CRYPTOGRAPHY LECTURE, 2023

Master level

Mirosław Kutyłowski

Cryptographic Random Numbers

Ideal model: again a Random Oracle:

a blackbox D outputting bits:

- at step t it outputs $D(t)$ selected at random by "coin tossing"
- unlike for hash functions: the outputs are bits, so collisions occur

Definitely useful: example a commitment

purpose: converting adaptive randomized protocols to non-adaptive randomized protocols

creating a commitment: Alice commits to a value r but does not present it to Bob

i. Alice chooses a k-bit string w

ii. Alice computes $C := \mathrm{Hash}(w, r)$

iii. Alice presents commitment C to Bob

opening a commitment: Alice presents r and proves that it corresponds to C:

i. Alice shows r and w

ii. Bob checks that $C = \mathrm{Hash}(w, r)$

Properties of commitment:

i. Bob cannot recover w based on C (one-way property of hashes, there are many solutions!)

ii. even if Bob knows w (for some reason), he cannot predict r and check

Conversion to non-adaptive protocols:

i. Alice chooses random numbers r_1, r_2, ... (r_i is the randomness for the ith step of the algorithm)

ii. Alice computes and presents commitments C_1, C_2, ... for r_1, r_2, ...

iii. at step i Alice opens C_i and executes the algorithm step deterministically for randomness r_i

Advantage:

- a randomized algorithm may assume that the participants are honestly executing "choose r at random"
- it is so risky in a multiparty protocol!
- via the conversion: a malicious participant cannot adopt to the situation and choices of other participants

Consensus protocols

- some number of participants: $A_1, ..., A_n$
- each A_i holds a value v_i
- task: reach an consensus for v which must belong to the set $\{v_1, ..., v_n\}$

example: leader election: v_i is the identifier of A_i

Problem: the participants can cheat for own advantage (***Byzentine nodes***)

example: virtual traffic lights

Example Solution for Leader Election

execution from the point of view of A_i:

i. A_i chooses r_i at random, i.e. $r_i := \mathrm{rand}()$ (k bit numbers)

ii. A_i computes $C_i := \mathrm{Commitment}(\mathrm{Hash}(r_i, \mathrm{ID}_{A_i}))$

iii. A_i broadcasts C_i and receives commitments from other participants

iv. once all commitments received: A_i sends opening to C_i

v. A_i computes $S := \mathrm{SORT}(r_1, \ldots, r_n)$

vi. A_i computes differences: if $S = (s_1 \ldots, s_n)$, then $d_i := s_{i+1} - s_i$ for $i < n$

and $d_n := s_1 + 2^k - s_n$

vii. A_j is the leader if $s_i = r_j$ and d_j is the biggest one

Indistinguishability game for a generator D

input: generator D or a true random source R, each with pbb $\frac{1}{2}$

operation: a distinguisher can run the generator any number of times

result: the distinguisher says "D'' or "R"

the generator D is not good if the distinguisher answers correctly with pbb $0.5 + \varepsilon$, where ε is not negligible

Derived properties

→ **forward unpredictability:** knowing the output to step t is is infeasible what will come next

→ **backwards unpredictability:** knowing the output starting from step t, it is infeasible to guess the output for steps 1 through $t-1$

→ **no properties like:** the average fraction of zeroes in the output is 0.4 ...

Randomness amplification

Random source R with some weaknesses (like bias for $0's$)

i. $z := R()$

ii. output($F(z)$) where F is a deterministic function mimicking Random Oracle

example: F is a good hash function

Pseudorandom number generator

model:

- internal state S changing in time
- transition function: $S_{t+1} := T(S_t)$
- output: $b_t := G(S_t)$

good practice: (bitsize of b_t) $\ll$ (bitsize of S_t)

(learning S_t from b_t impossible due to information theoretic argument)

(the attack does not work iff F has the property discussed)

Imperfect Generator Example

i. choose K at random

ii. generate $\mathrm{Hash}(K,1)\|\mathrm{Hash}(K,2)\|\mathrm{Hash}(K,3)\|\ldots$

correlated input secure hash function $\Rightarrow$ the output indistinguishable from true random

Problem

- adversary retreiving the internal state of the generator (side-channel attack, ...)
- after getting K the adversary can re-run the generator from the beginning (backwards predictable)

Securing PRNG – FIPS approach

a) transition function is a **one-way function**

$\Rightarrow$ leaked internal state does not endanger the previous outputs

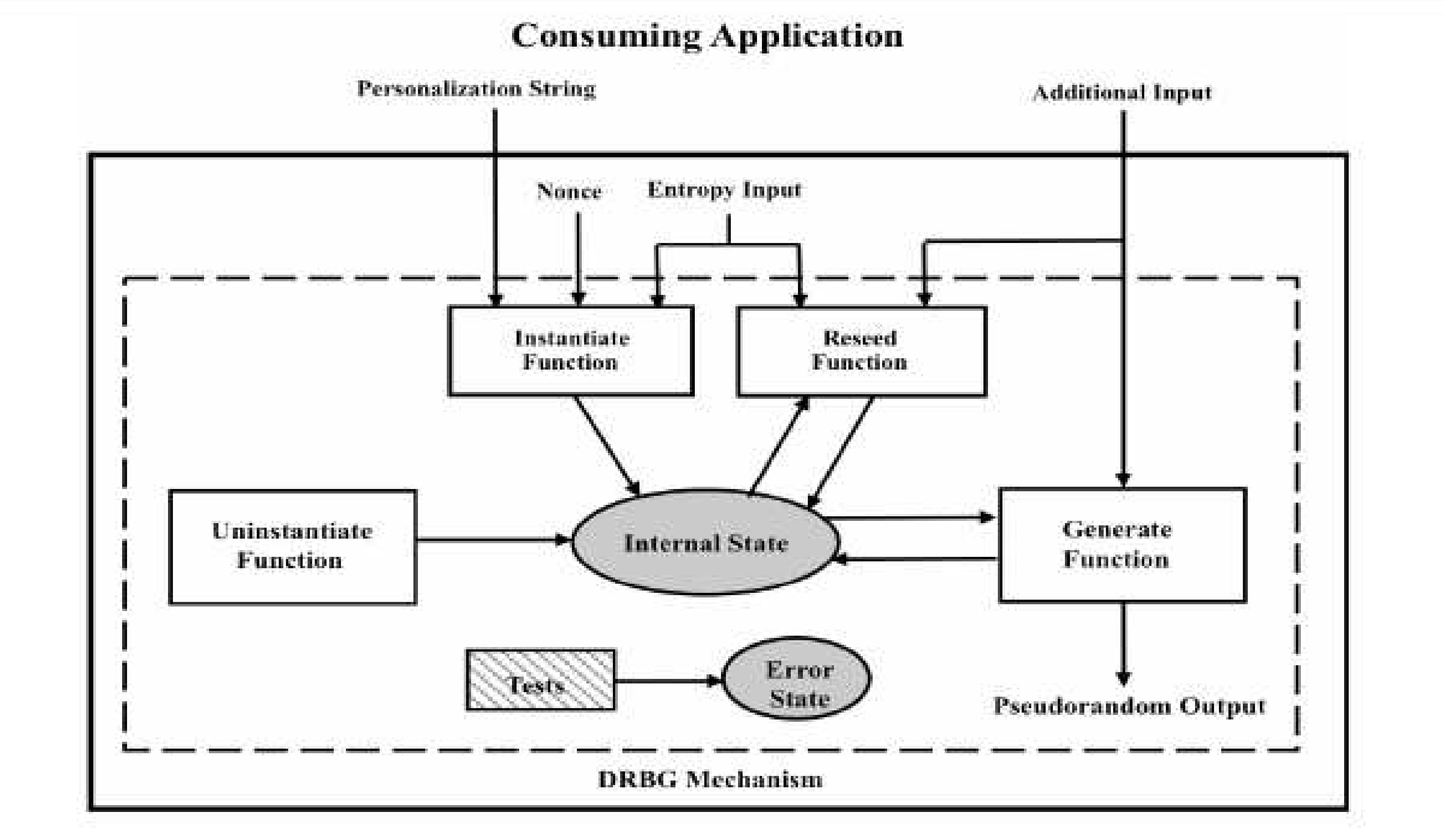
b) PRNG contains **internal entropy source**

$\Rightarrow$ refreshing procedure, to defend against seed retention by the PRNG provider

FIPS Approved Random Number Generators

NIST approach: standardization of cryptographic functions to be deployed on cryptographic secure modules according to FIPS 140-2

- **nondeterministic** generators not approved,
- **deterministic**: special NIST Recommendation, in fact "deterministic" means deterministic but with some random input
- first an approved entropy source creates a seed , then deterministic part


Consuming Application
Personalization String
Additional Input
Nonce
Entropy Input
Instantiate Function
Reseed Function
Uninstantiate Function
Internal State
Generate Function
Tests
Error State
Pseudorandom Output
DRBG Mechanism

Instantiation:

- the seed with limited validity period, once expired a new seed has to be used
- reseeding function creates a different seed
- different instantiations of a DRNG can exist at the same time, they MUST be independent in terms of the seeds and usage

Internal state:

- secret cryptographic chain value, the counter of output requests served so far
- different instantiations of DRBG must have separate internal states

Instantiation strength:

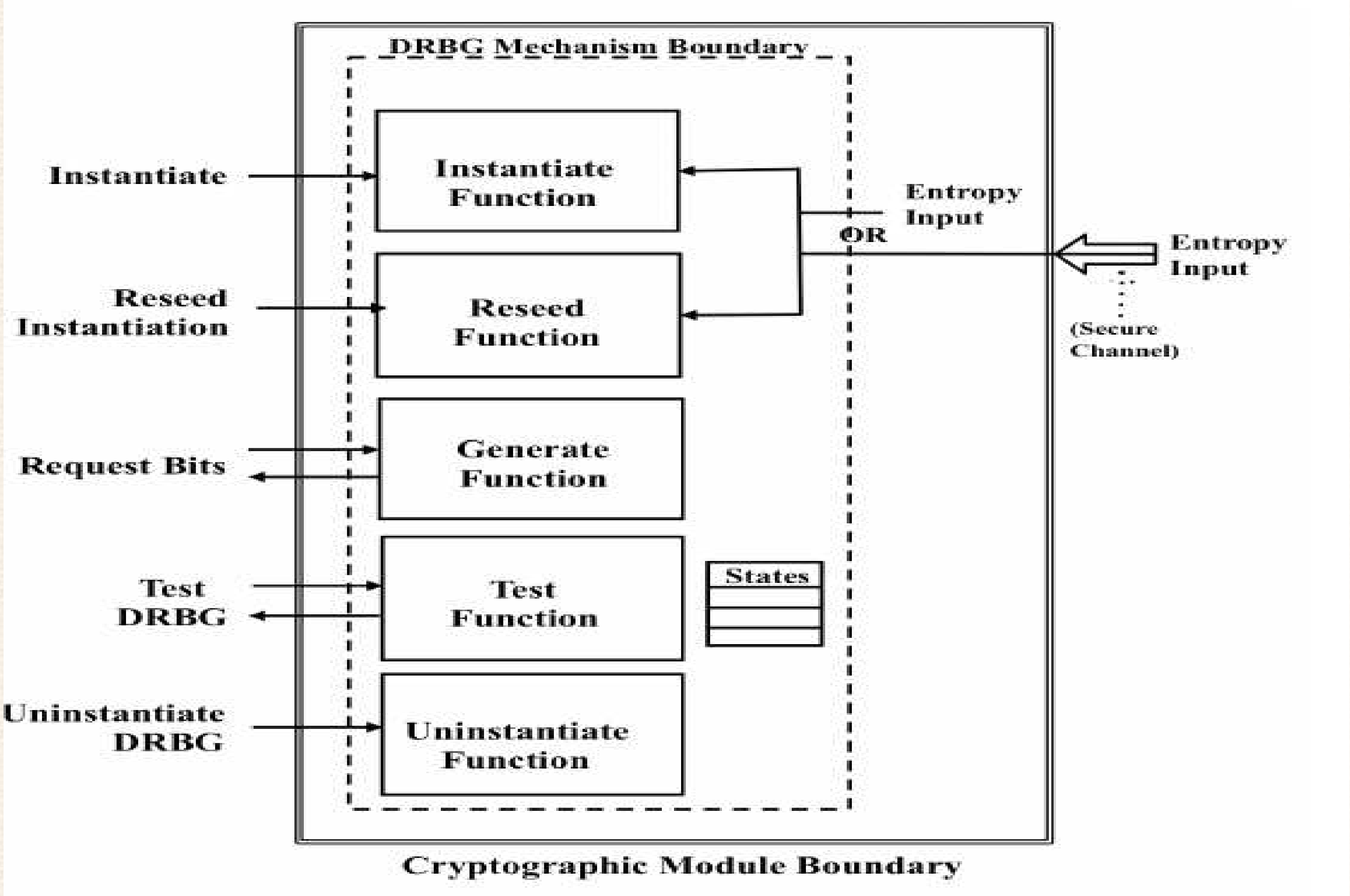
- formally defined as "112, 128, 192, 256 bits", intuition: number of bits to be guessed

Functions executed:

- **instantiate:** initializing the internal state, preparing DRNG to use
- **generate:** generating output bits as DRNG
- **reseed:** combines the internal state with new entropy to change the seed
- **uninstantiate:** erase the internal state, return to factory settings
- **test:** internal tests aimed to detect defects of the chip components

DRBG mechanism boundary:

- DRBG internal state and operation shall only be affected according to the DRBG mechanism specification
- the state exists solely within the DRBG mechanism boundary, it is not accessible from outside
- information about the internal state is possible only via specified output


DRBG Mechanism Boundary
Instantiate
Instantiate Function
Entropy Input
OR
Entropy Input
(Secure Channel)
Reseed Instantiation
Reseed Function
Request Bits
Generate Function
Test DRBG
Test Function
States
Uninstantiate DRBG
Uninstantiate Function
Cryptographic Module Boundary

Seed:

- **entropy** is obligatory, entropy strength should be not smaller than the entropy of the output
- ***approved randomness source*** is obligatory as an entropy source
- **reseeding**: a nonce is not used, the internal state is used
- **nonce**: it is not a secret. Example nonces:
  - a random value from an approved generator
  - a trusted timestamp of sufficient resolution (never use the same timestamp)
  - monotonically increasing sequence number
  - ...

reseed operation:

- “for security”
- argument: it might be better than `uninstantiate` and `instantiate` due to aging of the entropy source
- the main difference: the internal state is used! `instantiate` does not use the state

Hash_DRBG

variants:

- hash algorithms: SHA-1 up to SHA-512 (plug-and-play approach)
- parameters determined, e.g. maximum length of personalization string
- seed length typically 440 (but also 888)

state:

→ value V updated during each call to the DRBG

→ constant C that depends on the seed

→ counter `reseed_counter`: storing the number of requests for pseudorandom bits since new entropy_input was obtained during instantiation or reseeding

instantiation:

```
1. seed_material = entropy_input || nonce || personalization_string
2. seed = Hash_df (seed_material, seedlen)   (hash derivation function)
3. V = seed
4.  C = Hash_df ((0x00 || V), seedlen)
5. Return (V, C, reseed_counter)
```

reseed:

```
1. seed_material = 0x01 || V || entropy_input || additional_input
2. seed = Hash_df (seed_material, seedlen)
3.  V = seed
4. C = Hash_df ((0x00 || V), seedlen)
5. reseed_counter = 1
6. Return (V, C, reseed_counter)
```

generating bits:

```
1. If reseed_counter > reseed_interval, then return "reseed required"
2. If (additional_input $\neq$ Null), then do
   2.1 w = Hash (0x02 || V || additional_input)
   2.2 V = (V + w) mod $2^{\text{seedlen}}$
3. (returned_bits) = Hashgen (requested_number_of_bits, V)
4. H = Hash (0x03 || V)
5. V = (V + H + C + reseed_counter) mod $2^{\text{seedlen}}$
6. reseed_counter = reseed_counter + 1
7. Return (SUCCESS, returned_bits, V, C, reseed_counter)
```

Hashgen:

```
1. $m = \frac{\mathrm{requested - no - of - bits}}{\mathrm{outlen}}$
2. data = V
3. W = Null string
4. For i = 1 to m
    4.1 w = Hash (data).
    4.2 W = W || w
    4.3 data = (data + 1) mod 2seedlen
5. returned_bits = leftmost (W, requested_no_of_bits)
6. Return (returned_bits)
```

Other NIST standard constructions:

i. based on HMAC function

ii. based on block encryption

DUAL EC -standardized backdoor

NIST, ANSI, ISO standard for PRNG, from 2006 till 2014 when finally withdrawn

- problems reported during standardization process: bias finally 2007 a paper of Dan Shumow and Niels Ferguson with an obvious attack based on kleptography (199*)
- DUAL EC dead for crypto community since 2007 but not in industry
  - deal NSA -RSA company (RSA was paid to include DUAL EC)
  - products with FIPS certification had to implement Dual EC, no certificate when P and Q generated by the device
  - generation of own P and Q discouraged by NIST (true: one can make mistakes!)
  - Dual EC used in many libraries: BSAFE, OpenSSL, ...
  - in 2007 an update of Dual EC made the backdoor even more efficient
  - changes in the TCP/IP to ease the attack (increasing the number of consecutive random bits sent in plaintext)

Elliptic curve algebraic group

some details later, but:

- more secure than modular arithmetic $\Rightarrow$ parameters can be smaller for the same computational complexity of breaking
- $\Rightarrow$ time and space complexity practically lower (even if mathematics more complex)
- group elements: points on the plane $\mathbb{F} \times \mathbb{F}$ that satisfy some equality of 3rd degree , where $\mathbb{F}$ is a finite field
- and an abstract point $\mathcal{O}$ (called “point in infinity”)

two rules:

- $-(x, y) = (x, -y)$
- if a line intersects the curve on points $(x, y), (u, w), (s, z)$, then

$$(x, y) + (u, w) + (s, z) = \mathcal{O}$$

- additive notation: $k \cdot (x, y)$ means $(x, y) + \ldots + (x, y)$ (k times)

recall the basic principle:

→ state $s_{i+1} = f(s_i)$, where s_0 is the seed

→ generating bits: $r_i\colon = g(s_i)$

→ both f anf g must be one-way functions in a cryptographic sense

Dual EC, basic version:

→ points P and Q "generated securely" by NSA but information classified,

→ $s_{i+1}\colon = \mathrm{x}(s_i \cdot P)$ (that is, the "x" coordinate of the point on an elliptic curve)

→ $r_i\colon = \mathrm{x}(s_i \cdot Q)$

→ this option used in many libraries

Dual EC with additional input:

→ if additional input given then update is slightly different:

→ $t_i := s_i \oplus H(\mathrm{addtional\_input}_i), \quad s_{i+1} := x(t_i \cdot P)$

Attack: with a backdoor d, where $P = d \cdot Q$

for basic version:

→ from r_i reconstruct the EC point R_i (immediate by Elliptic Curve arithmetic , two solutions)

→ compute s_{i+1} as $x(d \cdot R_i)$ (no need to know the internal state s_i !)

Dual EC with additional input, attack:

- it does not work in this way since the $\oplus$ operation is algebraically incompatible with scalar multiplication of elliptic curve point
- it does not help much:
  - if more than one block r_i is needed by the consuming application, then the next step(s) is executed without additional input ...
  - ... and at this moment the adversary learns the internal state

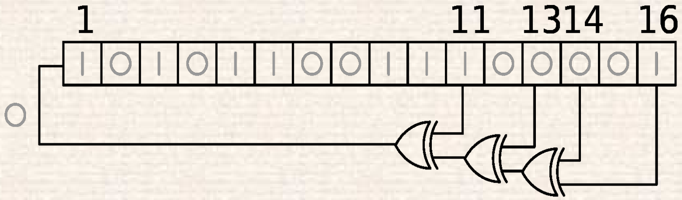
Simple hardware generators : LFSR ...

linear feedback shift register

- state: $b_0, b_1, b_2, \ldots, b_n$
- generate: output b_n
- transition:

  i. $d := \sum_{i=1}^{n} \alpha_i \cdot b_i \mod 2$ (where a few α's are 1, the rest is 0)

  ii. rightshift: $(b_0, b_1, b_2, \ldots, b_n) := (d, b_0, b_1, \ldots, b_{n-1})$



(Wikipedia)

Advantages: extremely fast and cheap if implemented in hardware,

if α's well chosen (correspond to some irreducible polynomial), then the period is maximal $2^l - 1$

Disadvantage:

linear algebra, weak in cryptographic sense, state can be easily recovered

Attempts to fix the problem:

- instead of $\sum \mod 2$ some nonlinear function
- output: $F(\mathtt{output}(\mathrm{LFSR}_1), \mathtt{output}(\mathrm{LFSR}_2), \mathtt{output}(\mathrm{LFSR}_3))$

Krawczyk's shrinking generator:

- two sequences generated $a = (a_{0,}a_1, a_2, \ldots)$ and $b = (b_{0,}b_1, b_2, \ldots)$ obtained from LFSR
- the output consists of b except for bits dropped:

  b_i dropped iff $a_i = 0$

Stream ciphers

random number generators come together with construction of stream ciphers:

$$\text{ciphertext} := \text{plaintext} \oplus \texttt{random}(\text{Key})$$

example: ChaCha

True Random Generators

- problem of bias, dependancies etc – apply Hash to it:

$$\mathrm{output} = \mathrm{Hash}(\mathrm{TRNG}())$$

- **problem of influencing the generator** via environment conditions (laser, temperature, radiation, ...
- **how do you know in what physical shape is the generator?**

  PRNG can be tested cryptographically,

  for TRNG it is hardly possible, except when it is evidently broken

- **maybe a fake**? no expensive TRNG inside but a cheap LFSR? You cannot check it...