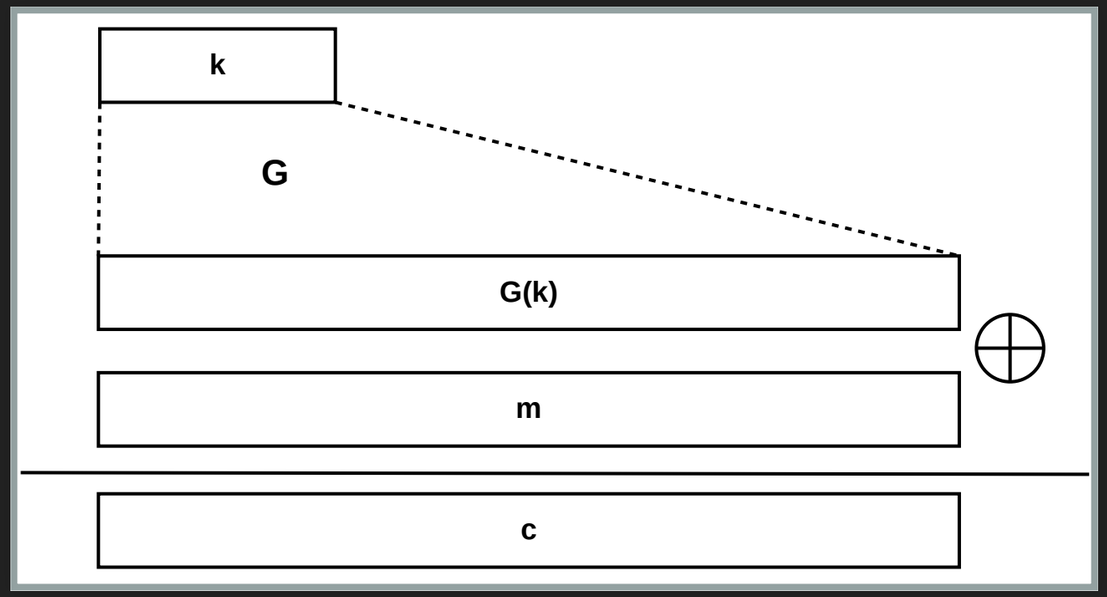
- Pseudozufallszahlengeneratoren (PRNG) erzeugen eine pseudo-zufällige Sequenz aus einem Seed
- Typischerweise so gebaut, das die Ausgabe gute statistische Eigenschaften hat
- Ausgabe ist **reproduzierbar** (das ist gut) und **vorhersagbar** (bei bekanntem Seed)

PSEUDO RANDOM NUMBER GENERATORS (PRNG)

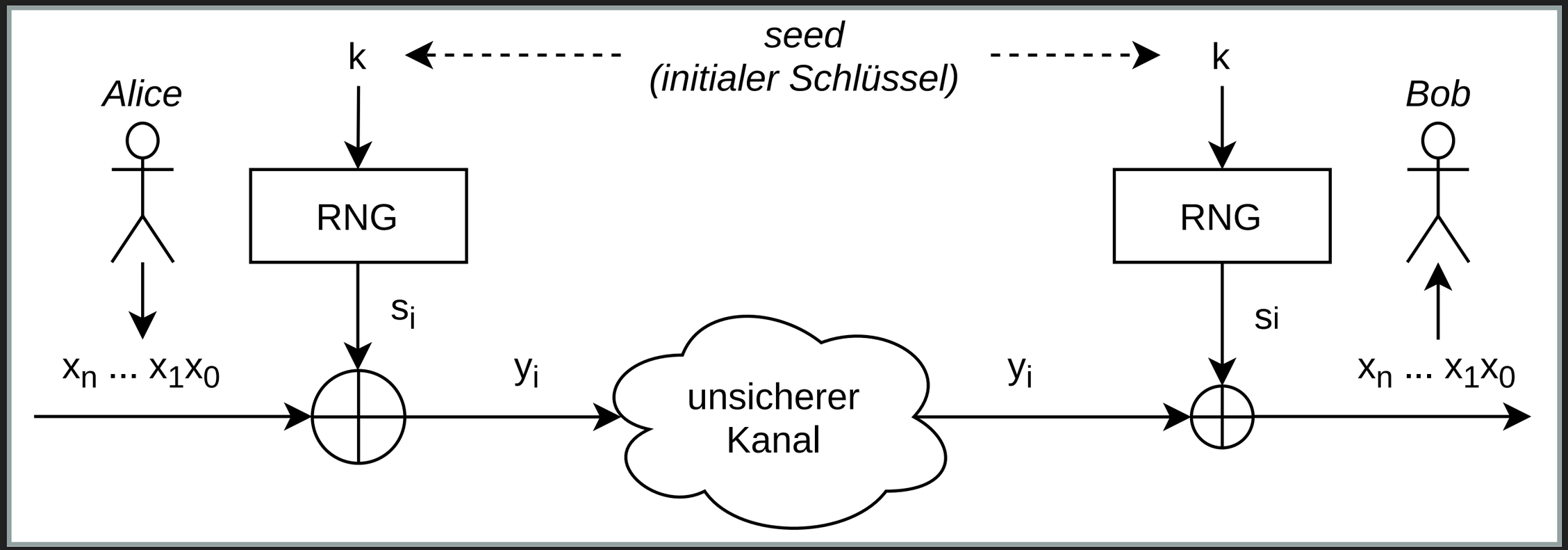
- Allgemein: Ein PRNG ist eine Funktion die wir Generator G nennen
- $G : \{0,1\}^s \rightarrow \{0,1\}^n$
  - G verwendet einen Seed s aus $\{0,1\}^s$
  - Die Ausgabe ist dann eine **viel** längere Ausgabe $\{0,1\}^n$ mit der Eigenschaft $n \gg s$
- Beispiel: Wir verwenden einen Seed mit 128 Bit und erzeugen eine Ausgabe die Mega/Giga/Terrabits lang ist
- G ist typischerweise ein deterministischer Algorithmus der effizient berechenbar ist
  - Nur die Eingabe, also der Seed, ist **zufällig** (random)
  - Die Ausgabe muss nur die Eigenschaft haben **zufällig** auszusehen

PSEUDO RANDOM NUMBER GENERATORS

- Den Ausgabestrom des PRNG kann man nun in einer Stromchiffre verwenden
- Der Seed ist hierbei der **geheime** Schlüssel
- Stromchiffre
  - $C = E_k(m) := m \oplus G(k)$
  - $D_k(m) := c \oplus G(k)$



STROMCHIFFRE ÜBERSICHT



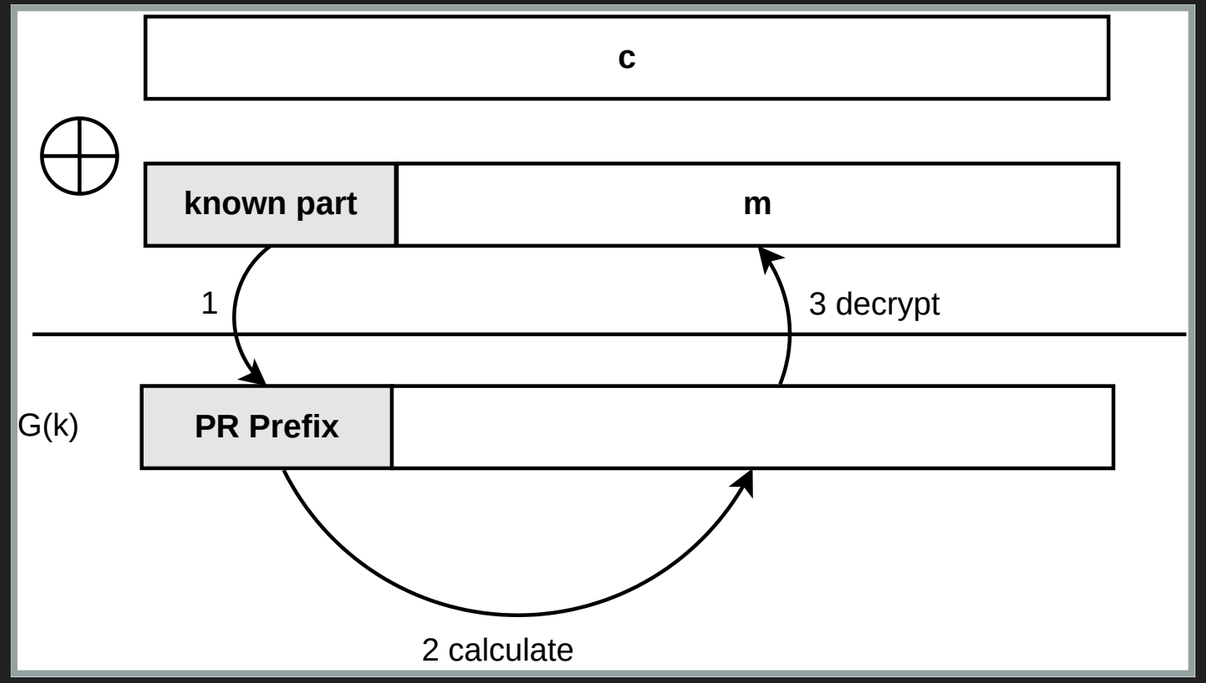
PRNG VORHERSAGBARKEIT

FRAGE: IST ES MÖGLICH DIE AUSGABE EINES PRNGS VORHERZUSAGEN OHNE DEN SEED ZU KENNEN?

SICHERHEIT VON STROMCHIFFREN

Die Ausgabe eines PRNG darf **nicht vorhersagbar** sein, wenn der Seed unbekannt ist!

- Nehmen wir an ein PRNG ist vorhersagbar
  - $\exists i : G(k) \mid 1, \ldots, i \rightarrow G(k) \mid i+1, \ldots, n$



SICHERHEIT VON STROMCHIFFREN

- Generator $G : k \rightarrow \{0,1\}^n$ ist vorhersagbar wenn:
- $\exists (effective\ algorithm) A, \exists 1 \leq i \leq n-1$:
- $Pr[A(G(k))\mid_{1,\ldots,i} = G(k))\mid_{i+1}] \geq \frac{1}{2} + \epsilon$
  - Für ein vernachlässigbar kleines $\epsilon \rightarrow (\epsilon \geq \frac{1}{2^{30}})$
- Wir definieren einen nicht vorhersagbaren PRNG, wenn **kein** effektiver Algorithmus existiert, der das nächste Bit mit einer Wahrscheinlichkeit besser als $\frac{1}{2} + \epsilon$ voraussagen kann.

FRAGE

Gegeben sei $G(k_0), G(k_1)$:
$G : k \rightarrow \{0,1\}^n \mid G(k_n) = G(k_{n-2}) \oplus G(k_{n-1})$

IST G VORHERSAGBAR?

- Nein, G ist nicht vorhersagbar
- Ja, wenn die ersten $(n-1)$ Bit bekannt sind kann das n'te Bit vorhergesagt werden
- Kommt drauf an

FRAGE

Gegeben sei $G(k_0), G(k_1)$:
$G : k \to \{0,1\}^n \mid G(k_n) = G(k_{n-2}) \oplus G(k_{n-1})$

IST G VORHERSAGBAR?

- Nein, G ist nicht vorhersagbar
- **Ja, wenn die ersten $n-1$ Bit bekannt sind kann das n'te Bit vorhergesagt werden**
- Kommt drauf an

CRYPTOGRAPHICALLY SECURE PRNG (CSPRNG)

- Ein CSPRNG muss **unvorhersagbar** sein
- **Genauer**: Bei gegebenen n aufeinanderfolgenden Ausgabebits $s_{i_{0 \ldots n}}$ sind die nachfolgenden Bit $s_{i_{n+1}}$ nicht vorhersagbar (in polynomialer Zeit)

- CSPRNG werden daher primär in der Kryptographie, insbesondere bei Stromchiffren, verwendet
- Anmerkung: Außerhalb der Kryptographie gibt es fast keine Anwendung die die Eigenschaft **Nichtvorhersagbarkeit** benötigen, aber viele (technische) Systeme benötigen einfachere PRNGs

KRYPTOGRAPHISCH UNSICHERER ZUFÄLLIGKEIT

- Die `glibc` Implementierung der Funktion `rand()` verwendet einen sog. Linearen Kongruenz-Generator (LCG)
  - Bei einem gegebenen Seed generiert `rand()` Zufallszahlen mit guten statistischen Eigenschaften (0, 1 gleichmäßig verteilt)
  - Das ist gut genug für Spiele, aber nicht für Kryptographie
- Es existieren **effiziente** Algorithmen die bei gegebenem $(n-1)$ Bit das n'te Bit berechnen können
- Die meisten Implementierungen von einfachen `random()`-Funktionen basieren auf LCGs oder dem sog. Mersenne Twister Algorithmus
  - LCGs: Parameter und interne Funktionen unterscheiden sich nur
- Sowohl LCGs als auch Mersenne Twister gelten als kryptographisch unsicher und sind berechenbar, wenn eine bestimmte Anzahl an Ausgaben zur Verfügung steht

	Verwenden Sie niemals die Funktion `random()` (Standard Implementierungen) für Kryptographie!

WOHER BEKOMMT MAN KRYPTOGRAPHISCH SICHERE ZUFALLSZAHLEN?

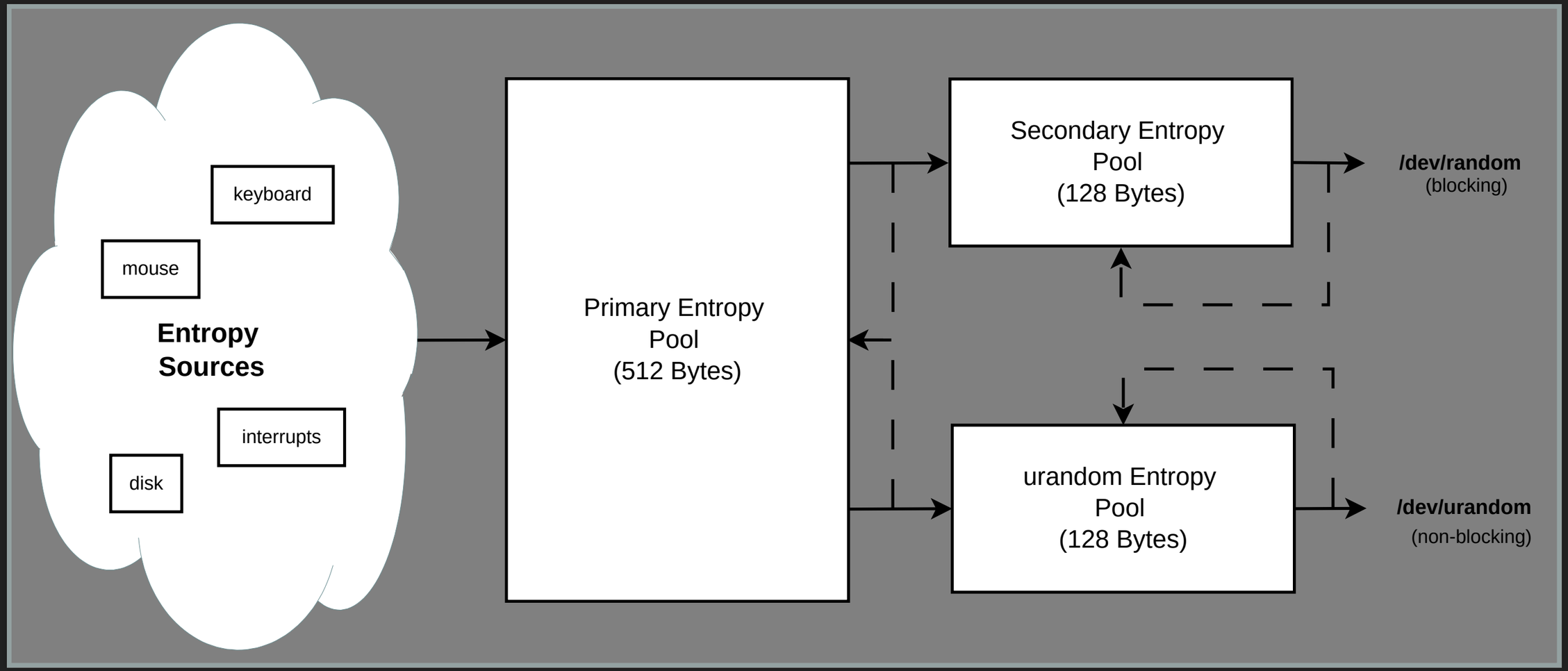
- Online:
  - https://www.random.org/
- Hardware
  - `RdRand()` Instruktion (ab Intel Ivy Bridge, AMD ab ca. 2015)
  - Spezielle Hardware die radioaktiven Zerfall, Leitungsrauschen, Wärmerauschen, . . . verwenden

WOHER BEKOMMT MAN KRYPTOGRAPHISCH SICHERE ZUFALLSZAHLEN?

- RNGs die externe Entropie-Quellen einbeziehen:
  - Unixoide Systeme (z.B. Linux) http://man7.org/linux/man-pages/man4/random.4.html
    - `/dev/random`
    - `(/dev/urandom)`
  - Microsoft
    - `CryptGenRandom()` http://msdn.microsoft.com/en-us/library/windows/desktop/aa379942%28v=vs.85%29.aspx
- Weitere Informationen http://www.ietf.org/rfc/rfc1750.txt

BEISPIEL UNIX/LINUX

Beispiel: `/dev/random`



- Details: (https://eprint.iacr.org/2006/086.pdf)

BEISPIEL JAVA?

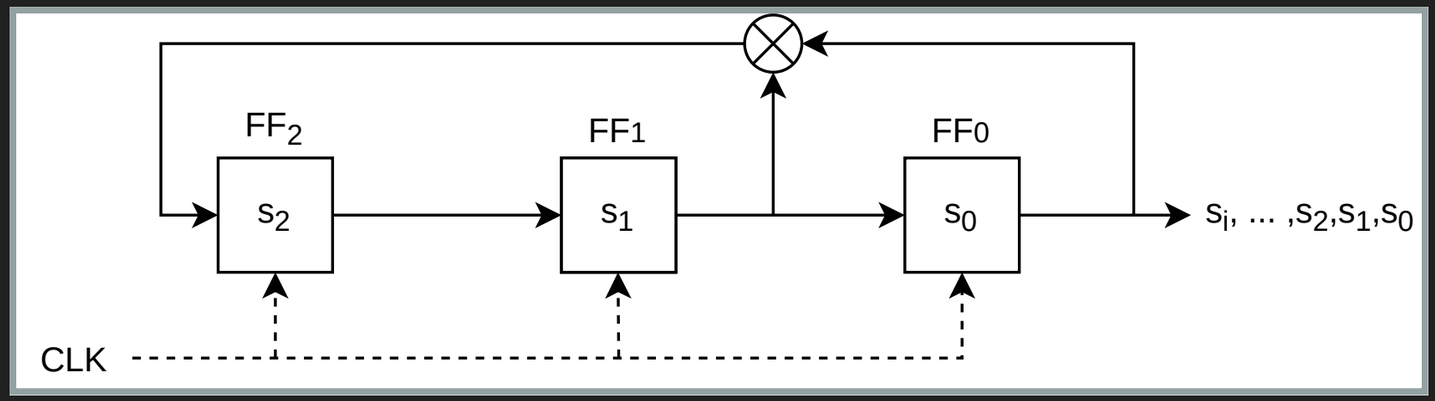
```java
SecureRandom random = new SecureRandom();
/* option 1 */
byte bytes[] = new byte[20];
random.nextBytes(bytes);

/* option 2 */
byte seed[] = random.generateSeed(20);
```

Abhängig von der Implementierung kann es sein das die Funktionen `generateSeed` und `nextBytes` blockieren. Dann muss erst wieder genug Entropie angesammelt werden. Das kann passieren wenn von `/dev/random` gelesen wird.

KONSTRUKTION VON PRNGS

LFSR BEISPIEL

Ein einfaches LFSR kann mit der Hilfe von Flip Flops und einer XOR-Schaltung relativ einfach beschrieben werden.



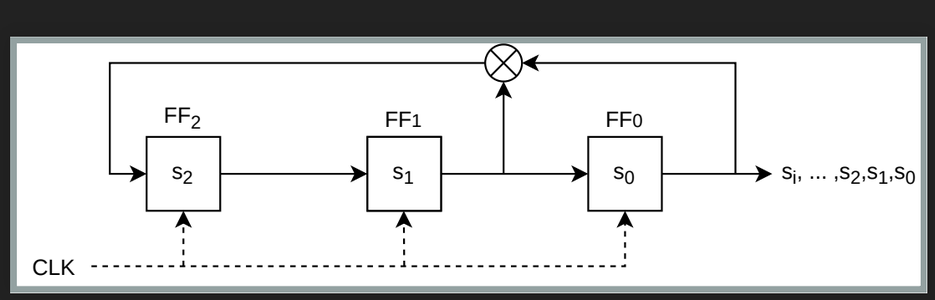
Nimmt man jetzt einen Anfangszustand, z.B. ($s_2 = 1, s_1 = 0, s_0 = 0$) kann man die generierte Zahlenfolge einfach berechnen.

Mathematisch lassen sich die Operationen beschreiben durch:

$$s_3 \equiv s_1 + s_0 \mod 2$$
$$s_4 \equiv s_2 + s_1 \mod 2$$
$$s_5 \equiv s_3 + s_2 \mod 2$$

LFSR - AUFGABE ✍

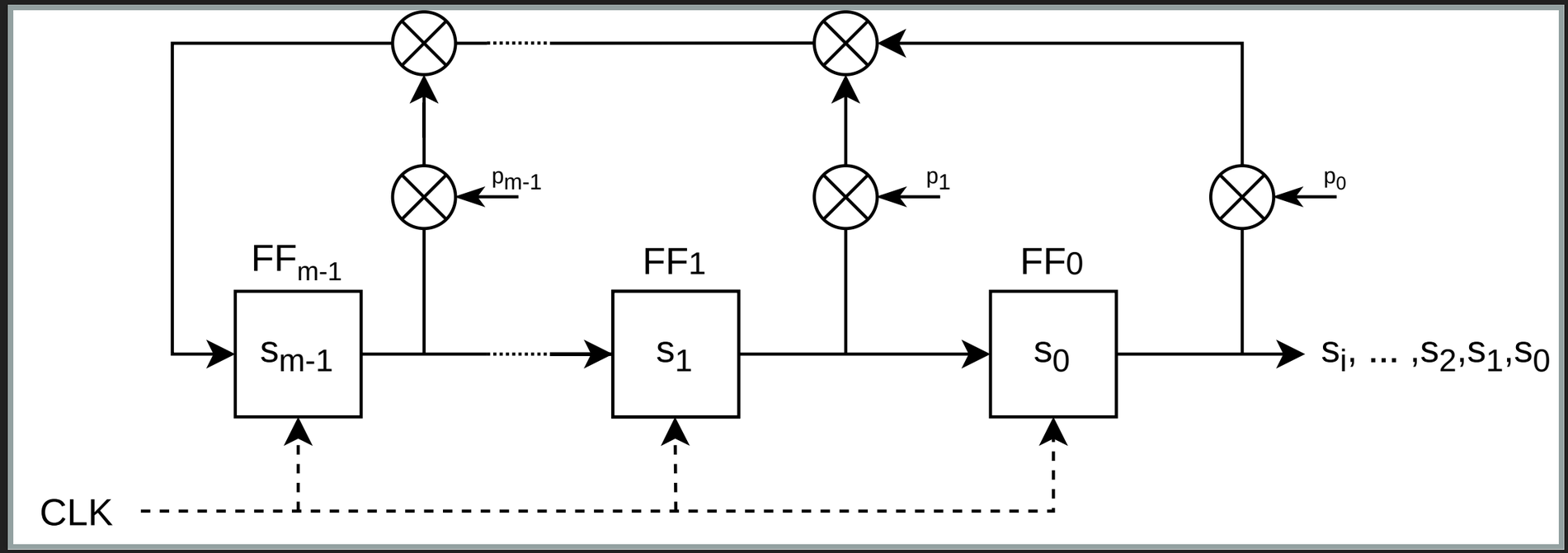
Berechnen Sie die generierte Ausgabefolge des angegebenen LFSR's. Füllen Sie hierzu die nachfolgende Tabelle aus (⏱ 10 Minuten). Fällt Ihnen hierbei etwas auf?



$$s_3 \equiv s_1 + s_0 \mod 2$$
$$s_4 \equiv s_2 + s_1 \mod 2$$
$$s_5 \equiv s_3 + s_2 \mod 2$$

Takt	FF_2	FF_1	$FF_0 = s_i$
0	1	0	0
1			
2			
3			
4			
5			
6			
7			
8			

LFSR - ALLGEMEINE FORM



Mathematische Form: $s_m \equiv s_{m-1}p_{m-1} + \ldots + s_1 p_1 + s_0 p_0 \mod 2$

Die Ausgangssequenz lässt sich mit folgendem Ausdruck beschreiben:

$$s_{m+i} \sum_{j=0}^{m-1} p_j * s_{i+j} \mod 2, \text{für } s_i, p_j \in \{0, 1\} \text{ mit } i = 0, 1, 2, \ldots$$

LFSR - ALLGEMEINE FORM

Man sieht, die Ausgabewerte werden anhand vorheriger Ausgangsbits berechnet.

Der angegebene Ausdruck $s_{m+i} \sum_{j=0}^{m-1} p_j * s_{i+j} \mod 2$ wird als lineare Differenzialgleichung bezeichnet. Das in einem LFSR nur endlich viele interne Zustände gibt, wiederholt sich die Ausgangssequenz eines LFSR periodisch. Die Periodenlänge hängt hierbei von den Rückkopplungskoeffizienten p_j ab.

	Die maximale Länge einer Bitfolge eines LFSR vom Grad m ergibt sich aus $2^m - 1$. Dies nennt man die **Maximalfolge** eines LFSR.

LFSR - MAXIMALFOLGE

Frage: Warum ist die Maximalfolge eines LFSR $2^m - 1$ und nicht etwa 2^m?

LFSR - MAXIMALFOLGEN

Nicht jeder mögliche Rückkopplungspfad (die Wahl bestimmter Rückkopplungskoeffizienten) erzeugt eine Maximalfolge.

Beispiele:

- LFSR vom Grad $m = 4$ mit ($p_3 = 0, p_2 = 0, p_1 = 1, p_0 = 1$) hat eine Maximalfolge mit der Periodenlänge $2^m - 1 = 15$
- LFSR vom Grad $m = 4$ mit ($p_3 = 1, p_2 = 1, p_1 = 1, p_0 = 1$) hat eine Maximalfolge mit der Periodenlänge 5, was **keine Maximalfolge** ist

	Jedes LFSR mit den Rückkopplungskoeffizienten $p_{m-1}, \ldots, p_1, p_0$ kann durch ein Polynom eindeutig beschrieben werden: $P(x) = x^m + p_{m-1} * x^{m-1} + \ldots + p_1 x + p_0$

LFSR - MAXIMALFOLGEN

Hierdurch ergeben sich folgende Polynome anhand der Form:

$$P(x) = x^4 + p_3 * x^3 + p_2 * x^2 + p_1 * x + p_0$$

- LFSR vom Grad $m = 4$ mit ($p_3 = 0, p_2 = 0, p_1 = 1, p_0 = 1$)
  - $P(x) = x^4 + 0 * x^3 + 0 * x^2 + 1 * x + 1$
  - $P(x) = x^4 + x + 1$
- LFSR vom Grad $m = 4$ mit ($p_3 = 1, p_2 = 1, p_1 = 1, p_0 = 1$)
  - $P(x) = x^4 + 1 * x^3 + 1 * x^2 + 1 * x + 1$
  - $P(x) = x^4 + x^3 + x^2 + x + 1$

	Nur sog. **primitive** oder **irreduzible Polynome** erzeugen Maximalfolgen.

LFSR - MAXIMALFOLGEN

Grundsätzlich sollten die Rückkopplungskoeffizienten so gewählt sein, dass das LFSR Maximalfolgen erzeugt. In dem Fall, erzeugt ein LFSR einen zufällig aussehenden Datenstrom der statistisch gute Eigenschaften aufweist und dessen Periodenlänge fast gleich des Grades des Polynoms $2^m - 1$ entspricht.

	Generell eigenen sich einzelne LFSR **nicht** dazu kryptographisch sichere PRNGs zu implementieren und sind somit ungeeignet für die Verwendung in Stromchiffren.

LINEARE FUNKTIONEN

Betrachten wir die Abbildung der Addition, also der linearen logischen `XOR` Funktion $\oplus$, in $\mathbb{Z}_2$.

$x + y \equiv z \bmod 2 \rightarrow x \oplus y \equiv z$ für alle $x, y, z \in \mathbb{Z}_2$

+	0	1	y
0	0	1	
1	1	0	
x			

- x soll nun den Plaintext, y den Schlüssel und z den Ciphertext abbilden.

LINEARE FUNKTIONEN

Wenn wir nun eine bitweise Operation betrachten, also z.B.:

$$
\begin{aligned}
x_0 \quad \oplus \quad y_0 \quad &\equiv \quad z_0 \bmod 2 \\
x_1 \quad \oplus \quad y_1 \quad &\equiv \quad z_1 \bmod 2 \\
x_2 \quad \oplus \quad y_2 \quad &\equiv \quad z_2 \bmod 2 \\
x_3 \quad \oplus \quad y_3 \quad &\equiv \quad z_3 \bmod 2
\end{aligned}
$$

dann können wir zu 2 gegebenen Werten, also z.B. y_0, z_0 sehr einfach x_0 berechnen.
Das liegt daran, dass die Funktion linear und umkehrbar ist.

NICHTLINEARE FUNKTIONEN

Bei nichtlinearen Funktionen, ist es bei gegebenem y_0, z_0 nicht möglich / entscheidbar x_0 eindeutig zu bestimmten. Die Multiplikation in $\mathbb{Z}_2$ entspricht dem logischen `AND` $\wedge$.

*	0	1	y
0	0	0	
1	0	1	
x			

$$
\begin{aligned}
x_0 \quad \wedge \quad y_0 \quad &\equiv \quad z_0 \mod 2 \\
x_1 \quad \wedge \quad y_1 \quad &\equiv \quad z_1 \mod 2 \\
x_2 \quad \wedge \quad y_2 \quad &\equiv \quad z_2 \mod 2 \\
x_3 \quad \wedge \quad y_3 \quad &\equiv \quad z_3 \mod 2
\end{aligned}
$$

Wenn hier gilt $x = z = 1$ dann weiß man, dass auch $y = 1$ sein muss. Aber falls gilt $x = z = 0$, dann kann y entweder $y = 0$ oder $y = 1$ sein.

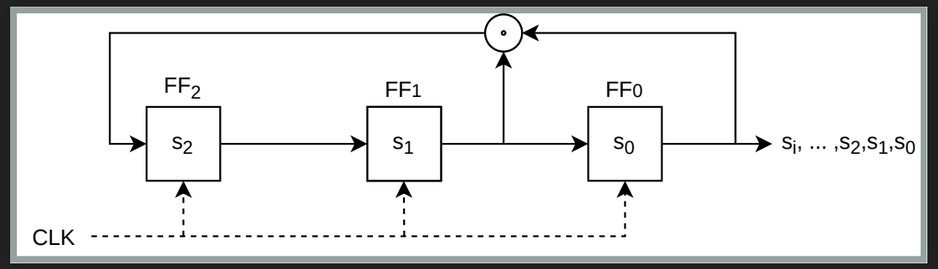
Da die Multiplikation in $\mathbb{Z}_2$ nicht invertierbar ist, kann sie natürlich nicht direkt für die Ver/Entschlüsselung verwendet werden.

AND-BASIERTE LFSRS

Da LFSRs allerdings nur einen Schlüsselstrom erzeugen, könnte man nun auf die Idee kommen im LFSR selbst logische `AND` 's anstelle von `XOR` 's zu verwenden. Hierbei ist allerdings das Problem, das die Maximalfolgen extrem eingeschränkt werden, bzw. es vorkommen kann das man einen Endzustand erreicht, der nicht wieder verlassen werden kann.

LFSR-AND AUFGABE ✍

Berechnen Sie die generierte Ausgabefolge des angegebenen LFSR's. Füllen Sie hierzu die nachfolgende Tabelle aus (⏱ 10 Minuten). Fällt Ihnen hierbei etwas auf?



$$s_3 \equiv s_1 * s_0 \mod 2$$
$$s_4 \equiv s_2 * s_1 \mod 2$$
$$s_5 \equiv s_3 * s_2 \mod 2$$

Takt	FF_2	FF_1	$FF_0 = s_i$
0	1	0	0
1			
2			
3			
4			
5			
6			
7			
8			

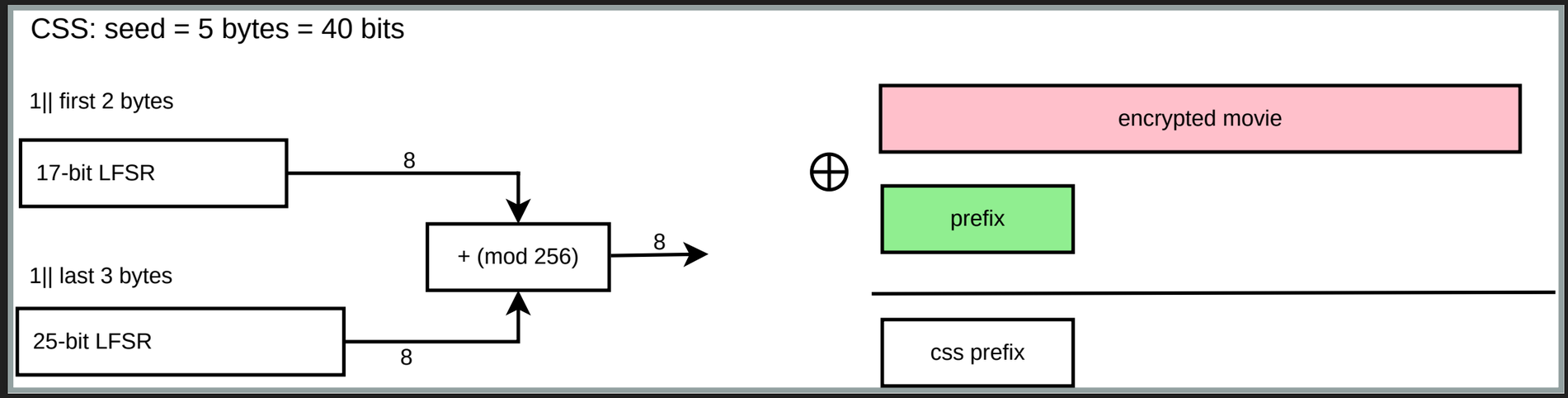
SICHERHEIT VON "ÄLTEREN" STROMCHIFFREN AUF BASIS VON LFSR

Es eigenen sich keine Netze aus LFSR (mit Maximalfolgen), die nur mittels **linearen** Bool'schen Operationen zusammengeschaltet sind. Dazu zählen u.a.:

- CSS (2 LFSRs) - Content Scrambling System
- A5/1 & A5/2 (3 LFSRs) - GSM Netze
- E0 4 LFSRs - Bluetooth

⚠	Diese Algorithmen gelten alle als gebrochen

BEISPIEL: BAD AND BROKEN CSS



Für alle möglichen Werte des 17-bit LFSR (10000_{16} ... $1FFFF_{16}$):

- Berechnung des 17-bit LFSR und Erzeugung von 20 Bytes der Ausgaben (O_{LFSR17})
- Subtraktion der Ausgabe von CSS prefix $\Rightarrow$ Mögliche Ausgabe von des 25-bit LFSR (O_{LFSR25})
- Prüfe ob O_{LFSR25} eine mögliche Ausgabe ist (Umkehrfunktion)
- Falls ja $\rightarrow$ Beide Initialen Eingaben gefunden

MODERNE STROMCHIFFREN BEISPIELE

MODERNE STROMCHIFFREN: TRIVIUM

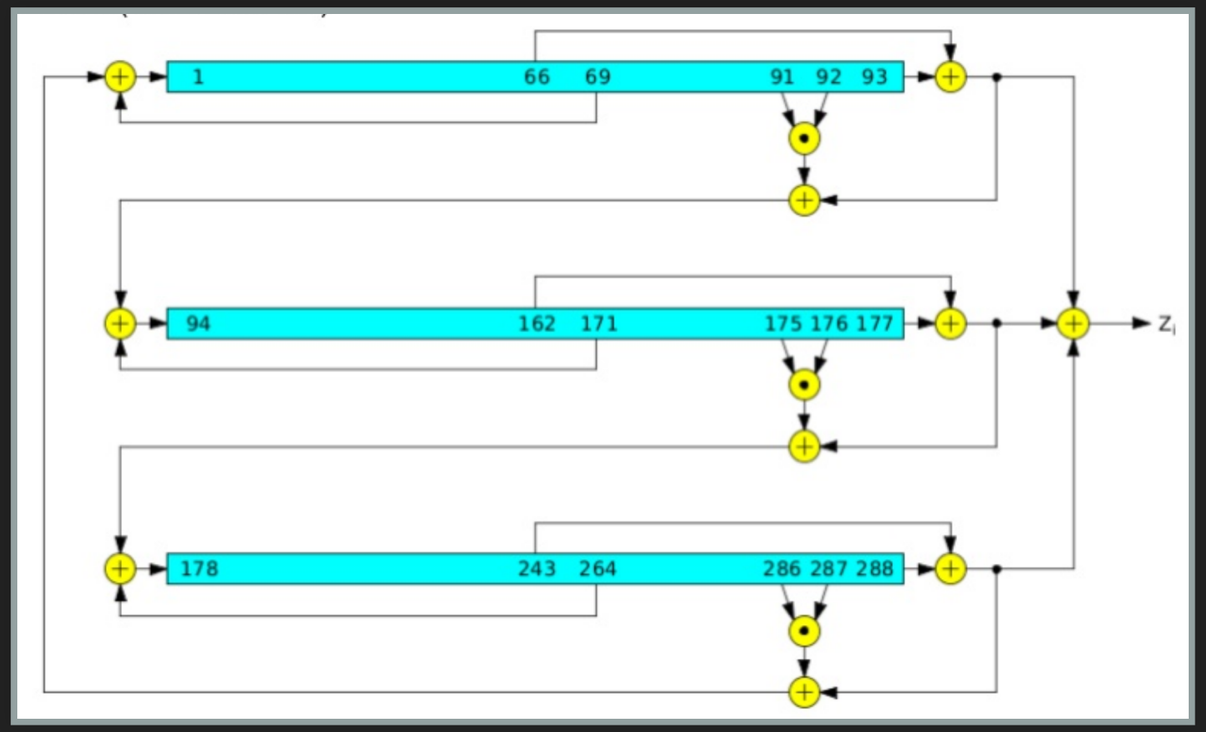
Trivium ist eine moderne Stromchiffre, die kryptographisch als sicher gilt. Es handelt sich hierbei also im Prinzip um einen CSPRNG. Trivium basiert grundsätzlich auf einem Netzwerk von LFSR's, die jedoch z.T. mit nicht-linearen Bool'schen Operationen (logisch **UND** (**AND**)) erweitert sind (NLFSR). Hierdurch ist es nicht länger möglich mit einfachen Mitteln das Gleichungssystem zu lösen.

Durch die Verwendung der bool'schen `AND` Operation, die in Modulo 2 der Multiplikation entspricht, ist die Chiffre nicht länger linear, da nun die Multiplikation von 2 Unbekannten erfolgt.

MODERNE STROMCHIFFREN: TRIVIUM

Initialisierung:

- Schlüssel in das obere LFSR, IV in das mittlere LFSR, fester Wert in das untere LFSR
- 1152 x Schiebeoperation ausführen



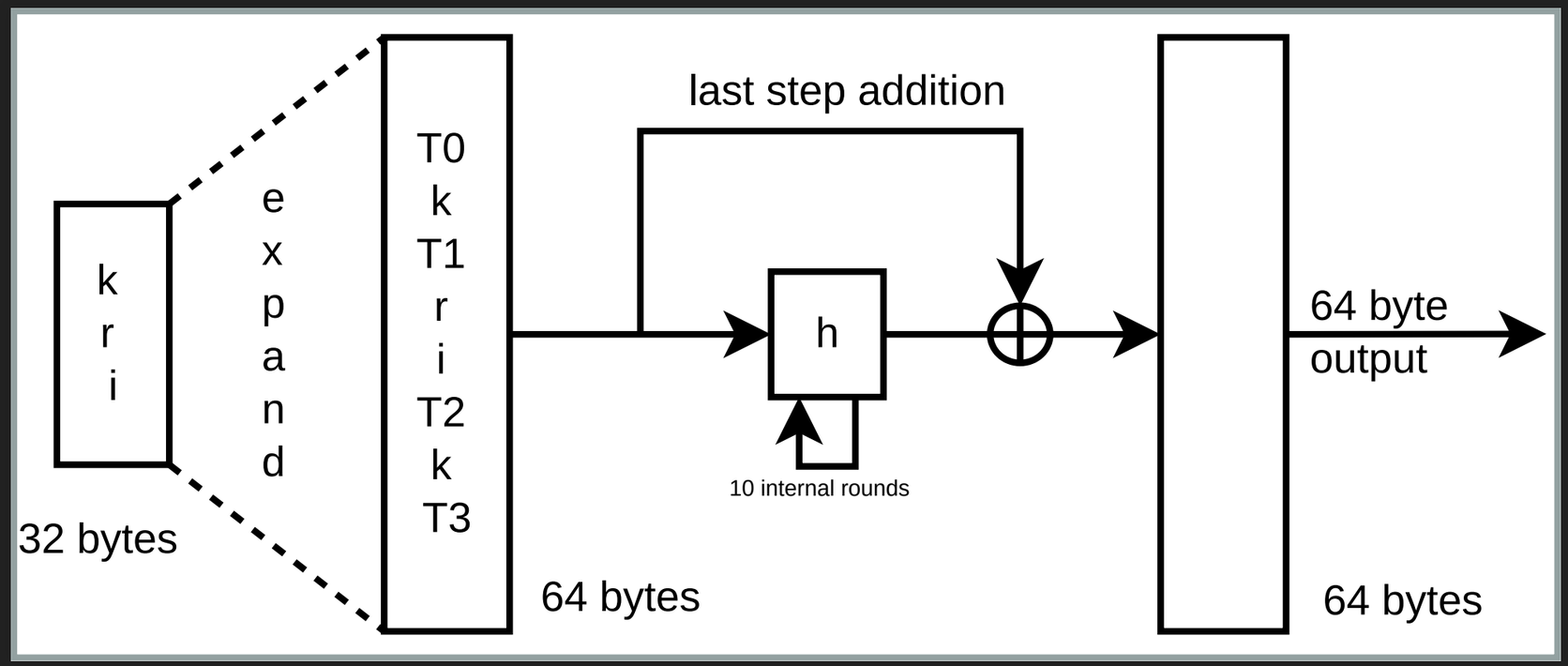
MODERNE STROMCHIFFREN: ESTREAM

- PRG: $\{0,1\}^s \times R \to \{0,1\}^n$
  - $\{0,1\}^s$ (Seed)
  - R (Nonce)
- Nonce wird nicht wiederholt für einen Schlüssel (z.B. inkrementieren)
- Verschlüsselung: $E(k,r,m) = m \oplus PRG(k,r)$

MODERNE STROMCHIFFREN: SALSA20

Salsa20: $\{0,1\}^{128|256} \times \{0,1\}^{64} \to \{0,1\}^{n}$

$$Salsa20(k,r) = H(k,(r,0))||H(k,(r,1))|| \dots$$



$T0, T1, T2, T3 \to$ Konstanten, i: Counter der nach jeder Ausgabe inkrementiert wird — h: Funktion die schnell auf X86 Systemen mit SSE Erweiterung läuft

SALSA20: INITIALISIERUNG

Verteilung Initialwerte Indices für QR

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

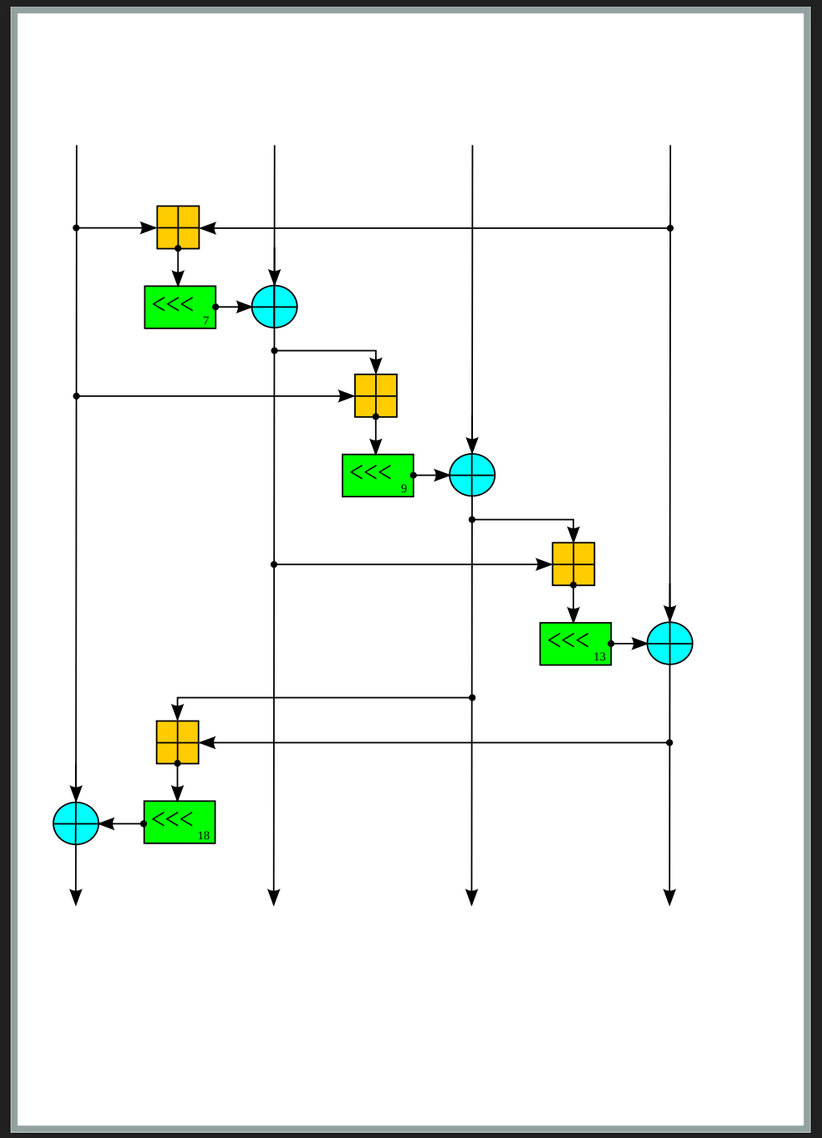
Salsa20 Initiale Werteverteilung

T1	Key	Key	Key
Key	T2	Nonce (R)	Nonce (R)
Pos(i)	Pos(i)	T3	Key
Key	Key	Key	T4

Pos(i): Counter der nach jeder Ausgabe von 64 Byte automatisch inkrementiert wird.
Nur dieser Wert verändert sich!

SALSA20: RUNDENFUNKTION H

```c
#include <stdint.h>

#define ROTL(a,b) (((a) << (b)) | ((a) >> (32 - (b))))
#define QR(a, b, c, d)(   \
  b ^= ROTL(a + d, 7),  \
  c ^= ROTL(b + a, 9),  \
  d ^= ROTL(c + b,13),  \
  a ^= ROTL(d + c,18))
#define ROUNDS 20

void salsa20_block(uint32_t out[16],
                   uint32_t const in[16])
{
  int i;
  uint32_t x[16];

  for (i = 0; i < 16; ++i)
    x[i] = in[i];
  // 10 loops × 2 rounds/loop = 20 rounds
  for (i = 0; i < ROUNDS; i += 2) {
    // Odd round
    QR(x[ 0], x[ 4], x[ 8], x[12]); // column 1
    QR(x[ 5], x[ 9], x[13], x[ 1]); // column 2
    QR(x[10], x[14], x[ 2], x[ 6]); // column 3
    QR(x[15], x[ 3], x[ 7], x[11]); // column 4
    // Even round
    QR(x[ 0], x[ 1], x[ 2], x[ 3]); // row 1
    QR(x[ 5], x[ 6], x[ 7], x[ 4]); // row 2
    QR(x[10], x[11], x[ 8], x[ 9]); // row 3
    QR(x[15], x[12], x[13], x[14]); // row 4
  }

  for (i = 0; i < 16; ++i)
    out[i] = x[i] + in[i];
}
```



CHACHA20 (SALSA20 VARIANTE 2008)

Verteilung Initialwerte Indices für QR

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

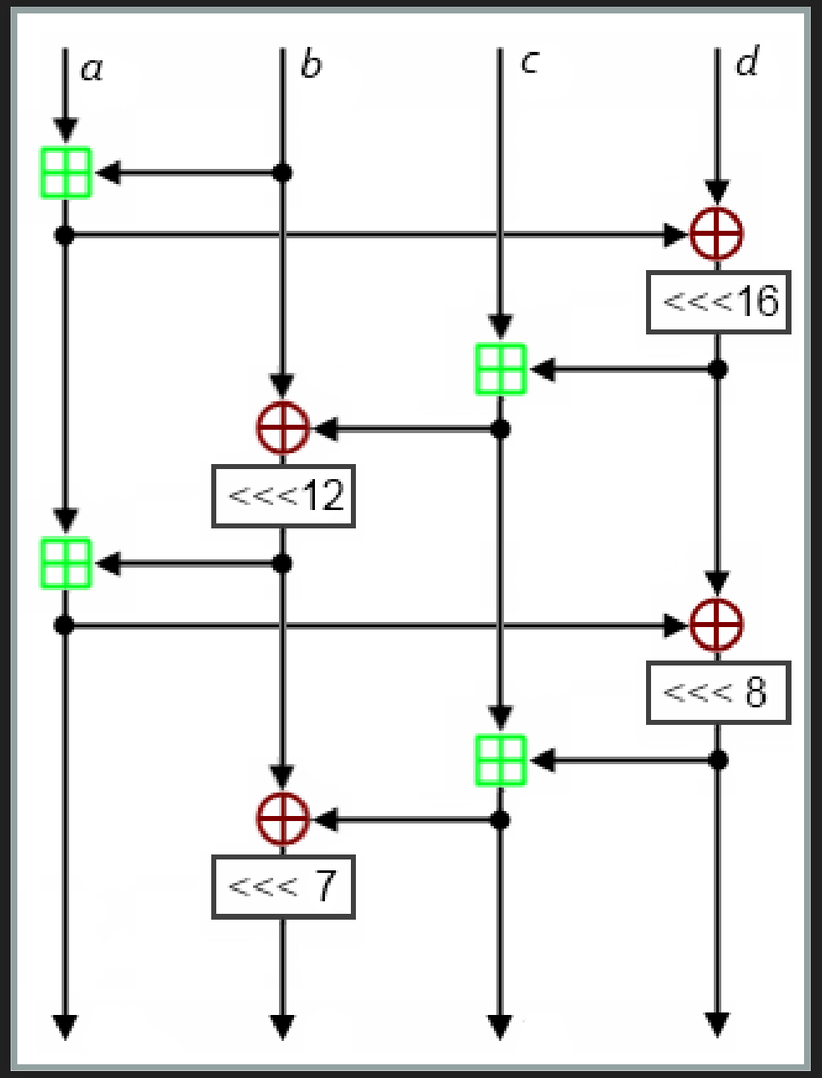
ChaCha20 Initiale Werteverteilung

T1	T2	T3	T3
Key	Key	Key	Key
Key	Key	Key	Key
Pos(i)	Pos(i)	Nonce (R)	Nonce (R)

Pos(i): Counter der nach jeder Ausgabe von 64 Byte automatisch inkrementiert wird.
Nur dieser Wert verändert sich!

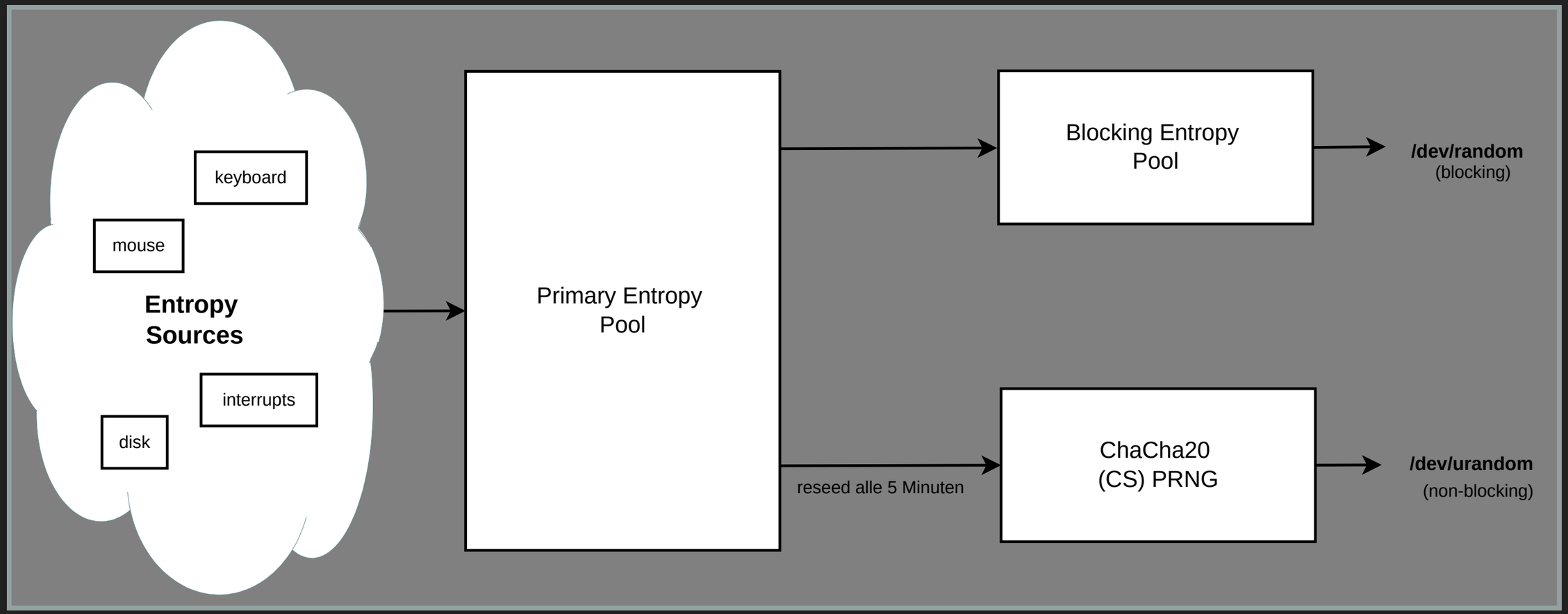
CHACHA20 (SALSA20 VARIANTE 2008)

```c
#include <stdint.h>

#define ROTL(a,b) (((a) << (b)) | ((a) >> (32 - (b))))
#define QR(a, b, c, d) (      \
  a += b,  d ^= a,  d = ROTL(d,16), \
  c += d,  b ^= c,  b = ROTL(b,12), \
  a += b,  d ^= a,  d = ROTL(d, 8), \
  c += d,  b ^= c,  b = ROTL(b, 7))
#define ROUNDS 20

void chacha_block(uint32_t out[16],
                  uint32_t const in[16])
{
  int i;
  uint32_t x[16];

  for (i = 0; i < 16; ++i)
    x[i] = in[i];
  // 10 loops × 2 rounds/loop = 20 rounds
  for (i = 0; i < ROUNDS; i += 2) {
    // Odd round
    QR(x[0], x[4], x[ 8], x[12]); // column 0
    QR(x[1], x[5], x[ 9], x[13]); // column 1
    QR(x[2], x[6], x[10], x[14]); // column 2
    QR(x[3], x[7], x[11], x[15]); // column 3
    // Even round
    QR(x[0], x[5], x[10], x[15]); // diagonal 1
    QR(x[1], x[6], x[11], x[12]); // diagonal 2
    QR(x[2], x[7], x[ 8], x[13]); // diagonal 3
    QR(x[3], x[4], x[ 9], x[14]); // diagonal 4
  }
  for (i = 0; i < 16; ++i)
    out[i] = x[i] + in[i];
}
```



REVISIT: UNIX/LINUX (4.8) RANDOM

Beispiel: `/dev/random`



- Details: BSI Analyse

AUFGABEN: MANY TIME PADS ✍

Bearbeiten Sie die Aufgaben MTP1, MTP2 und MTP3

Bei der Aufgabe *MTP3* handelt es sich um eine Aufgabe die bei der Abgabe Bonuspunkte gibt.