Pseudorandom numbers generators

Alexander Shevtsov

May 17, 2016

Contents

1 Introduction

The need of random numbers occurs in different fields, for example in statistics, in cryptography, in computational mathematics[1].

We won't go into a detailed discussion of what randomness really is. An informal discussion suffices for our purposes. A good informal definition is that random data is unpredictable. A bit later we will state formal properties of a cryptographically secure pseudorandom number generator.

Let's find out the difference between random data and pseudorandom data.

1.1 Real random data.

Real random data can be obtained by measuring some physical characteristic of a physical process. The best means of obtaining unpredictable random numbers are done by measuring physical phenomena such as radioactive decay, thermal noise in semiconductors, sound samples taken in a noisy environment, and even digitised images of a lava lamp. However, few computers (or users) have access to the kind of specialised hardware required for these sources, and must rely on other means of obtaining random data.

Modern approaches which don't rely on special hardware have ranged from precise timing measurements of the effects of air turbulence on the movement of hard drive heads [8], timing of keystrokes as the user enters a password, timing of memory accesses under artificially-induced thrashing conditions, and measurement of timing skew between two system timers (generally a hardware and a software timer, with the skew being affected by the 3-degree background radiation of interrupts and other system activity). In addition a number of documents exist which provide general advice on using and choosing random number sources [9].

Consider an ideal coin toss, its result can be interpreted as a random bit of data and coin tossing as a random number generator. One could say, that if we knew all the forces at the moment of tossing, exact velocities of the air flow when the coin is in the air and all factors of that kind, we could predict the result of the toss. However, there are physical processes, result of which *cannot be predicted* at all, that is what quantuum mechanics teaches us. Therefore, "real" random numbers do exist and one

[1]For instance, digits of number π can be computed using random numbers, first it was done by French naturalist Georges-Louis Leclerc, Comte de Buffon in 1777.

day there could be a hardware random number generator that is unbreakable by any means of the attacker.

The measure for randomness is called entropy [4]. Here's the high-level idea. If you have a 32-bit word that is completely random, it has 32 bits of entropy. If the 32-bit word takes on only four different values, and each value has a 25% chance of occurring, the word has 2 bits of entropy. Entropy does not measure how many bits are in a value, but how uncertain you are about the value. You can think of entropy as the average number of bits you would need to specify the value if you could use an ideal compression algorithm. Note that the entropy of a value depends on how much you know. A random 32-bit word has 32 bits of entropy. Now suppose you happen to know that the value has exactly 18 bits that are 0 and 14 bits that are 1. There are about $2^{28.8}$ values that satisfy these requirements, and the entropy is also limited to 28.8 bits. In other words, the more you know about a value, the smaller its entropy is.

It is tempting to be optimistic about the amount of entropy that can be extracted from various sources. There exists software that will generate 1 or 2 bytes of supposedly random data from the timing of a single keystroke. Cryptographers in general are far more pessimistic about the amount of entropy in a single keystroke. A good typist can keep the time between consecutive keystrokes predictable to within a dozen milliseconds. And the keyboard scan frequency limits the resolution with which keystroke timings can be measured. The data being typed is not very random either, even if you ask the user just to hit some keys to generate random data. Furthermore, there is always a risk that the attacker has additional information about the "random" events. A microphone can pick up the sounds of the keyboard, which helps to determine the timing of keystrokes. You should be very careful in estimating how much entropy you think a particular piece of data contains.

Aside from the difficulty of collecting real random data, there are several other problems with its practical use.

- First of all, it is not always available. If you have to wait for keystroke timings, then you cannot get any random data unless the user is typing. That can be a real problem when your application is a web server on a machine with no keyboard connected to it. A related problem is that the amount of real random data is always limited. If you need a lot of random data, then you have to wait; something that is unacceptable for many applications.
- A second problem is that real random sources, such as a physical random number generator, can break. Maybe the generator will become predictable in some way. Because real random generators are fairly intricate things in the very noisy

environment of a computer, they are much more likely to break than the traditional parts of the computer. If you rely on the real random generator directly, then you're out of luck when it breaks. What's worse, you might not know when it breaks.

- A third problem is judging how much entropy you can extract from any specific physical event. Unless you have specially designed dedicated hardware for the random generator it is extremely difficult to know how much entropy you are getting.

1.2 Pseudorandom data.

Is there any way for an algorithm to produce random data? No, because algorithms have in its very basic idea the property of outputting the same result, whenever they are given the same input data. That's why algoritimically generated random data (or pseudorandom data) technically is not random at all, however there are still reasons for that title.

Pseudorandom data is generated by a determenistic algorithm with given initial value called *seed*. If you know the seed, you can predict the pseudorandom data. We want generated data be unpredictable, the exact form of this requirement depends on the usage of that generator. Usually there two types of pseudorandom number generators (PRNG): ordinary ones and cryptgraphically secure. Ordinary PRNG should pass statistical randomness tests, while the latter should have following properties:

- Given the first k bits of a random sequence generated by cryptographically secure PRNG, there is no polynomial-time algorithm that can predict the (k+1)th bit with probability of success better than 50%. Andrew Yao proved in 1982 that a generator passing the next-bit test will pass all other polynomial-time statistical tests for randomness. [5]
- Cryptographically secure random number generator should withstand "state compromise extensions". In the event that part or all of its state has been revealed (or guessed correctly), it should be impossible to reconstruct the stream of random numbers prior to the revelation. Additionally, if there is an entropy input while running, it should be infeasible to use knowledge of the input's state to predict future conditions of the CSPRNG state.

Traditional pseudorandom number generators, or PRNGS, are not secure against a clever adversary. They are designed to eliminate statistical artifacts, not to withstand an intelligent attacker. We have to assume that our adversary knows the algorithm

that is used to generate the random data. Given some of the pseudorandom outputs, is it possible for him to predict some future (or past) random bits? For many traditional PRNGS the answer might be yes. For a proper cryptographic PRNG the answer is no.

2 Pseudorandom number generators

In this section we will take overview of some generators, that are not cryprographically secure.

2.1 Linear congruential generators

Linear congruential generator was introduced by American mathematician Derrick Henry Lehmer in 1949. Its idea is rather simple and can be described as following:

$$X_{n+1} = (a \cdot X_n + c) \mod m.$$

Seed of this generator is the initial value X_0, other parameters are integers m, a, c. Of course, the maximal period length of this generator cannot be greater than m, however it can be smaller than m. For example we can consider $m = 9$, $a = 2$, $c = 0$ with initial seed $X_0 = 3$. Then:

$$X_1 = 2 \cdot 3 \mod 9 = 6$$

,

$$X_2 = 2 \cdot 6 \mod 9 = 12 \mod 9 = 3.$$

So only after 2 steps we came to the initial state. Try to think about needed requirements on a, c, and m that will make the cycle as large as it could be.

The exact requirements are:

1. Numbers c and m are coprime.
2. The number $b = a - 1$ is a multiple of p for any prime divisor p of the integer m.
3. The number b is a multiple of 4, if m is a multiple of 4.

Further information: [3].

Exercise 1

Choose some values for a, c, m and X_0. Write Wolfram code implementing linear congruential generator.

Answer of exercise 1

```
a=51;c=36;m=113;
X[0]:=5
X[n_]:=Mod[a*X[n-1]+c,m]
```

Exercise 2

Find the period of your generator.

Answer of exercise 2

We want to find the distance between two successive occurances of the same number. For this generator it's sufficient to find the step at which the seed (initial value) is occured.

```
i=1;
While[X[i]!=X[0], i++]
```

Here is a table for some popular values producing big periods.

Where it's used	m	a	c
Borland Delphi, Virtual Pascal	2^{32}	134775813	1
Turbo Pascal	2^{32}	33797	1
C++11's minstd˙rand	$2^{31}-1$	48271	0
Java's java.util.Random	2^{48}	25214903917	11

2.2 Lagged Fibonacci generator.

Recall the famous Fibonacci sequence, that is defined as following:

$$x_n = x_{n-1} + x_{n-2}.$$

With $x_1 = x_2 = 1$. We can consider that sequince modulo some integer m, it will produce sequence that can be called random. However studies has shown, that resulting sequence produces predictable results and has to be modified.

Lagged Fibonacci generator is defined as following:

$$X_{n+1} = X_{n-k} + X_{n-j} \mod m.$$

Usually m is equal to some power of two, for example 2^{32}. To produce a good generator integers j and k should satisfy the following property: polynomial $x^k + x^j + 1$ must be primitive modulo 2. Some of the pairs satisfying this property are:

$$j = 24, k = 55; j = 38, k = 89; j = 37, k = 100; j = 30, k = 127; j = 83, k = 258.$$

It has to be seeded with first $max\{j, k\}$ values, not all of them should be even.

Exercise 3

Implement such a generator with $j = 24$ and $k = 55$ (choose yourself X_q for $q < max\{j, k\}$).

Answer of exercise 3

```
j=24;k=55;
X[n_]:=If[n>=Max[k,j],Mod[X[n-k]+X[n-j],2^32],n+1]
```

3 Cryptographically secure PRNGs

The design of CSPRNGs is usually a lot more complicated, than that of ordinary PRNG. However there are some, that can be described quite easily.

3.1 Blum Blum Shub

Blum Blum Shub is a CSPRNG designed by Lenore Blum, Manuel Blum and Michael Shub in 1986. It's based on Michael Rabin's oblivious transfer mapping [6].

$$X_{n+1} = X_n^2 \mod m.$$

Where $M = pq$ is a product of two large primes p and q. The seed X_0 should be an integer that is coprime to M and not equal to 0 or 1. The two primes, p and q, should both be congruent to 3 mod 4.

Exercise 4

Implement Blum Blum Shub using Wolfram Mathematica.

Answer of exercise 4

```
X[n_]:=Mod[(X[n-1])^2,2]
```

Useful property of Blum Blum Shub is that any value X_i can be computed without computing intermediate values:

$$X_n = X_0^{2^n \mod (p-1)(q-1)} \mod m.$$

Computational difficulty of cracking this algorithm is equivalent to quadratic residuosity problem [7].

Examples of other CSPRNG:

- Yarrow algorithm designed by Bruce Schneier. `https://www.schneier.com/academic/yarrow/`
- Fortuna algorithm. [1]
- Algorithms using block ciphers and\or cryptographic hash functions for computing the value.

References

[1] Niels Ferguson, Bruce Schneier, Tadayoshi Kohno. *Cryptography Engineering: Design Principles and Practical Applications* Wiley Publishing, Inc., 2010

[2] Peter Gutmann. *Software Generation of Practically Strong Random Numbers* Proceedings of the 7th USENIX Security Symposium San Antonio, Texas, January 26-29, 1998

[3] Donald Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms.* Addison-Wesley Professional, 2014

[4] Claude Shannon. *A Mathematical Theory of Communication.* The Bell Systems Technical Journal, 27:370-423 and 623-656, July and October 1948.

[5] Andrew Chi-Chih Yao. *Theory and applications of trapdoor functions.* In Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science, 1982.

[6] Michael O. Rabin. *How to exchange secrets by oblivious transfer.* Technical Report TR-81, Aiken Computation Laboratory, Harvard University, 1981.

[7] Lenore Blum, Manuel Blum, Mike Shub. *A Simple Unpredictable Pseudo-Random Number Generator.* SIAM Journal on Computing. 15 (2)

[8] Don Davis, Ross Ihaka, and Philip Fenstermacher. *Cryptographic Randomness from Air Turbulence in Disk Drives* Proceedings of Crypto '94, Springer-Verlag Lecture Notes in Computer Science, No.839, 1994.

[9] Tim Matthews. *Suggestions for random number generation in software.* RSA Data Security Engineering Report, 15 December 1995.