Pseudo-Random Number Generators in Computer Security

Vickram Rajendran

Swarthmore College

Context

Pseudo-random number generators (PRNG's) are used ubiquitously in computer science as a means to mimic randomness allowing for greater security in cryptography and any computer security that requires randomness. Since there is no way to algorithmically create truly random numbers, the goal of PRNG's is to create a sequence of numbers that cannot be distinguished from random numbers.

Testing Randomness

George Marsaglia, one of the greatest minds in Random number generation, created a series of tests in order to measure how random certain PRNG's were suppposed to be. These "Diehard Tests", published in 1995, were the hallmark of any good PRNG, and the tests also invalidated many of the existing PRNGs at the time. A few of them are denoted below.

The Diehard Tests

- **Birthday Spacings:** Use the PRNG to find random points on a large interval- the spacing between these points should be exponentially distributed, as the birthday paradox tells us.
- **Overlapping Permutations:** Use the PRNG to find several sequences with the same five consecutive numbers. The $5! = 120$ possible orderings should occur with equal probability (up to sampling).
- **Minimum Distance:** Place 8000 points in a 10,000 x 10,000 grid, find the minimum distance between pairs. The square of this should be exponentially distributed.
- **Monkey Tests:** Create a sequence of random numbers. See if there are any repeats in the sequence, or any overlapping subsequences in the sequence.

Modern Secure PRNGs

There are three main ways to design a cryptographically secure PRNG.

- The first way is to base the PRNG on a cryptographic primitive, such as a cipher. Examples are Block Ciphers in Counter Mode.

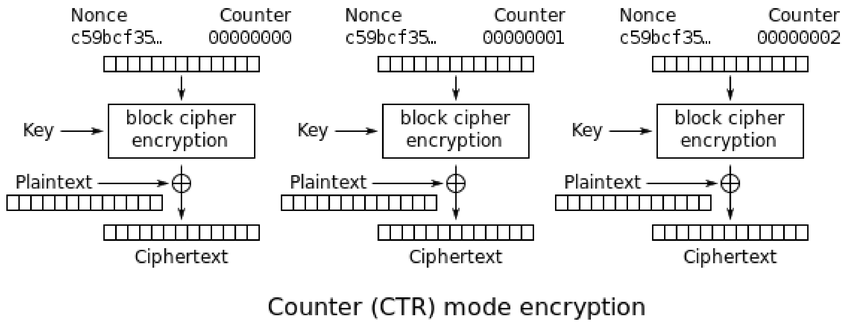

Figure 1: Example of Block Ciphers in Counter Mode

- The second way is to reduce the PRNG to a problem that is known to be hard. That is, an adversary that has non-negligible advantage towards the PRNG could be used to construct a solution to a hard problem. Then if we believe that the problem is truly hard, it would be computationally infeasible for any adversary to gain advantage over the PRNG. The classic problem to reduce to is integer factorization, as this problem is believed to be hard, and many of the cryptographically secure PRNG's will reduce to this. An example is the Blum Blum Shub Algorithm[1]:

$$\text{Given a seed } X_0, \text{let } X_n = X_{n-1}^2 \mod M \text{ where } M \text{ is a very large semi-prime.}$$

- The final way is to have the PRNG to be cryptographically secure by construction, in that it is specifically designed with cryptography in mind. Examples of this include Yarrow and Fortuna - these algorithms are the ones used in python cryptography and linux's dev/random[2].

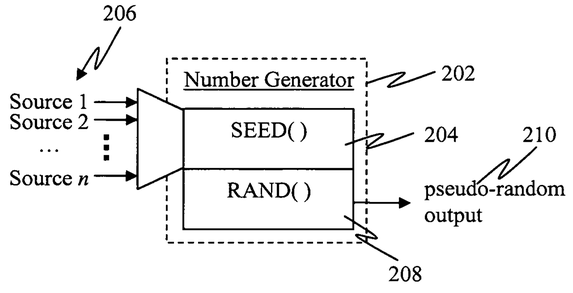

Figure 2: Yarrow and Fortuna's Basic Implementation

Popular PRNGs

Many prevalent PRNG's today sacrifice some security for computational efficiency, such as the following:

xorshift

- Incredibly Fast
- Low Space Complexity
- Failed some Diehard tests.

Mersenne Twister

- Considered the best PRNG of the time [3] (1997)
- Passed all Diehard tests
- Runs slowly
- High space complexity

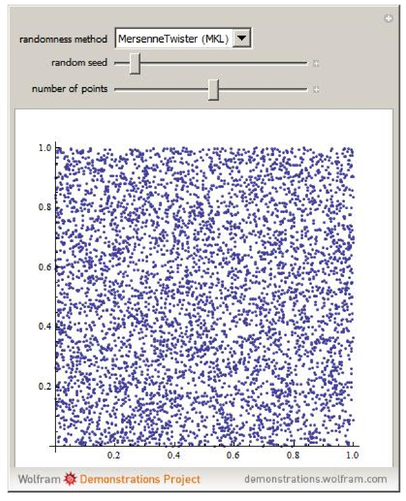

Figure 3: The Mersenne Twister [3]

References

[1] L. Blum, M. Blum, and M. Shub. A simple unpredictable pseudo-random number gener-ator. *SIAM*, 15(2):364, 2006.

[2] B. Schneier. Fortuna. *https://www.schneier.com/academic/fortuna*

[3] A. Jagannatam. Mersenne twister - a pseudo random number generator and its variants,2000.