Programmatically Combatting Pseudorandom Number Generators With Uniform Integer Distributions: A Modern C++ Approach

Joshua A. Schiavone | jshschiavone@gmail.com
Computational Research, DoubleThreat Security & Engineering
Toronto, Ontario, Canada

May 2, 2021

Abstract

Pseudorandom Number Generators come as a great assistance to programmers. Although, they come with a great deal of security flaws as they do not truly generate a "random" sequence of numbers. The C++ Standard Template Library provides a solution to this problem as programmers can now implement more secure seed-able random number generators to provide a proper integer distribution of non-deterministic random values, to better support their programming practices.

Keywords: *pseudorandom, random, programming, programmers, numbers, pseudorandom number generators, security, security flaws.*

1 Introduction

Outline It has been demonstrated that PRNGs (Pseudorandom number generators) have been an adequate method of producing basic sequences of randomness when security is not of a great concern. When developing a program that acts as a Texas Holdem' poker dealer, you would not want the players to guess the cards with ease. Therefore, poker dealers must shuffle a deck of cards in such a way that takes little computable inputs into account to make the card-guessing process more strenuous on the players. This concept is similar when referring to random number generators, we don't want a variety of inputs that lead to a deterministic result. This would put the individuals moderating the game at a great monetary disadvantage. In statistical research, proper random number generation implementations are very crucial, especially in cryptographic algorithms. The goal for this paper is to properly outline common mistakes that C++ programmers make when implementing PRNGs, to theoretically define these flaws, and to provide a superior solution – programmatically. Also, it is crucial to note that the programmatic solutions I will be providing will not be capable in generating true randomness as that is never the case in computation. Computers are not random machines, they are logic based. Therefore, my methodology defines better software implementations that combat the use of a traditional PRNG, through modern C++ features based around non-deterministic seeds and uniform integer distributions. All of the mentioned provide non-substandard security features when dealing with salient implementations.

2 Related Work

The methodology provided within this paper is related to the 2011 C++ standard `random` template. The utilization of these template models tend to go *unseen* for a vast majority of software engineers. My outline of it's proper use through uniform integer distributions correlate's to Donald E. Knuth's and Andrew C. Yao's, "The complexity of nonuniform random number generation. Algorithms and Complexity: New Directions and Recent Results", 1976. In some sections of this paper, I designed method's to increase complexity through procedural summations as well as an organized container implementation through vector storage.

3 Common Pseudorandom Number Generators

Every PRNG shares the same design concept. As in most cases they all depend on a seed which is the starting value for the PRNG function, and computes values accordingly with respect to the seed value.

3.1 Linear Congruential Generator

The Linear Congruential Generator (also referred to as the LCG), is an algorithm that produces a sequence of pseudorandom values with a discontinuous piecewise linear equation [1]. It takes four mandatory inputs that are required to produce a pseudorandom value. These inputs are, a modulus value, multiplier, incremental constant, and a seed. The LCG is defined as:

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

where X is the sequence of pseudorandom values, and:

$$m,\ 0 < m$$
$$a,\ 0 < a < m$$
$$c,\ 0 \leq c < m$$
$$X_0,\ 0 \leq X_0 < m$$

3.2 Middle-Square Algorithm

The middle-square method was invented by John von Neumann in 1949 as it serves as another method to produce pseudorandom values. It comes with severe design flaws as it takes a seed for an input, squares that seed, then extract the middle portion of that result (with respect to the number of digits) and repeat that process. The output of each previous result, will become the next seed. It is obvious why the MSA is not implemented when security is of great concern. Note that sequence generates a deterministic result that falls under, $X_{n+1} \in [0,\ 9999]$ when computed traditionally.

4 Deterministic and Non-Deterministic Seeds

A function passed with a constant seed value will produce the same output value every time the algorithm is computed. In terms of the LCG, $X_{n+1} \in [0,\ m-1]$ will always be true when the seed does not change. Let's provide some inputs into this PRNG,

$$m = 7829,\ a = 378,\ c = 2310,\ X_n = 4321$$

Upon computing,

$$(4321 \cdot 378\ + 2310) \bmod 7829$$

We get 7216. This value will always remain the same due to the fact there isn't a true distribution of random values and it is always seed-dependant. Therefore, $0 \leq X_{n+1} \leq m - 1$.

4.1 Non-Deterministic Approach

A more logical approach is passing a seed through the Mersenne Twister Algorithm from a generated uniform distribution of integers. The seed will be randomly selected by the MTA and then can later be computed through a PRNG.

This makes the approach non-deterministic providing better security. A theoretical example: $MTA(T_{gen})$ will represent our Mersenne Twister Algorithm that takes some random number engine (crucial).

$$\beta(x) = X_n \in [\min(R),\ \max(R)]$$

, will represent our distribution range function where *R* is the bit length of whichever type is passed through the algorithm. Lastly, we'll evaluate,

$$\forall\, x \in \beta(x),\ \beta(MTA_{x_{n+1}})$$

In essence, we pass some random number engine through the MTA, then iterate through a vector with respect to our uniform integer distribution and then compute an MTA sequence for every *x* occurrence in our distribution.

4.1.1 Constructing Verbosity

If we wanted further complexity with a non-static vector (*V*) size, we can define a procedural summation that can be computed as such:

$$\sum_{n=1}^{b} \beta(MTA_{X_n}), where\; n \in [V_1, V_b]$$

In the case of `gen_unbias_data(gen)` where $|V| = 5$,

$$\sum_{n=1}^{5} \beta(MTA_{X_n}) = X_1 \in \beta(x)+X_2 \in \beta(x)+X_3 \in \beta(x)+X_4 \in \beta(x)+X_5 \in \beta(x),$$

would suffice a complex iteration to provide more verbosity. If we wanted to adhere to this, we can pass the summation as an argument to the MTA:

$$MTA[\sum_{n=1}^{b} \beta(MTA_{X_n})]$$

5 Effective Computation Through Uniform Distributions

The C++ standard template library provides us with distribution engines that makes use of a uniform discrete distribution. The `std::uniform_int_distribution` feature is a template that produces integer values according to a uniform discrete distribution, in which it is described by the following *probability mass function* [2]:

$$P(k|a,b) = \frac{1}{b-a+1}\,, a \leq k \leq b$$

This distribution produces some value $k \in [a, b]$ where each possible value has an equal likelihood of being produced [2]. The goal for our C++ implementation is to provide a sequence through a distribution where each integer in the distribution has a probability of $\frac{1}{n}$ of being produced. Thus, $\mathcal{U}\{a, b\}$ must support:

$$k \in \{a, a+1,\ ..., b-1,\ b\}$$

5.1 STL Implementation

Let's represent our $\beta(x)$ function previously introduced in section 3.1 as `gen_unbiased_data(gen)` from a modernized perspective:

```
template <typename rand_type>
std::uniform_int_distribution<rand_type>
  gen_unbiased_data(
   const std::mt19937 mtgenerator) {

   std::uniform_int_distribution<rand_type>
     distribution(
      std::numeric_limits<rand_type>::min(),
      std::numeric_limits<rand_type>::max());
   return distribution;

}
```

The above code sample generates a uniform integer distribution that takes in a `std::mt19937` which is C++'s representation of the Mersenne Twister Engine. Now we must create a function to store our distribution into a vector container. Although, before that iteration, to create a non-deterministic seed, we will pass an `std::random_device` object into the Mersenne Twister engine and then push back elements into the vector with respect to our uniform integer distribution:

```
template <typename rand_type>
std::vector<rand_type> gen_nd_vector(
```

```
        std::vector<rand_type>& dataset) {

        std::random_device rdevice;
        std::mt19937 mtgenerator(rdevice());
    auto dist = gen_unbiased_data<rand_type>(
        mtgenerator);

        for (auto elem = SET_BEGIN; elem < SET_SIZE;
            ++elem) {
                dataset.push_back(dist(mtgenerator));
        }
        return dataset;
}
```

Note that the range of the iteration is between previously defined macros, `#define SET_BEGIN 0` and `#define SET_SIZE 12` ($\forall\, elem \in [Vec_{begin},\, Vec_{size}]$). This is just a sample range for the vector. In continuation, let's compute the above code samples within our `main()` function:

```
int main(void) {
        std::vector<short> dataset;
        gen_nd_vector<short> (dataset);

        for (const auto& content : dataset) {
                std::cout << content <<          ;
        }
}
```

6 Results

Here is the output after each test case. Note: a new test case begins each time the program is executed.

Test Case	**Output of:** `gen_nd_vector<rand_type>(std::mt19937)`
Case 1	22782 20067 27385 17392 -5306 10430 2267 -17873 -6163 14796 -7473 25514
Case 2	-666 3325 -12598 -11171 -11805 -14701 21044 -27357 -9424 -12691 31450 -2479
Case 3	-20485 -2225 -27266 -27704 29701 -27773 -4999 -19936 587 31796 13820 -11365
Case 4	5008 -31059 -24342 5234 -28484 -26810 30440 -6193 6239 -6045 -31900 -9339
Case 5	19381 -25926 1628 1288 3769 20582 6060 -15384 -32708 31558 -28763 -30383

Table 1: $\beta(x)$ case observations

It is clear that the data presented is indeed sequences of randomness and when tested with a large quantity of test cases and elements, it increases the complexity of predicting the outcome of the next iteration. This is the sheer power that non-deterministic seeds hold and how providing a non-deterministic seed through a uniform integer distribution, that creates an un-bias, can improve security. This is especially crucial when cryptographic implementations are taken into account.

7 Cryptographic Security Concerns

The Mersenne Twister Algorithm is one of the most notable random number generators. Although, it does bring the security concern of seed-predicting which makes it not cryptographically secure – when implemented by itself. With the examples provided in 3.1 – 4.1, it essentially makes the MTA prone to a lot less security concerns. Without the implementation of `std::random_device`, the Mersenne Twister Algorithm would be a very poor choice to implement in any software that takes security into account.

7.1 Notable Flaws

A. The most obvious reason why Mersenne Twister on its own is not cryptographically secure is due to the fact that it's based off of a linear recursion. The MTA produces a a long sequence of outputs which one can easily implement as predefinitions to predicting the next output. The basics of a recurrence relation is described as such:

an equation that expresses each element of a sequence as a function of the proceeding ones. A recurrence relation has the form [3]:

$$u_n = \varphi(n, u_{n-1}) \;\; for\; n > 0$$

where,

$$\varphi : N \times X \rightarrow X,$$

is a function, where X is a set to which the elements of a sequence must belong [3]. To put this referenced definition into perspective, the factorial function is also defined by the recurrence relation:

$$n! = n(n-1)!\; for\; n > 0$$

B. It is also known that recursive algorithms do not bring the best performance, but that is normally input dependant. In terms of computation, recursion can also exceed the maximum size container of the call stack – which is not ideal. For a program that just has the goal of a producing a sequence of integers that appear random to the user, using the Mersenne Twister Algorithm is perfectly fine. It provides a fair bit of complexity in which it can make the average user think they are observing pure randomness, although through computation and recursion we know that it's not the case whatsoever. In theory, iteration algorithms take less time to compute over a given data set than recursive algorithm. Although, the computational speed of the Mersenne Twister Algorithm is relatively fast when fed with a small container size.

After initialization, MTA produces a state, than a recursive twist, produces another state with random values, recursive twist, and than it repeats itself. The previous always occurs after a given seed is initialized (traditionally — non-deterministic.

The visual steps that the MTA takes to produce *randomness* are as such:

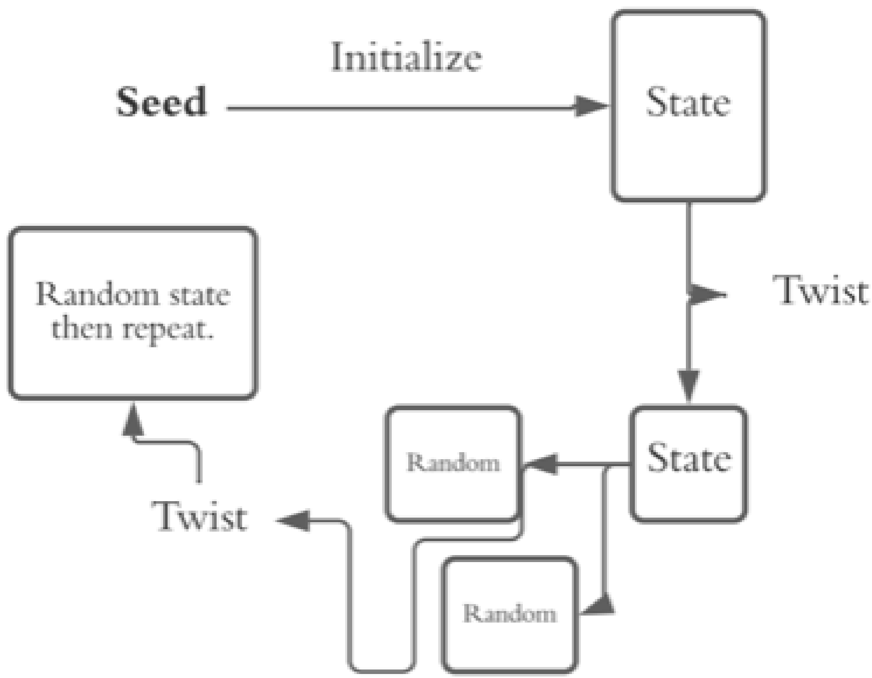

Figure 1: MTA procedural steps via flow chart visualization

8 True Randomness vs. Pseudo-Randomness

As pointed out in section 1, true randomness cannot be generated by a computer on its own. We can only retrieve pseudo-randomness which in a sense is an emulated sequence that just appears random. Generating true random numbers can only be done by an external hardware device that generates a random number from physical phenomena rather than a computable algorithm. For example, a device that produces random numbers based on radio frequencies is considered a true random number generator as RFs are not step-oriented, and cannot be reversed or guessed. Pseudorandom number generators were developed as an assistance for individuals who do not have access to such hardware devices, so they developed PRNGs to emulate randomness to users. The best true hardware random number generators stem from quantum random properties. This is because quantum mechanical physical randomness cannot be predicted as in a lot of cases they occur in the atomic or sub-atomic level. Popular quantum phenomena methods include: **a**. Semi-transparent mirror containing traveling photons. **b**. Shot noise: a quantum mechanical noise source in electronic circuits. **c**. Vacuum energy fluctuation detection. **d**. Nuclear decay radiation source.

9 Success Strategies For C++ Programmers

For programmers who would like to have random number implementations into there software, avoid 1–2 and utilize method 3:

1. `std::rand()`, `srand()` and `rand()`. As they all require a deterministic seed and a basic PRNG implementation. They also lack a proper integer distribution.

2. Avoid using a pseudorandom number generator by itself. The best implementation is when assisted with an integer distribution.

3. For producing random sequences, generate a new random integer (with respect to the distribution) and store that integer one by one, into a container. As this is very efficient when using the MTA as the time it takes to produce *N* is *O(1)*. Note: the complexity for the whole process is in linear time.

10 Conclusions

The methodology provided in this paper demonstrates that the templates in `<random>` provides programmers with a better alternative to `std::rand()`. It gives C++ programmers the capabilities of producing a more secure mechanism of generating pseudorandom sequences through uniform integer distributions. It has been highlighted that using a uniform distribution prevents a PRNG bias caused by deterministic seeds. Therefore, we theoretically defined a function $\beta(x)$, that fills a vector with a secure sequence of pseudo randomness from the non-deterministic random engine, T_{gen} . My goal from this research is to exaggerate overlooked STL features that should be implemented in software where security is of great concern when random numbers are in question.

Acknowledgements

I'd like to thank the people at DoubleThreat Security Engineering in Toronto, Canada that made this paper possible. Also, I'd like to thank my GitHub con-

tributors who provided me with the motivation to creating this paper to greater assist C++ programmers that are curious in seeking superior methods to combat PRNGs.

References

[1] Linear Congruential Generator - Wikipedia
`https://en.wikipedia.org/wiki/Linear_congruential_generator`

[2] `uniform_int_distribution` - C++ Reference
`https://www.cplusplus.com/reference/random/uniform_int_distribution`

[3] Recurrence Relation - Wikipedia
`https://en.wikipedia.org/wiki/Recurrence_relation`

[4] Luc Devroye. *Non-Uniform Random Variate Generation.* Springer-Verlag, New York, 1986

[5] Donald E. Knuth and Andrew C. Yao. *The complexity of nonuniform random number generation. Algorithms and Complexity: New Directions and Recent Results.* pages 357–428, 1976.

[6] John von Neumann. *Various techniques used in connection with random digits. Applied Math Series.* 1951