Pseudo random bit generator

Practical approach

12/19/2007

Rasmus Bach Nielsen [rusmus@daimi.au.dk]
Anders Nørklit Thingholm [Anders@Thingholm.com]

Pseudo Random Bit Generators – Practical approach.

001100-010010-011110-100001-101101-110011[1]

Contents

Intro: 3

Measure of randomness: 3

PRBG-Strategies: 4

- Feedback Shift registers. 4
  - Linear feedback shift register (LFSR): 4
  - Generalized feedback shift register (GFSR): 5
  - Twisted Generalized feedback shift register (TGFSR): 5
  - Modular exponentiation: 5
  - Hashing 5
- Linux random number generator (LRNG) 6

Implemented PRBG: 7

- BlumBlumShub (BBS): 8
- RSA generator: 9
- Mersenne Twister: 10

Conclusion: 11

References: 12

[1] Also known as the code for paradox-correcting time travel.

Intro:

We have chosen to write this paper on pseudo-random bit generator, because it's a vital instrument in cryptology. Random bits are used everywhere, in encryption schemes and signature schemes use random bits in their execution, to ensure security. The problem with random bits is, that they are quite expensive to generate, since they have to rely on something being completely random, like flipping a coin. But since flipping a coin isn't convenient, alternative methods are used for generating random bits. This could be the time between emissions of particles during radioactive decay, air turbulence within a disk drive, causing random fluctuations in the disk read latency times, and so on. But since this isn't available for the masses to rely on, so Pseudo Random Bit Generators (PRBG) is used instead, and why we have chosen to focus our attention on this area, in this paper. PRBG is a deterministic algorithm, which takes an input of random bits, and expands those to a larger number of pseudo random bits.

$$Let\ k, l\ be\ positive\ integers\ such\ that\ l\ \geq k+1.$$

$$A\ (k,l) - bit\ generator\ is\ a\ functin\ f: (\mathbb{Z}_2)^k \rightarrow (\mathbb{Z}_2)^l\ \ that\ can\ be\ computet\ in\ polynomial\ time^2$$

There are many different ways to do this, and we will look at a couple of these in this note. We have chosen to look into some of the more popular PRBG, Blum Blum Shub, Mersenne twister, first because they are very popular, and secondly because they use different strategies to generate these random bits. This will give us something to compare with in our tests. We will also show how Linux combine different strategies, to make their LRNG[3], which are used for ex implantation of TSL/SSL, choosing TCP sequence numbers, choosing primes in RSA, and so on. But before looking at PRBG's, we will look at ways to ensure that the random bit generators are in fact "random enough". For this purpose statistical tests are used. Here we will also cover cryptographically security.

Measure of randomness:

Working with pseudo random bits, it is necessary to ensure a certain randomness to ensure that that it isn't significantly easier for an adversary to guess bits in the sequence, compared to genuine random numbers. We use cryptographically security as a way to explain what we mean with a sequence being random enough. A PRBG that passes the next bit test is said to be cryptographically secure.

The "next bit" test is defined as follows:

*Given a generated random sequence **s**, and a subsequence of the first **l** bits of **s** .If the probability that an adversary can guess the (**l**+1)-bit, is negligible in the length of the sequence **s**, then the generator is said to pass the next bit test.*

This definition says that if you have the first l bits in a cryptographically secure system, you won't be able to guess the next bit with more than negligible probability. The problem is how to check that a given PRBG is in fact a real PRBG with high enough level of randomness. It must not be significantly easier to be broken by

[2] The definition is taken from def 8.1 in [1]

[3] PNRG: Pseudo random number generator: These generate numbers instead of bits. But it can be converted quite easy.

an adversary. To do that, we can describe some test to run on the PRBG to detect any weakness it might have, and rating the randomness of the random bit generator. In a later chapter we will show how these statistical tests work. But even though these statistical tests are a good indication of the security of the bit sequence, the implementation is still very important. Not just how it generates from the seed, but where it gets it seed from. In an earlier version of the Netscape browser, the seed used for generation of the security in SSL connections was generated from the system clock. An adversary would, from monitoring the user, have a quite good estimate what time the seed was taken from the system clock, and therefore only have to test on a rather limited set of seeds. In the following section we will describe some of the strategies, used to make PRBG's, then later describe our implementations.

PRBG-Strategies:

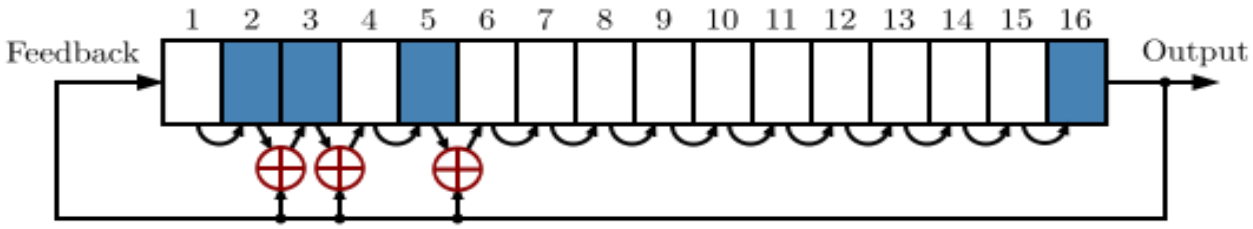
Feedback Shift registers.

First we will look at FSR, which are registers that uses previous states to generate input for them self. We have chosen to show 4 of these. The first because of its simplicity, the se

Linear feedback shift register (LFSR):

LFSR is a feedback shift register, that uses linear function, on some of the bits in the previous state, to generate a new input function, then right-shift the sequence, add the function output in the front, and the element being pushed out in the back, it the output. Since the only linear function on single bits is XOR (or inverse-XOR) the output it generates is deterministic. Because of that, it's important how the feed-back function is chosen, so the output might appear random.

Ex:



Generalized feedback shift register (GFSR):

GFSR uses instead of just a couple of bits in the previous state, use a entirely previous state and XOR this with the current state. This way the period of the output is greater than for LFSR. Practical period is $2^n - 1$, though the theoretical is $2^{nw} - 1$. Though this strategy for generating random numbers comes greater period, compared to most implementation of LFSR, it also has several drawbacks, compared to what we wish of a function. The cost of space required to run the algorithm is far greater, and time consumed to generate the random numbers is also larger. And the strategy it is still depending on a well-chosen seed, to ensure randomness.

$$x_{l+n} \coloneqq x_{l+m} \oplus x_x, \quad (l = 0,1\ldots)$$

where each x_l is a sequence of bits of size w.

Twisted Generalized feedback shift register (TGFSR):

TGFSR is an augmentation of GFSR to get rid of some of drawbacks. The way this is done, is to change the linear recurrence to

$$x_{l+n} \coloneqq x_{l+m} \oplus x_x A, \quad (l = 0,1\ldots)$$

where A is a $w \times w$ matrix with bit components, and x_l is regarded as a row vector over GF(2)[4]. Now we can with suitable choice of n, m and A attain a maximal period of the theoretical max $2^{nw} - 1$. This way we can also disregard the initial face as being critical, because only a seed with only 0's would be a problem, which additionally causes the initial face to be faster.

Modular exponentiation:

A technique used for both cryptosystems and PRBGs. Using this technique seem fairly obvious considering the properties we use during encryption. We try to make something indistinguishable from a random string by raising it to some power and computing the modulus. One of the advantages of this technique is that a lot of research has been conducted in the area, thanks to the applications in cryptography. This means that quite a lot of (probably) intractable problems are known, providing a wide range of possibilities that one can base a provably secure PRBG on.

Hashing:

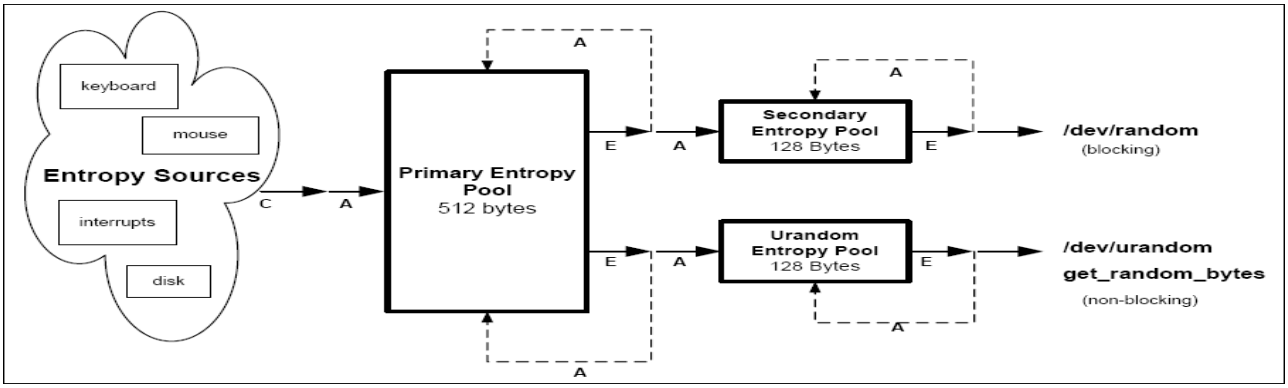
Some of the entropy sources that are readily available, are, while random, also correlated or biased in some way that enables an attacker to gain an advantage when trying to compute next number/bit in a sequence. One popular way of solving this problem, is by using hash functions. By applying a (secure) hash function to the bit stream/number before outputting, the properties of the hash function prevent any existing correlation/bias from being evident in the output.

Before going to our implementations, we will show a PRBG that combines several of the strategies, to make random numbers.

- [4] GF(2)is the Galois field (or finite field) of two elements

Linux random number generator (LRNG)

The LRNG is one of the most popular open source pseudo-random number generators, imbedded in the Linux kernel, which makes it widely used, in servers, symbian phones, PDA, routers, and so on. It contains around 2500 lines of code.



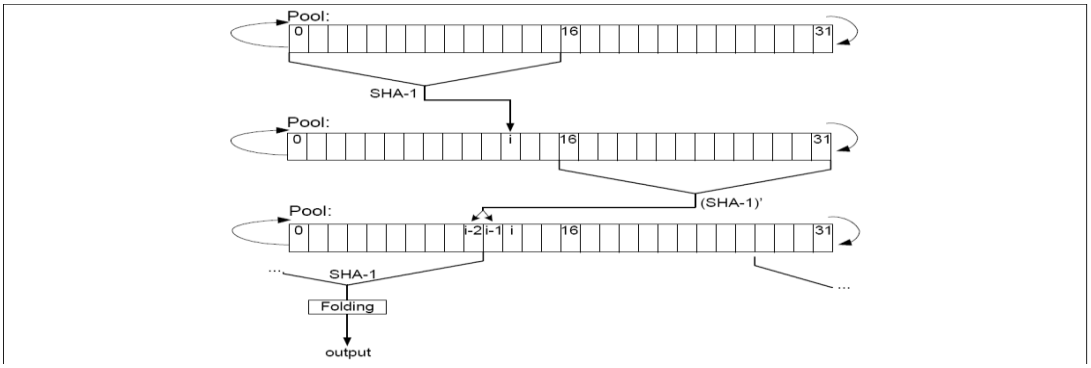
When looking at the LRNG it can be described as 3 asynchronous components.

1. C: Component for translating system events into bits
2. A : Component for adding these bits into the generator pool
3. E: Component, adding 3 consecutive SHA-1 operations to generate the output.

The first component is translating system events, like mouse clicks, drive access, interrupt, pressed key ect, onto 2 32-bit word. The first word is timestamp of the event, whereas the second is a binary interpretation of the given event.

The second component then takes these 2 events, and put them onto a generator pool. Here the TGFSR is used on the data received from component C. Whenever data is extracted using component E, the data is put back again, using component A.

The third is used to extract the entropy from the pools. When entropy is extracted, from either of the pools, the component alters the data. This involves hashing the content with SHA-1



Extraction algorithm.

Implemented PRBG:

In order to test the characteristics of the strategies mentioned above, we have implemented 3 PRBGs. Two of them are based on modular arithmetic, and the last is a TGSFR. We have chosen the two modular based algorithms because their security proofs, are based on familiar assumptions, and they are fairly easy to implement. The TGSFR was chosen because it has some nice properties, and because it is not cryptographically secure. A property we hope to expose through statistical tests. The test suite we have chosen to use is from NIST (National Institute of Standards and Technology), and comprises 16 tests:

1. The Frequency (Monobit) Test,

2. Frequency Test within a Block,

3. The Runs Test,

4. Test for the Longest-Run-of-Ones in a Block,

5. The Binary Matrix Rank Test,

6. The Discrete Fourier Transform (Spectral) Test,

7. The Non-overlapping Template Matching Test,

8. The Overlapping Template Matching Test,

9. Maurer's "Universal Statistical" Test,

10. The Lempel-Ziv Compression Test,

11. The Linear Complexity Test,

12. The Serial Test,

13. The Approximate Entropy Test,

14. The Cumulative Sums (Cusums) Test,

15. The Random Excursions Test, and

16. The Random Excursions Variant Test.

The test battery is designed to detect as many types of non-randomness as possible. This is done in a variety of ways, from the Monobit test that simply counts whether$\#1s \approx \#0s$, to template matching that searches for fixed target strings in the bit-stream. The tests have been run on files of various sizes in such a way that 100 streams could always be generated (the file with 100M bits was used to test streams of length 1000000). This was done because the tests operate with a 99% confidence interval, meaning that 100 tests (or more) must be conducted in order to detect deviations.

The generators are implemented in java, and all running-times have been tested on the same machine (a Pentium 4, 3,4 GHz, 1GB ram). In order to ensure comparability between the test statistics, all the algorithms harvest only the least significant bit.

BlumBlumShub[5] (BBS):

The BBS PRNG is a modular exponentiation based generator, that has a very strong security proof. The proof is based on the intractability of the Composite Quadratic Residues problem[6], which means that it is at least as hard to crack BBS as it is to do prime factorization. This proof along with its simplicity has made BBS a very popular PRBG for cryptographic purposes.

The sequence of numbers produced by BBS is defined as follows:

$$s_{i+1} = s_i^2 mod\ n$$

where

$n = pq, p \equiv q \equiv 3mod\ 4, q\ are\ \left(\frac{k}{2}\right) - bit\ primes$. The seed s_0 is chosen from the set of quadratic residues modulo n (often done by selecting $s_{-1} \in \mathbb{Z}_n^*$ and computing $s_0 = s_{-1}^2 mod\ n$) and we define:

$$f(s_0) = (z_1, z_2, \dots, z_i)$$

where

$$z_i = s_i mod\ 2$$

to be the bitstream produced by the BBS generator.

We tested the generator using the NIST tests, and found it to be secure for cryptographic purposes (or rather, it didn't fail any tests). It is, however important to note that this does not mean that the BBS generator is suitable for any application where random numbers are needed. Besides being too slow for stochastic simulations (see table below), the generator makes no guarantees as to the properties of the $s_i$that are produced. (We actually came across a report claiming that these numbers are totally unsuited for simulations[7]).

# bits generated	10^4	10^5	10^6	10^7	10^8
Time spent (ms)	89	471	4425	42300	47817

Time spent generating bits, by BBS.

For time sensitive applications, the generator can be optimized by harvesting the $r < \log_2 \log_2 n$ least significant bits, for a speedup of factor r.

5. The name is due to Lenore Blum, Manuel Blum and Michael Shub who first proposed the generator in 1986.
6. [1] pp. 338.
7. http://www.cs.dartmouth.edu/~akapadia/project2/node11.html

RSA generator:

Like BBS the RSA generator is based on modular exponentiation, however it is based directly on RSA encryption and owes its security characteristics to RSA. Because of its close relation to RSA encryption the security of the generator is straight forward, and anyone familiar with RSA encryption should be able to understand and construct an RSA PRNG.

We define the RSA generator as follows:

Generate $\left(\frac{k}{2}\right) - bit\ primes\ p, q$ and define $n = pq$.

Choose $b\ s.t. \gcd\left(b, \phi(n)\right) = 1$. A seed s_0 is any $k - bit$ element of $\mathbb{Z}_n^*$

For $i \geq 1$ define:

$s_{i+1} = s_i^2 mod\ n.$

Then define:

$f(s_0) = (z_1, z_2, \dots, z_i).$

where

$z_i = s_i mod\ 2.$

to be the bitstream produced by the RSA generator.

The RSA generator performed just as well as BBS on the NIST tests and is subject to the same limitations regarding the s_i produced during bit-generation. It is significantly slower than BBS, in its basic form but it can be optimized by harvesting more bits from each s_i and by finding a small b the exponentiation won't affect the running-time too much. Note that the size of b does not impact the performance of the algorithm as a whole. Below is a table of running-times for the RSA generator.

# bits generated	10^4	10^5	10^6	10^7	10^8
Time spent (ms)	1240	10332	71729	729256	7162381

Time spent generating bits, by RSA.

The above running-times were achieved by a basic generator, meaning that b is generated by picking a random long and incrementing it until$\gcd\left(b, \phi(n)\right) = 1$.

Mersenne Twister:

Mersenne Twister (MST19937) is a twisted generalized feedback shift register ((T)GFSR)-based generator. Given a seed, it will initialize a state vector, of length n, using a LSFR, and then apply a GFSR and a twist transformation to each of the entries. When extracting numbers, they are tempered through a series of bit-shifts and logical operations. Both the twist and the tempering are applied in order to ensure that the GFSR reaches its maximal period.

The parameters for MST19937 are as follows:

w: word size (in number of bits)
n: degree of recurrence
m: middle word, or the number of parallel sequences, $1 \leq m \leq n$
r: separation point of one word, or the number of bits of the lower bitmask, $0 \leq r \leq w - 1$
a: coefficients of the rational normal form twist matrix
b, ***c***: TGFSR(R) tempering bitmasks
s, *t*: TGFSR(R) tempering bit shifts
u, *l*: additional Mersenne Twister tempering bit shifts
lower_bitmask: bitmask for the r least significant bits

upper_bitmask: bitmask for the w-r most significant bits

We now define:

$$x_{i+n}A = ((x_i \,\&\, \text{upper\_bitmask} \mid\; x_{i+1} \,\&\, \text{lower\_bitmask}) \;\oplus\; (x_{i+n}))A, \;\; i = 0,1 \ldots$$

and

$$xA = \begin{cases} x \gg 1 & if\ x_0 = 0 \\ (x \gg 1) \oplus a & if\ x_0 = 1 \end{cases}$$

After completing the above operations our state vector is filled with pseudo-random numbers, however before they can be used, they must be tempered, in order to ensure that the GSFR can reach its maximum period. For MST19937 the tempering of x is defined as:

$y \coloneqq x \oplus (x \gg u)$

$y \coloneqq y \oplus ((y \ll s)\&b)$

$y \coloneqq x \oplus ((y \ll t)\&c)$

$z \coloneqq y \oplus (y \gg l)$

$return\ z$

In the case of MST19937 the actual values of the parameters are:

$(w, n, m, r) = (32,624,397,31)$

$u = 11$

$(s, b) = (7, 9D2C5680_{16})$

$(t, c) = (15, EFC60000_{16})$

$l = 18$

We tested the MST19937 in the same way we tested the BBS and RSA generators, and like the other generators it passed all the tests. However this does not mean that MST19937 is suitable for cryptographic purposes. Once 624 numbers have been seen, the state vector can be deduced, and all future values can be predicted. Even though we only use the least significant bit from each number, it is still possible to predict the bit stream. (It is sufficient to know the parity of each number, in order to predict the next bits). One of the reasons for MST19937's success is that it has an extremely long period (2^{19937}-1) and that it is fast:

# bits generated	10^4	10^5	10^6	10^7	10^8
Time spent (ms)	40	100	592	5490	54091

Time spent generating bits, by MST19937.

But it is not sufficient that a generator passes a test-suit. It have been proven and shown, that MST isn't a secure generator. The only way to ensure that a generator is secure is to use cryptanalysis. Those interested can find the test-runs at the address http://www.daimi.au.dk/~kingguru/ , together with the documentation of the NIST-Test suite.

Conclusion:

We have in this paper shown how 3 different PRBG can be generated (the source code can be find at http://www.daimi.au.dk/~kingguru/), explained the theory behind, what it will say to be a secure generator, how they take a seed or random number, and expand them to a larger number, using different strategies.
We have shown how Linux implements it PRNG, which is used in various places, combining strategies for generating random numbers, to implement its PRNG.
Last, the result from the NIST Statistical test suite for random and pseudorandom number generator for cryptographic application, have shown that they all looks to be random. All of the generators passed this Test. Even Mersenne twister passes the test-suite, though it's not considered to be cryptographically secure.

References:

1. Cryptography – Theory and practice. 3'rd edition
   Douglas R. Stinson

2. Handbook of applied cryptography –
   Alfred J. Menezes, Paus C. van Oorschot & Scott A. Vanstone

3. Wikipedia on Random Number Generation

4. Wikipedia on Pseudorandom number generation

5. Crypto analysis of the random numbers generator of the Windows Operating system
   Zvi Gutterman, Benny Pinkas & Tzachy Reinman

6. Crypto analysis of the Linux random numbers generator
   Zvi Gutterman, Benny Pinkas & Tzachy Reinman

7. A Statistical test suite for random and pseudorandom number generator for cryptographic application, NIST special publication 800-22
   Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, James Dray, San Vo.

8. Twisted GFSR Generators
   Makoto Matsumoto & Yoshiharu Kurita