Perceived Randomness: Pseudo-Random Number Generators & Human Interpretation

Kyle Carson
kylecarson@cs.ucsb.edu
Department of Computer Science
University of California Santa Barbara

June 17, 2017

Abstract

Random number generators represent a core building block within cryptography, in some cases forming the basis of software and hardware encryption. While true randomness requires specially designed hardware, many software implementations, known "pseudo-random" or "deterministic" number generators, exist. From efficiency of computation to the level of precision, these software-implemented algorithms differ along many vectors. In applied cryptography and beyond, there exist instances where humans directly interacting with the results of these generators may interpret them with their own concept of what is considered random. In this paper, we compare various DRNGs in terms of multiple quality metrics. As an additional exercise, we analyze the shortcomings of these algorithms against human interpretation, and design a method which introduces various types of biasing to these generators with the intent of improving the "perceived randomness".

1 Introduction

While many deterministic number generators, or "DRNGs", share a basis in modular arithmetic of primes, the finer details of their implementation tend to play a crucial role in their overall performance. Contrary to "true" random number generators which are designed at the hardware level around some source of classical or quantum physical entropy, the randomness of DRNGs is backed by certain assumptions about the complexity or irreversible nature of the building blocks used to form the algorithms.

For a DRNG algorithm to be considered for all intents and purposes cryptographically secure, there exists a general set of properties and behaviors that need to be met. The algorithm should take some k-bit input as a seed, preferably from a source of true randomness, and should output some n-bit random value. Some part of this

output should then contribute to the seeding of the next call to the algorithm, in a way that produces a statistically uniform distribution of random values over multiple calls. Of important note here is that, for a given seed, successive calls to a DRNG algorithm will produce some statics stream of values. That is, if one were to make successive calls again to the algorithm with the same initial seed as before, the stream of values produced will be exactly the same as before.

Some tests used to quantify the algorithms include the polynomial-time statistical test, which holds if no polynomial-time approach can distinguish the statistical difference of randomness between a DRNG and a truly random generator with probability greater than 0.5. Another test, the next-bit test, holds true if for the first m-bits of some output O, the m+1 bit cannot be predicted with statistical probability greater than 0.5. Following the basic model for structuring the algorithm and passing these tests quantifies a DRNG algorithm as cryptographically secure.

2 DRNGs

To attain a more applied understanding of these algorithms as described above, we've implemented seven different approaches including LCG, BBS, RSA, MSRSA, RAB, PG, and NR. Below we discuss the general principles for each algorithm, and anything of interest regarding each algorithms' actual implementation. All of the code was written in C, and can be found at `https://github.com/carsonkk/drngs`.

2.1 LCG

Linear Congruential Generator is a simple deterministic algorithm based on modular arithmetic of the previous output to produce the next output:

$$x_{\text{i+1}} = a * x_i + b(\text{mod n})$$

Depending on the values used to seed the algorithm, a sequence of some static window size is formed, such that once all of the values in the window have been produced, the sequence will wrap back around and repeat itself. There's many different options for what values to set the multiplicative and adder factors too depending on the application, however going by ANSI standard we set $a = 1103515245, b = 12345, n = 0x80000000$.

2.2 BBS

Blum-Blum-Shub is based on the following equation, where the modulus value is the multiplication of two large primes:

$$x_{\text{i+1}} = x_i^2(\text{mod n})$$

$$n = p * q$$

Some important properties must hold for the algorithm to behave as desired, including the seed value and n being co-prime and the primes p, q should both be congruent to 3mod 4. The actual output of the single pass is used to seed the next pass, however the only information used from an output towards the generated value is a singular bit of the output. Common bit outputs include even parity, odd parity, and the least significant bit. Therefore to actually generate a value, one must define some desired level of precision and run the algorithm that many times to fully form a generated value.

This pattern of using a single bit and doing multiple passes for some level of precision is quite common, and ends up being used in all of the following algorithms

2.3 RAB

The Rabin algorithm is a modified form of the the BBS algorithm:

$$x'_{\mathrm{i+1}} = x_i^2 (\mathrm{mod\ n})$$

$$x_{\mathrm{i+1}} = (x'_{\mathrm{i+1}} < n/2)\ ?\ x'_{\mathrm{i+1}} : n - x'_{\mathrm{i+1}}$$

$$n = p * q$$

2.4 RSA

RSA like many of these algorithms is based on the factoring problem, using a much more involved series of equations to generate its asymmetric results:

$$x_{\mathrm{i+1}} = x_i^e (\mathrm{mod\ n})$$

$$e \in [2, \phi - 1] \text{ with } gcd(e, \phi) = 1$$

$$\phi = (p - 1) * (q - 1)$$

$$n = p * q$$

2.5 MSRSA

A modified version of RSA, in this case referred to as Micali-Schnorr RSA, improves the overall efficiency of traditional RSA. Simply speaking, this is done by extracting more bits from each exponentiation, making modifications to both the range for the exponent e and how the bits for both successive seeding and output are chosen.

2.6 PG

The Power Generator algorithm derives its strength from using primitive elements of a set for a given primes to perform modular exponentiation. Due to how expensive it is to compute primitive elements for significantly large primes, we pre-computed a set of primes and associated primitives and used them to drive the algorithm. While this greatly increased the efficiency of the computation, a less than desirable result of this

was a significant limit on the range of values that could be generated, particularly when using smaller primes. A better approach would be to invest the computational power to generate primitives for large primes and index them for future use, however this would take a non-trivial amount of time to complete.

$$x_{\mathrm{i+1}} = g^{x_i} (\mathrm{mod\ p})$$

2.7 NR

The Naor-Reingold algorithm is similar to the Blum-Blum-Shub algorithm in that tis complexity is dependent upon the inability of an attacker to factor large prime integers. It uses a few building blocks of its own for it's computation, including a binary vectoring function, a modular product function, and a integer vector comparison function. The binary vectoring function ($bin_k(u)$) takes two integer parameters, k and u, and translates the k-least-significant-bits of u into a binary vector array. If u has less bits than the value of k, the vector array's most-significant-bits are padded with 0's. The modular product function (dot) takes two binary vectors of the same length and takes the summation of the products of each bit from the two vectors, applying a modulus of 2 to the result. Finally the vector comparison function ($f(A, b)$) takes two vector parameters, A which is of integers and length $2k$, and b which is a binary vector of length k. The values in A are in pairs, where the first value is relevant if the corresponding bit in b is 0, or the second value is relevant if the bit is 1. Thus the function selects each of the relevant integers from A using b, then returns their summation. These functions are then used in the actual computation of the generated value:

$$b = \mathrm{bin}_k(i)$$

$$u = f(A, b)$$

$$v = g^u (\mathrm{mod\ n})$$

$$x_{\mathrm{i+1}} = x_i \text{ dot } \mathrm{bin}_{2k}(v)$$

3 Testing

3.1 Setup

The test setup for measuring the quality of each algorithm was based on two interrelated metrics: CPU cycles and real-world time of execution. To somewhat normalize the precision between each algorithm, each one's random function call uses the same methodology as LCG in which the upper-order magnitude of bits is used when returning the actual output. The experiment was performed multiple times, varying the number of successive calls made to each algorithm as outlined below in Table 1 and Table 2.

Algo \ Calls	10	25	100	250	10000	100000	1000000
LCG	0.0	0.0	0.0	0.0	0.0	0.001	0.012
BBS	0.0	0.0	0.0	0.0	0.006	0.064	0.468
RAB	0.0	0.0	0.0	0.0	0.008	0.077	0.524
RSA	2.53	6.242	26.536	70.27	-	-	-
MSRSA	0.0	0.0	0.0	0.0	0.051	0.275	2.432
PG	0.0	0.0	0.0	0.0	0.033	0.276	2.206
NR	4.89	28.934	444.311	504.85	-	-	-

Table 1: Algorithm speed in terms of real-world computation time (seconds)

Algo \ Calls	10	25	100	250	10000	100000	1000000
LCG	1	1	2	3	100	1126	21486
BBS	4	11	41	105	6750	60364	453863
RAB	6	14	53	132	8616	68644	538605
RSA	2044421	7107596	25926498	66659430	-	-	-
MSRSA	41	32	243	607	37529	287388	2544693
PG	29	55	214	533	37895	283791	2154358
NR	4177409	27015995	416989667	2652261381	-	-	-

Table 2: Algorithm speed in terms of CPU cycles

3.2 Results

From the data above we can see a massive disparity in the performance of RSA and NR compared to the rest of the algorithms. RSA's inefficiency in the case is do to the large size of e as a result of the values used to seed the algorithm, requiring each pass to loop a non-trivial number of times. MSRSA avoids this by using a commonly agreed upon static value of e, greatly reducing the runtime. The results of PG are also a bit skewed, as it uses a relatively small index of prime/primitive pairs, where the primes themselves are quite small. This becomes more evident when using the *print* test in the repository and observing the values output by each algorithm side-by-side, as the resolution of PG's output is noticeably more limited.

4 BRAND

While each of the aforementioned algorithms strives to produce a uniformly distributed set of seemingly random variables, they tend to have some shortcomings when used in applications. The range of values they produce are often limited to some $[0, RAND\_MAX)$, which may be problematic in that both the desired range size may too big or too small, or the min and max values themselves aren't ideal. If a user wants to use one of these algorithms in an applied manner for a range they control, a naive solution is often something along the lines of:

$$val = min + rand()\%(max - min)$$

The problem with this is it introduces a statistically detectable bias when the modulus doesn't perfectly divide into the $RAND\_MAX$ of the $rand()$ function.

As a solution to this, we provide *brand*, a set of wrapper functions around C's implementation of $rand()$, which is fundamentally based upon LCG. With this library, the default min and max for the range of random values is still 0 and $RAND\_MAX$, however now the user can set the limits of the range, then use the wrapper functions to generate a statistically uniform distribution of random variables within the given range.

The standard flow for using *brand* is to first seed C's built in $rand()$ function however you like; typically this is done through a call to $srand(time(NULL))$. Next, the user can use *brand*'s limit setting functions $bsetn()$, $bsetx()$, and $bsetnx()$ to define the desired range. Next the user simply calls $brand()$ as many times as they want to get the desired value, and finally when they're done using the library call $bcln()$ to clean up dynamically allocated memory. An additional function $brandd()$ is also available, which uses $brand()$ internally to generate a random value in the range $[0.0, 1.0)$, with the resolution of the result controlled by the limits set.

With this we now have a well functioning RNG library based on LCG in which we can control the range, but now we want to add one more piece of functionality. When it comes to human perception of randomness, people tend to apply their own biases to what they perceive as random, sometimes without even realizing it. A prime example of this is the "shuffle" button in any sort of music or media software. The

expectation when listening to a library of music with the shuffle button enabled is that the software will cycle through the music and select the order to play songs in at random. If, for example, a user has a library of 1,000 songs being shuffled, and the RNG function happens to select the same song twice back to back before playing any other song, the user will likely be annoyed by hearing a repeat, breaking their biased perception of "randomness".

Thus *brand* has an additional wrapper function, *brandus*(), that aims to produce a uniquely random stream of values for a given range. For a given range of size N, it achieves this by allocating an array of length N and setting each value of the array to be the corresponding value in the range. It then generates a random value within the range, removing the minimum offset, and uses that as an index into the array to find the value to return. With that value saved, it then shifts all values beyond the output value down in the array, effectively removing it from the set of available values, and decreases the maximum range value by one. Some performance implications of this include the cache array must be $O(N)$, an $O(1)$ lookup time as the random value generated is just an index, and a $(N/2)$ average update time factor.

While this works great for smaller ranges and always ensures a unique stream of random values for N passes on a range of size N, it has two shortcomings. The first being that, when all N values have been generated, it's behavior is that it will reset the values in the array and begin a new unique stream. This is fine, except that say in our previous example song 1000 plays at the very end in the first full pass, and the very beginning in the second full pass, then we'll have two back-to-back instances of the same object and will have again broken our notion of "perceived randomness". The second shortcoming is the memory footprint of this method. For any non-trivial range of values, the method will take up a substantial amount of memory as it needs to index every single value (at least initially).

Some potential future work for this cached indexing approach could include an alternative cache method which only caches the most recent X amount of values out of the total N range, where $X << N$. Some potential issues with this include lookup failures and having to do multiple retries, however these could likely be mitigated by [placing a strict limit on the size of the local cache relative to the overall size of the range.

5 Conclusion

DRNGs can vary widely in both the approaches they take to achieve cryptographic security and statistical randomness, and the actual performance that comes as a result. Some are much more specialized, such as RSA for asymmetric key encryption, while others are much more broad in their applications, like LCG. Beyond the algorithms themselves, ensuring statistical uniformity in a random distribution where the range or uniqueness of a value matters is surprisingly non-trivial to implement, however in applications that interface directly with users or use cases such as procedural generation, these properties can be invaluable.