Advanced cryptology

Random Number Generators

prof. Ing. Róbert Lórencz, CSc., Ing. Josef Hlaváč, Ph.D.

České vysoké učení technické v Praze, Fakulta informačních technologií
Katedra počítačových systémů

Příprava studijních programů Informatika pro novou fakultu ČVUT je spolufinancována Evropským sociálním fondem a rozpočtem Hlavního města Prahy v rámci Operačního programu Praha — adaptabilita (OPPA) projektem CZ.2.17/3.1.00/31952 – „Příprava a zavedení nových studijních programů Informatika na ČVUT v Praze“.
Praha & EU: Investujeme do vaší budoucnosti

Tato přednáška byla rovněž podpořena z prostředků projektu č. 347/2013/B1a Fondu rozvoje vysokých škol Ministerstva školství, mládeže a tělovýchovy

Contents of lectures

- Random numbers in cryptography
- Entropy
- Pseudo-random generators
- True random generators
- Testing randomness

Random numbers in cryptography - Motivation (1)

A lot of cryptographic applications require random numbers

- Generating cryptographic keys
- Generating numbers *nonce*, *salt*, *padding*
- Vernam cipher (*one-time pad*)

Required „quality“ of randomness at various different applications

- *Nonce* in some protocols is sufficient when unique
- Key generation requires higher quality
- Unbreakability of the Vernam cipher is guaranteed only if the key was obtained from a true random source with high entropy

Random numbers in cryptography - Motivation (2)

The basic principle:

„Cryptosystem is only as strong as its weakest point.“

Corollary:

Incorrectly designed or incorrectly used random number generator can pose a fatal weakness of the whole cryptosystem.

Random numbers in cryptography – Definition (1)

Random number is a number generated by a process that has an unpredictable result and its course can not be exactly reproduced. This process is called *RNG* (RNG - Random Number Generator).

For one number we cannot discuss whether or not it is generated by a random number generator. So we always work with *sequences of random numbers* or also *a random sequence*.

The numbers are represented in the computer using bits. Instead of random numbers therefore we often work with *random* bits respectively *with random bit generators* and *random bit sequences (strings)*.

Random numbers in cryptography – Definition (2)

From random sequences we expect *good statistical properties*:

- Uniform distribution - all values are generated with the same probability
- The individual generated values are *independent* - there is no correlation between them

The important term is *entropy*.

Entropy (1)

Value of *entropy* describes the degree of randomness - how difficult is a value (random number, random sequence, a string of random bits) to estimate.

Entropy is a quantity of uncertainty or unpredictability of a value and depends on the probabilities of possible results of a process that generates it.
The entropy is given by the relation

$$H(X) = -\sum_{i-1}^{n} p_i \log_2 p_i \tag{1}$$

where X is the generated value (eg. random bit string of a given length) and $p_1, \ldots, p_n$ are probabilities of all values of $X_1, \ldots, X_n$ that can be possibly generated by the generator.

Entropy (2)

Entropy always relates to the attacker and his ability (or inability) to predict the generated value. If an attacker knows the following generated value with certainty, the entropy is zero (and zero is the security of application that uses the generated random number).

It can also be said that entropy is the average number of bits necessary for encode a value using the optimal coding, or that entropy expresses the amount of information measured in bits.

When is the entropy of the random bits maximal?

Entropy of generator is the maximal, if for a given length (number of bits), all possible sequences are generated, each with equal probability.

Pseudorandom generators - Introduction

Computers work deterministic $\rightarrow$ how to generate random numbers?

Pseudorandom number generator (PRNG): The algorithm, whose output is a sequence, which although actually *is not* random but that *seems* to be random, if the attacker does not know any parameters of the generator.

- Algorithmic $\Rightarrow$ easily realizable
- Usually fast
- As a rule, they have good statistical properties
- But: the output is predictable

Linear congruential generator (1)

One of the oldest and most popular ways to generate pseudorandom numbers is *linear congruential generator*:

$$X_{n+1} = (aX_n + c) \bmod m \tag{2}$$

where X is a sequence of pseudorandom numbers, $m > 0$ is the modulus (often a power of two), a is a multiplier, c is the increment and X_0 is the initial value (*seed*).

The pseudo-random sequence X repeats after a maximum of m iterations. This maximum is achieved if all the following conditions are fulfilled:

- Numbers c and m are relatively prime
- $a - 1$ is divisible by all prime factors of m
- If $4|m$, then also $4|a - 1$

Linear congruential generator (2)

The quality of the generator is heavily dependent on the choice of parameters m, a, c. However, from cryptology point of view this is a very problematic RNG:

- If the generated numbers are used as the coordinates of points in n - dimensional space, then the resulting points will lie in a maximum of $m^{\frac{1}{n}}$ hyperplanes
- When m is a power of two, the low bits of X have a much shorter period than the entire sequence. The generator for example produces alternately odd and even numbers.
  (Note. Therefore from the values X_n we usually choose only some bits.)

Linear congruential generator *is not cryptographically secure*.

Cryptographically secure PRNG (1)

Requirements for *cryptographically secure* pseudorandom generators:

- „*Next-bit test*“: If the first k-bit random sequence is known, there does not exist any polynomial algorithm that could predict $(k+1)^{\text{th}}$ bit with probability of success greater than $1/2$.
- „*State compromise*“: Even if the internal state of the generator is detected (whether whole or in in part) one cannot retroactively reconstruct the current generated random sequence. Moreover, if additional entropy enters the generator at run time, it should not be possible from the knowledge of inner state to predict the internal state of the following iterations.

Most used PRNG fulfills these requirements only under certain conditions.

Cryptographically secure PRNG (2)

Examples, how to construct cryptographically secure PRNGs:

- Secure block cipher in counter mode: Choose a random key (*seed*) and the initial counter value i. Encrypt sequential numbers i, $i + 1$, etc. Obviously, the period for an n-bit block cipher is 2^n and the key and initial value must not be revealed.
- Cryptographically secure hash function applied to a counter: Choose a random initial counter value i. Hash separately the values i, $i + 1$, etc. Again, the counter value must not be revealed.
- Stream ciphers are basically PRNGs whose output is used to XOR the plaintext.
- Number theory based algorithms that were proven to be secure (at least in some way).

Blum-Blum-Shub PRNG (1)

Example of a PRNG that is considered cryptographically secure, is the Blum-Blum-Shub algorithm:

$$X_{n+1} = X_n^2 \bmod m \tag{3}$$

Modulus $m = pq$ is a product of two large primes p a q. Initial element (*seed*) is $X_0 > 1$. It should hold that $p, q \equiv 3 \pmod 4$, and $\gcd(\phi(p-1), \phi(q-1))$ should be small.

The output is usually not the raw value X_n, but its parity or some least significant bits.

Blum-Blum-Shub PRNG (2)

Pseudorandom generator Blum-Blum-Shub:

- Slow
- Relatively strong proof of security (relating to the problem of integer factorization)
- Able to compute the i-th element directly:

$$X_i = (X_0^{2^i \bmod (p-1)(q-1)}) \bmod m \tag{4}$$

Cryptographically secure PRNG (3)

Mentioned PRNG require random and secret input, *seed*:

- We still need some *true* randomness
- The quality of PRNG is deived also from the quality of generating the *seed*

Entropy of the PRNG output: given by the entropy of its input (*seed*), **an algorithm by itself can never increase entropy**.

True random generators (1)

Common property of cryptographically secure PRNG: Cannot work without parameters (esp. *seed*), which must be chosen randomly. PRNGs by themselves are not enough for cryptograpy; we must be able to generate truly random values (bits).

True Random Number Generators (TRNG) use an *entropy source*, usually a physical effect or outer influence. For example:

- Radioactive decay (the HotBits project)
- Atmospheric noise (random.org)
- Thermal noise, e.g. on analog parts
- User behavior (mouse movement, keystroke timing)
- ...

True random generators (1)

True random number generator properties:

- Ouptut is unpredictable, even if we know all parameters
- Output has usually worse statistical properties $\rightarrow$ postoprocessing is necessary
- Implementation is more complex, often requires dedicated hardware
- Entropy source must be continually tested, its properties may degrade with time

True random generators (2)

Postprocessing (*post-processing*) aims to improve statistical properties of the TRNG, namely:

- Removing the imbalance of ones and zeros (*bias*) and ensuring uniform distribution
- Entropy extraction – elevating the entropy of output bits at the cost of lowering the generating speed (*bitrate*)

True random generators (3)

John von Neumann corrector (decorelator) is able to eliminate bias and lower correlation of the output. Bits are taken in pairs.
Output is as follows:

Input	**Output**
00, 11	– (input is discarded)
01	0
10	1

True random generators (4)

More possibilities to improve statistical properties of TRNG output:

- Output of TRNG is XORed with a cryptographically secure PRNG
- Merging (XOR) outputs of two or more different TRNG („*software whitening*“)
- Hashing output of TRNG by a cryptographically secure hash function

Random generator testing (1)

To verify the properties of random generators, statistical tests are used. The tests verify if the generated sequence fulfills some properties of random sequence.

Beware: Statistical tests can indicate that a particular generator is probably NOT good, but they cannot prove that it IS. Even if a generator passes all tests, it is still possible, that it contains a weakness that was not discovered (due to the nature of the tests).

Statistical tests are based on *statistical hypothesis testing* on a certain *significance level*. Null hypothesis is that the tested sequence is random. Test return a *p-value*, that expresses the strengh of presumption against the null hypothesis. If the *p-value* goes under a certain level, we consider the null hypothesis invalid.

Random generator testing (2)

Randomness test examples:

- Frequency test – tests if the sequence contains approximately the same number of zeros and ones. Tested are the whole sequence and partial subsequences.
- „Runs“ test – tests if the count and length of strings of same bits (all 1s or all 0s) in the tested sequence corresponds to a random sequence. Again the whole sequence, as well as partial subsequences are tested.
- Matrix rank test – focuses of ranks of disjunct submatrices, the aim is to discover linear dependence of a fixed length subsequence.
- Spectral test – discrete Fourier transform, aims to discover periodicity

Random generator testing (3)

- Maurer's universal statistical test – tests if the sequence can be significantly compressed without loss of information. Significantly compressible sequence does not contain enough information.
- „Monkey" test – The generator is used to generate „letters" of an „alphabet" and the output is tested how often certain predetermined „words" occur. The name comes from the adage about thousand monkeys at a thousand typewriters.
- Tests based on the birthday paradox. Ex. the „Birthday Spacing" test – random points on a large interval are chosen, the spaces between them should asymptotically follow the Poisson distribution.

Random generator testing (4)

Mentioned tests are usually part of known and used sets („bateries“) of tests.

Known test sets:

- Diehard (George Marsaglia): 12 different tests, relatively strong
- Dieharder (Robert G. Brown): re-implementation of Diehard tests according to their description, more tests added
- NIST (National Institute of Standards and Technology): 16 tests

These sets were however developed mainly to test PRNGs. When testing TRNG, it is necessary to deeply analyze the entropy source and design and implement targeted tests that could discover possible weaknesses specific for the entropy source.

Attack examples

Example of a real cryptanalytical attack using weaknesses of a random generator: contactless RFID cards with the Mifare Classic chip

- Used encryption algorithm (Crypto-1) was not publicly known („*security by obscurity*“ – never works). It was discovered to be weak.
- Authentication protocol is *challenge-response*. Uses a „randomly“ generated value *nonce*.
- Random generator is realized by plain counting of clock pulses from the inserting of the card into the field of the reader (= power on).
- If the reader sends the authentication command always at the same time from power on, *nonce will be always the same* (!) – and the state space for brute-force search is substatially reduced (the key can be discovered quickly).