Sicherheit in Kommunikationsnetzen
(Network Security)

Random Numbers

Dr.-Ing. Matthäus Wander

Universität Duisburg-Essen

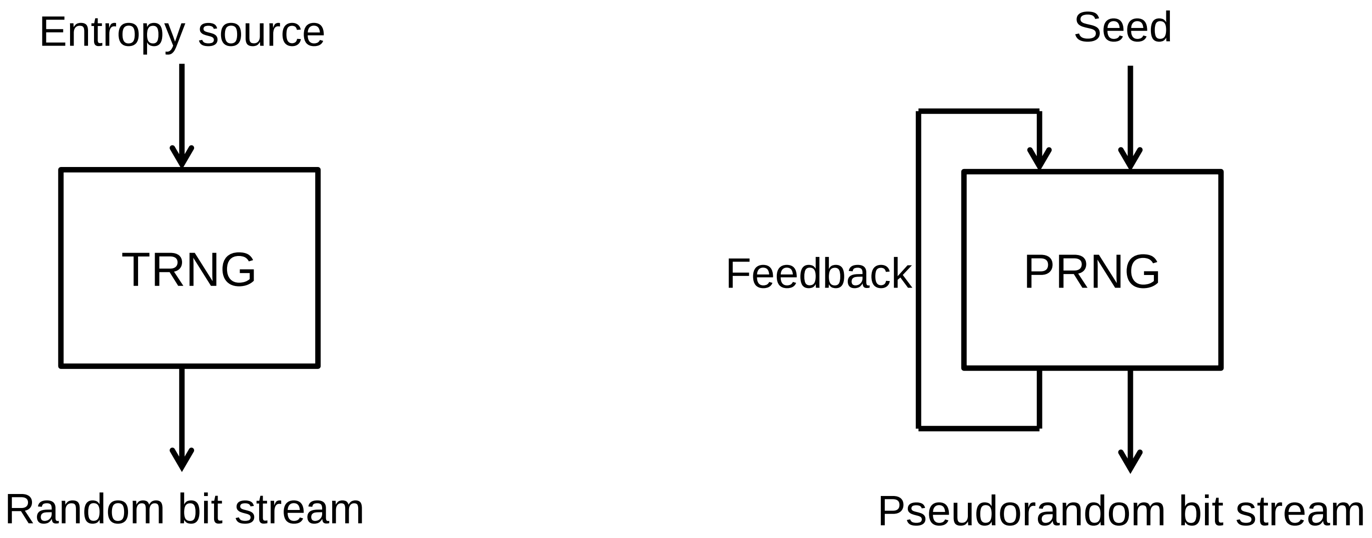
Motivation

- We need random numbers in cryptography
- Example: key generation
  - e.g. unpredictable key for one-time pad
  - e.g. random primes for RSA
- Example: initial value/initialization vector
  - e.g. in CBC or GCM mode
- Problem: computers are built for deterministic computation, not random results
  - Difficult to generate true randomness

Random Number Generator

- Definition: a random bit generator (RBG) or random number generator (RNG) is a device or algorithm that generates random numbers
- The major challenge is to generate a random sequence of bits $b \in \{0, 1\}$
  - From that, we can derive any random number
- Generate random number $r \in \mathbb{Z}$ with $0 \le r \le n$
  1. Generate bit sequence of length $\lfloor \log_2(n) \rfloor + 1$
  2. Convert bits to non-negative integer r
  3. If $r > n$, discard r and repeat from step 1

Random Number Generator (2)



Entropy source

TRNG

Random bit stream

Seed

Feedback

PRNG

Pseudorandom bit stream

True Random Number Generator

- Definition: a true random number generator (TRNG) is an RNG with the following properties:

1. Unpredictability: given a subsequence of generated numbers, one cannot infer another number from the sequence
   - If first n bits known, one cannot predict bit n+1
2. Uniform distribution: distribution of generated numbers in the sequence is uniform
   - Bit values 0 and 1 occur with ½ probability each

True Random Number Generator (2)

- We cannot generate randomness with any deterministic algorithm
- We need an entropy source
  - Entropy: amount of information without redundancy
  - Term borrowed from information theory
  - In cryptographic context: amount of randomness
- Types of entropy sources
  - Hardware-based: external device
  - Software-based: utilize events visible on a computer

Hardware-based Entropy Source

- Generate bits based on physical phenomena
  - Unpredictable events due to our state of knowledge
  - Whether they are truly random is subject to physical models and philosophical discussion
- Examples
  - Radioactive decay: time between emission of particles
  - Thermal noise from semiconductor diode or resistor
  - Atmospheric noise detected by radio receiver
  - Fluctuation in disk drive access due to air turbulence
  - Ambient sound recorded by a microphone

Software-based Entropy Source

- Generate bits from events readable by software
  - System clock or clock drift
  - Time between use key strokes or mouse movement
  - Network packet inter-arrival time
  - Operating system values such as system load or statistics for hard disk access
- Note: entropy sources become confidential data
  - Operating systems usually do not treat them as such
- Increase entropy by mixing multiple sources

De-Skewing

- Some of these phenomena produce uncorrelated but skewed bit sequences
    - i.e. non-uniform distribution of bit values {0, 1}
- De-skew the bit sequence, for example:
    - Read pairs of bits, remove „00“ and „11“ pairs
    - Replace „01“→0 and „10“→1
- Result: uniform distribution of {0, 1}
    - Simple algorithm but we discard 75% of input bits
- Alternative: apply cryptographic hash function

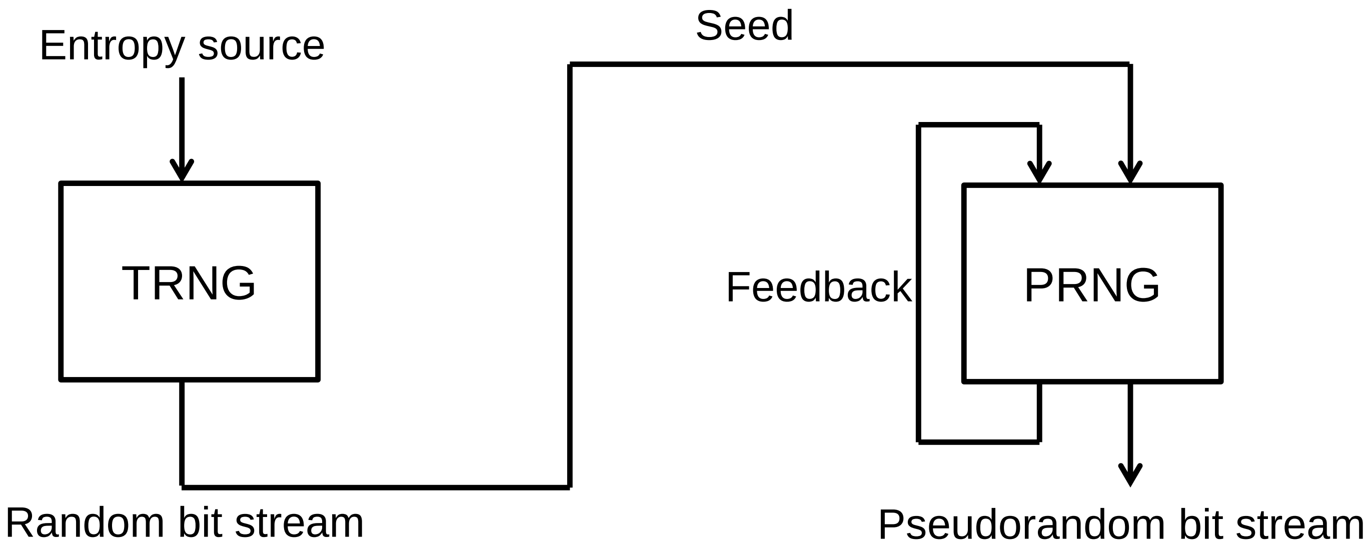
Random Integers within a Range

- Given an RNG that generates integers $0 \leq r \leq n$
- What if you need an integer $m < n$?
  - e.g. random() generates 8-bit numbers $0 \leq r \leq 255$
  - Your application needs $0 \leq r \leq 9$
- Idea: r = random() MOD 10
  - Problem: introduces bias for 0 to n−m MOD m
- Instead:
  - Divide interval [0, n] into subintervalls of equal size
  - Discard numbers that lie outside of subintervalls

Pseudorandom Number Generator

- Definition: a pseudorandom number generator (PRNG) is a deterministic algorithm that:
- Input: receives k truly random bits
  - This (short) input is called seed
- Output: produces a long binary sequence that appears random
  - The output is not truly random but derived deterministically from the seed
  - Thus pseudorandom

Pseudorandom Number Generator (2)



- True RNG are slow (subject to entropy source)
- Use True RNG to seed a Pseudo RNG
  - Produces quickly a pseudorandom bit stream

Linear Congruential Generator

- A Linear Congruential Generator (LCG) is a PRNG that produces a sequence of numbers with:
- $y_{i+1} = (a \times y_i + b) \text{ MOD } q$
  - a, b and q are integer constants
  - y_0 is the seed
- Example: $y_{i+1} = (1103515245 \times y_i + 12345) \text{ MOD } 2^{31}$
  - Used in rand() function in ANSI C
  - Good distribution of output numbers
- But: predictable output and thus insecure

Cryptographically Secure PRNG

- Definition: next-bit test
  - A PRNG passes the next-bit test if there is no polynomial-time algorithm that predicts the n+1 bit from n known bits with a probability of >50%
  - i.e. it is infeasible to predict the next bit
- Definition: a cryptographically secure pseudorandom number generator (CSPRNG) is a PRNG that passes the next-bit test
  - Statistical tests cannot distinguish the output of a CSPRNG from a TRNG
  - Uniform distribution and practically unpredictable

Statistical Tests for Randomness

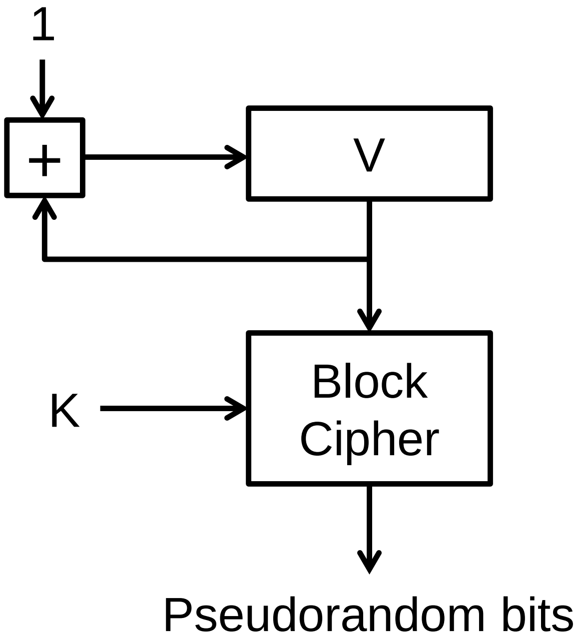
- Test randomness of PRNG e.g. with:
  - Monobit test: same number of 1 and 0 bits?
  - Serial test (two-bist test): same number of 00, 01, 10 and 11 pairs?
  - Runs test: is the number of runs (sequences of only either 0 or 1) for various lengths as we would expect for random numbers?
  - Maurer’s universal test: can we compress the sequence without loss of information?
- Note: passing statistical tests gains confidence, but does not guarantee to pass the next-bit test

Construction of CSPRNG

- PRNG that are not cryptographically secure
  - Linear Congruential Generator (LCG)
  - Linear Feedback Shift Register (LFSR)
- Stream ciphers are basically CSPRNG
  - Usually very fast, but with rather thin security margin
- Generic schemes for constructing CSPRNG
  - Based on block ciphers
  - Based on cryptographic hash/MAC functions
  - Based on problems from asymmetric cryptography

Block Ciphers in CTR/OFB Mode

- Seed consists of key K and value V



1

+

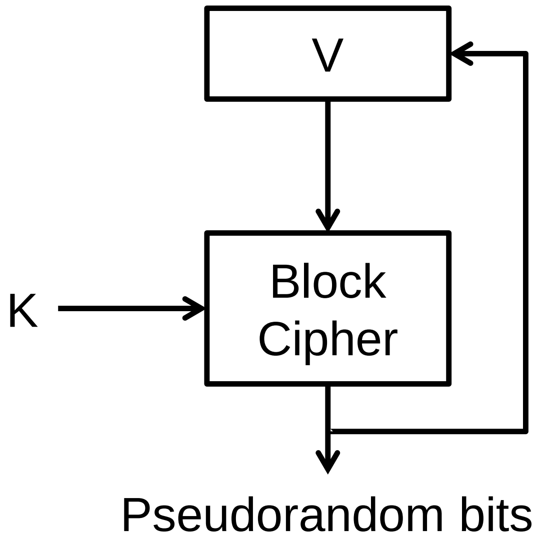
V

K

Block Cipher

Pseudorandom bits

CTR mode



V

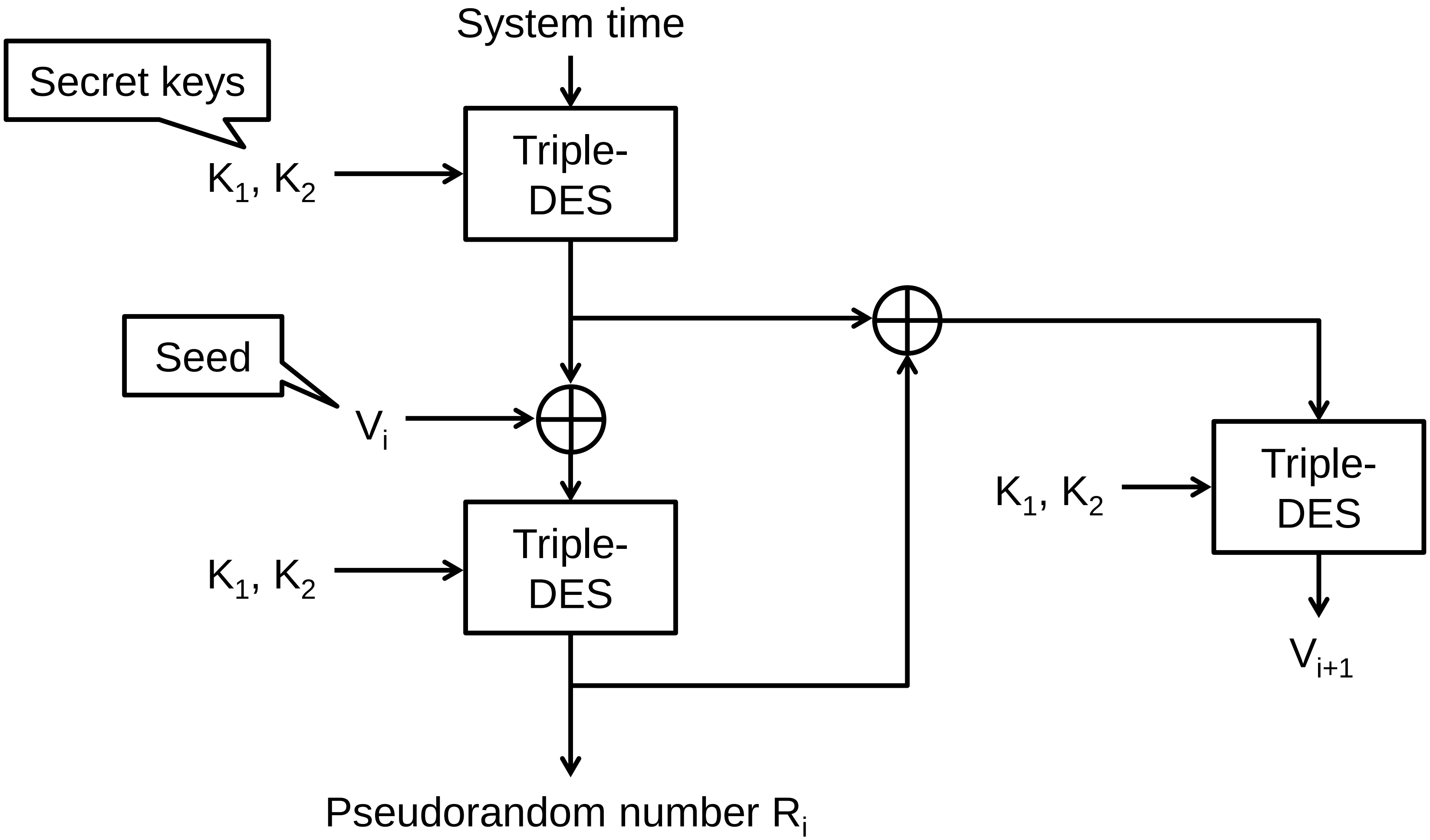
K

Block Cipher

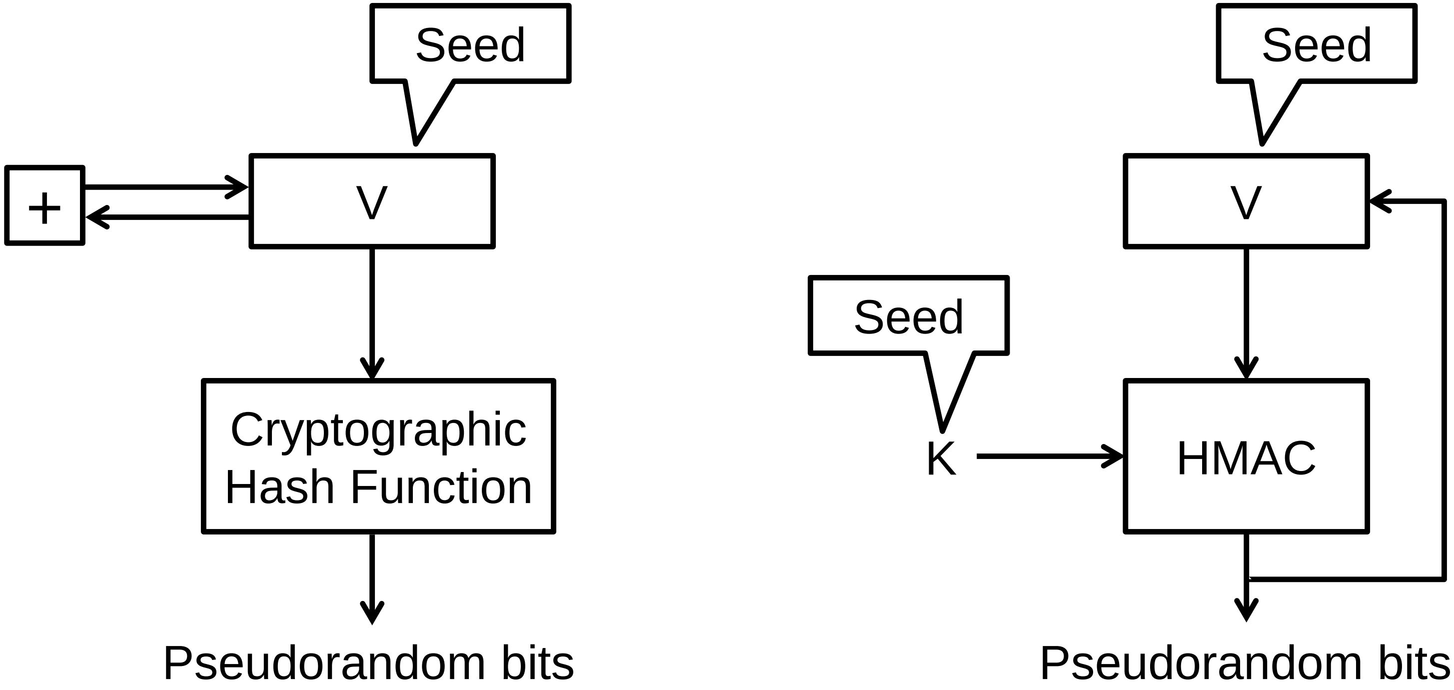
Pseudorandom bits

OFB mode

Block Ciphers with ANSI X9.17



Hash/MAC Functions



Seed

+

V

Cryptographic Hash Function

Pseudorandom bits

Seed

V

Seed

K

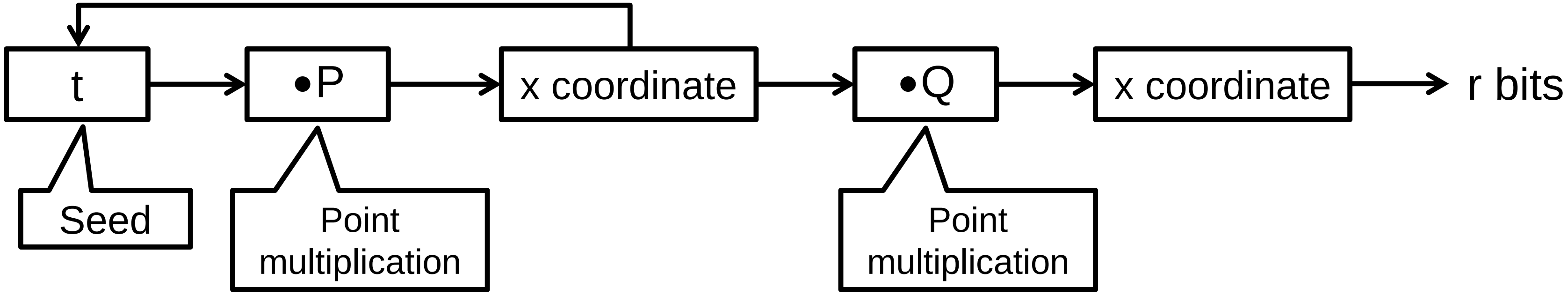
HMAC

Pseudorandom bits

RSA Generator

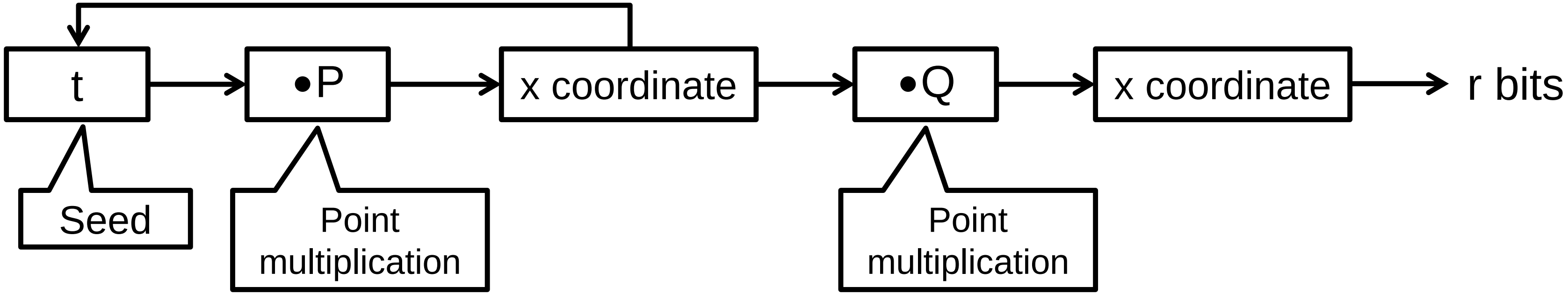
- Generate RSA public key: e, n
  - Private key can be discarded
- Choose integer y_0 as seed
- RSA encrypt: $y_i = (y_{i-1})^e \bmod n$
- Output least significant bit: $z_i = y_i \,\&\, 1$
- Repeat on next y_i to get $z_1, z_2, \ldots, z_n$
- Security based on RSA integer factorization
  - Very inefficient: one RSA encryption per random bit

Dual EC DRBG



- P, Q are constant points on an elliptic curve
- t is a scalar, initialized with seed
- Point multiplication over elliptic curves
  - Security based on elliptic curve discrete logarithm
  - Result is another point on curve, we use x-coordinate

Dual EC DRBG (2)

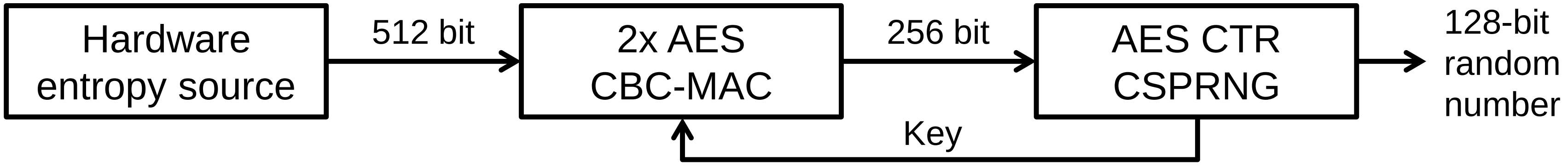


- Secure if P and Q are independent
  - Let $P=e \bullet Q$ for a secret e and corresponding e^{-1}
  - Attacker can derive internal state from output bits
- Published by NIST with constant P, Q values
  - Theory: predictable RNG due to NSA back door
  - NIST standard withdrawn in 2014

RNG in Operating Systems

- RNG provided by operating system or standard library of programming languages
  - Fast generation of numbers
  - Uniform distribution
- But: usually not cryptographically secure
  - Do not rely on random() for cryptographic purposes!
- CSPRNG under Linux
  - /dev/random: read blocks once entropy is depleted
  - /dev/urandom: read never blocks, seeded PRNG from same entropy pool like /dev/random

Example: Intel Digital RNG



- Hardware RNG built in Intel Ivy Bridge CPUs
  - Thermal noise from two inverters (NOT gates)
- Hardware entropy source fed into AES CBC-MAC
  - Removes skew or bias of entropy source
  - CBC-MAC output fed into an AES-based CSPRNG
- RDRAND instruction returns 128-bit number
  - Secure if implemented correctly

Example: Intel Digital RNG (2)

- Problem: we cannot look into the CPU hardware
  - Thus security audit is impossible
  - We can test whether the output passes statistical tests
  - We don‘t know whether there are back doors that allow to recover or tamper with random numbers
- Linux uses RDRAND as entropy source mixed with software entropy sources
  - If RDRAND is bad, it won‘t increase entropy
  - Careful mixing required, otherwise a malicious entropy source could cancel out other entropy sources

Conclusions

- Generating truly random numbers is hard
  - Hardware and software-based entropy sources
  - Uniform distribution of numbers desired
  - Impossible to predict output bits
- Initialize pseudorandom RNG with random seed
- Not all PRNG suitable for cryptography
- Cryptographically secure PRNG implementations use block ciphers or hash functions in practice
  - Infeasible to predict output bits