PV181 Laboratory of security and applied cryptography

Random values and Random Number Generators

Marek Sýs
sysox@mail.muni.cz, A405

CROCS
Centre for Research on Cryptography and Security

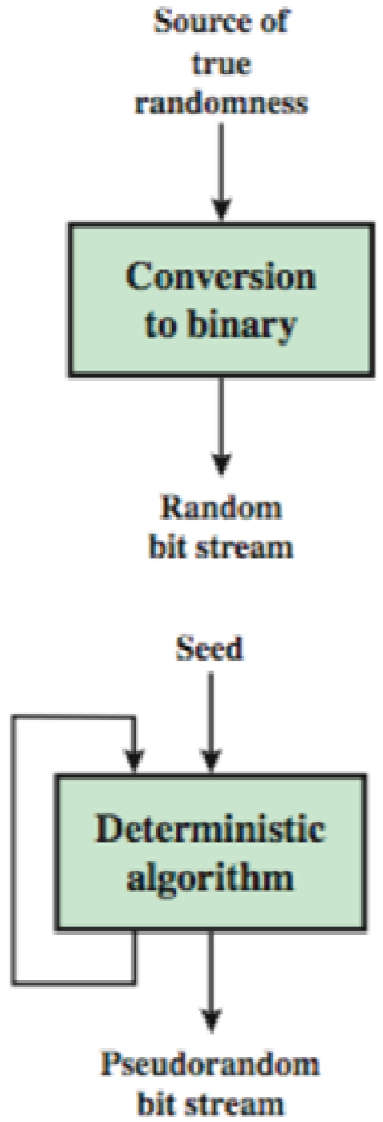
You will learn

- What types of RNG you can find in libraries.
- What RNGs are (in)apropriate for crypto.
- bitwise operations (heavily used in crypto).
- How to improve randomness of RNG output.
  - using hash function and bitwise XOR
- How to generate secure random values:
  - in ***python***, *C*, *C++*
- Why standard **rand()** and others (e.g. Mersenne Twister) are insecure.

RNG types

1. True random (TRNG)
   - Source: physical device (noise) radio decay, thermal noise, …
   - non-deterministic, aperiodic, slow

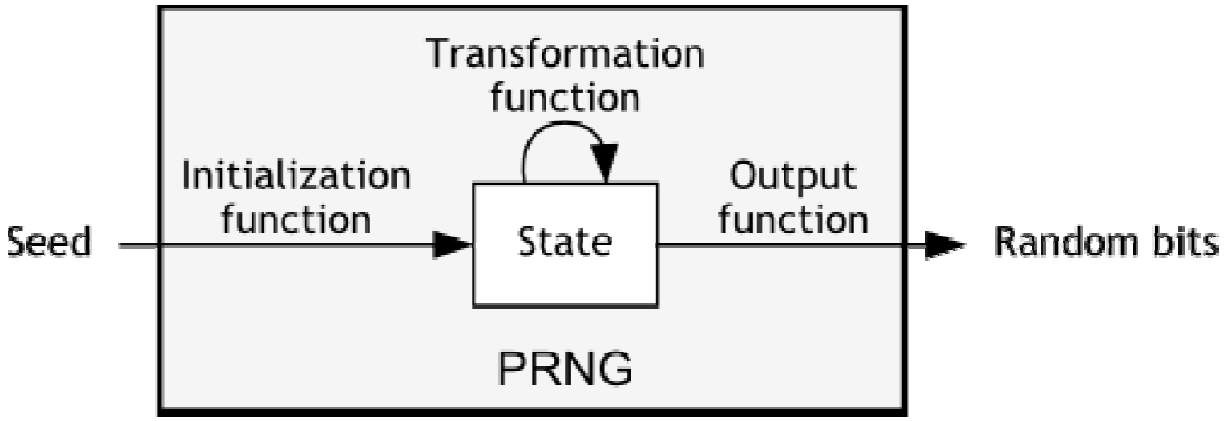
1. Pseudo random (PRNG)
   - Source: software function
   - deterministic, periodic, very fast



PRNG

defined by 3 functions: Init, Transform, Output

State = Init(Seed)
State = Trans(State)
rnd = Out(State)



Cryptographically secure (CSPRNG) - generated rnd values give no information about **next** or **previous** rnd values ⇒ no info about Seed, State

Standard library functions

ANSI C(rand), Java(java.util.random),... - uses fast but **very insecure** LCG generator

Linear Congruential Generator(LCG)

- $s_{n+1}=a*s_n+b \bmod m$ (fixed constants a,b,c)

Out is identity (**id**) func. i.e., generated *rnd*=*State*

⇒ next rnd values easily computed

Trans is linear: $f(x) = ax+b \bmod m$

⇒ previous rnd values can be computed easily

Example
ANSI C portable functions

```
static unsigned long int next = 1;

int rand(void)   //  RAND_MAX assumed to be 32767
{
      next = next * 1103515245 + 12345;
      return (unsigned int)(next/65536) % 32768;
}

void srand(unsigned int seed)
{
      next = seed;
}
```

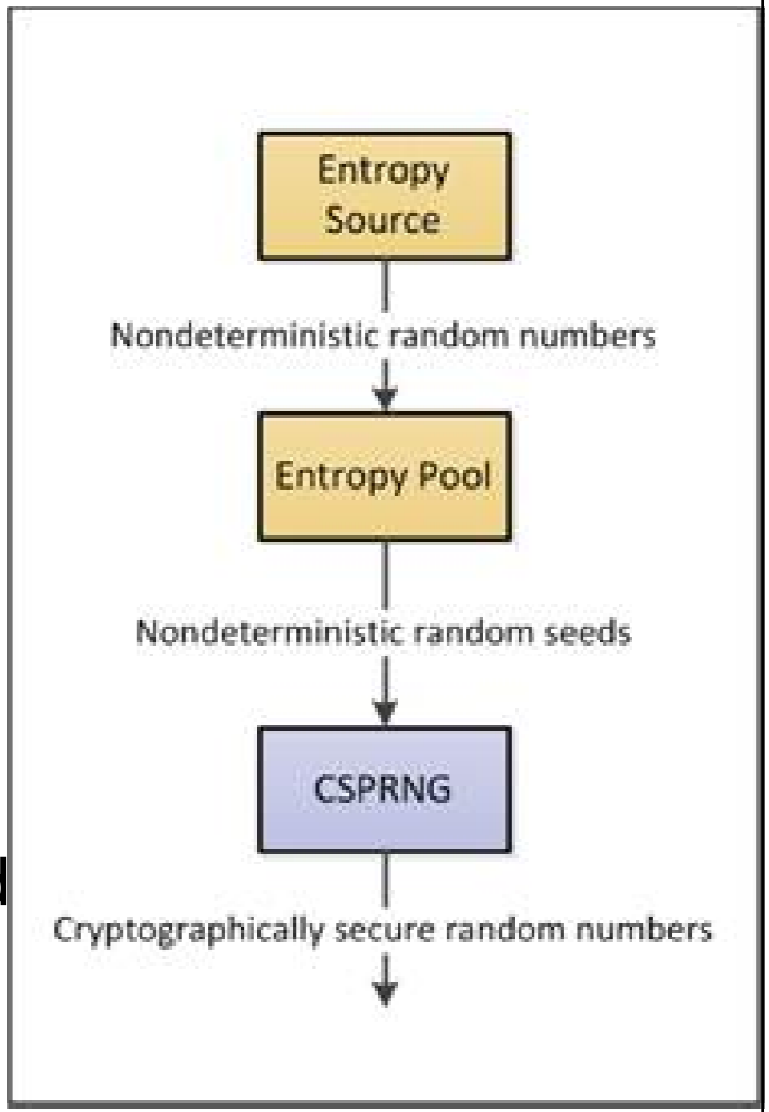
Practice

PRNG values:

- CSPRNG seeded by TRNG

TRNG (entropy source):

- typically combined internally with PRNG
- output stored in “entropy pool”
  - depends on all previous generated rnd values (chaining of values, not replacement)



TRNG

Linux: two entropy pools (files) *dev/(u)random*

- keyboard timings, mouse movements, IDE timings

Windows: similar to Linux

- binary register *HKEY_LOCAL_MACHINE\SYSTEM\RNG\Seed*

Additional entropy sources (if available):

- TPM, RNRAND instruction, hardware system clock (RTC), Interrupt timings, havege daemon

Weak generators

Python random() - “Mersenne Twister(MT) as the core generator. It produces 53-bit precision floats and has a period of 2**19937-1”

C rand(): LCG generators (+ some tweaks)

- glibc (used by GCC) rand() - LCG and “linear additive feedback” (r[i] = r[i-31] + r[i-3])

C++: LCG or MT or Lagged fibonacci

- minstd_rand(0 or 1), mt19937(_64),

Unix infrastructure

Special files - reading files provides random data

- /dev/random
  - always produces entropy but,
  - blocking - can block the caller until entropy available (entropy estimation)
- /dev/urandom
  - amount of entropy not quaranteed
  - always returns quickly (non blocking)

Usage:

- /dev/urandom preferred, only shortly after boot use /dev/random (see Myths about dev/urandom)

Linux RNG design

- 3 entropy pools (store random data)
  - can be viewed as PRNG - “Init” func **mixes** (using SHA1) input rnd data to the state ⇒ state depends input data and **all** previous states!!
- input_pool (state of 4096 bits)
  - accumulate (collects, compress) the entropy from hardware events to the state
  - feeds exclusively (no access to this pool)
    - blocking_pool (state of 1024 bytes)
    - non-blocking_pool (ChaCha20 stream cipher)
      - only key (256) is fed by true rnd values
    - state (“seed” for other pools) is saved at shutdown

See Gauvrit’s blog with nice scheme

Unix infrastructure

Operations on files:

- to get entropy just open and read from the file
  - use read(2) but always check if returned value == requested number of bytes (reading can be interrupted!!!)
- It is also possible to write to **/dev/random**
  - privileged (harmless) user can mix random data into the pool - entropy is increased (but not entropy counter)
- information about the pool: **proc/sys/random/***

Unix: methods and quality

Good sources(C):

- direct read from initialized random/urandom
- getrandom() + flags:
  - source: random or urandom
  - blocking or non-blocking (also blocks until initialised)
- get_random_bytes() - kernel space
- similar in Python: os.urandom(), os.getrandom(), secrets.token_bytes()

Weak sources:

- rand, time(rdtsc instruction, clock func,...), uninitialized urandom

Practice

1. Go to https://mybinder.org
2. Copy link https://github.com/sysox/PV181_RNG/ to Github field, press launch
3. Start with PV181_RNG_python.ipynb
4. Then PV181_RNG_C.ipynb
5. Write down the answers to Questions - they will be discussed at the end of seminar.