Reversing Florida State University's Scalable PseudoRandom Number Generator (SPRNG)

Caleb Sbani
Computer Science
Florida State University
Tallahasee, FL
ccs22g@fsu.edu

***Abstract*—In using a stream of pseudo-random numbers, they always generate in a set order given the seed. Using that seed, it becomes possible to generate the same period in reverse. by using two different streams to generate a single repeatable period with two different processors and a single seed, it becomes possible to generate random numbers in parallel without extra input. Additionally, "storing" numbers from a generator by operating the same generator in reverse provides easy storage without significant overhead. In this paper we took the generators used in SPRNG and evaluated how they could be reversed in practice whilst maintaining the current generation period.**

I. Introduction

A Pseudo-Random Number Generator (PRNG) is a sequential Random Number Generator (RNG) that outputs seemingly unassociated numbers in order after being fed a strict "seed". The development of and where some programs might obtain this seed is unimportant to the scope of this project because we assume the generator has already been created in its default (seeded) state. Knowing that the PRNG always has predeterminate outcome states depending on the input seed of the program allows for us to know it is possible to predict the next or previous values in the sequence given the current output and seed. However, all PRNGs are not created equal, different PRNGs need to be reversed in diverse ways. Establishing that there is some pre-determined outcome allows us to say yes; it is possible to reverse and find out what the previous numbers in the system were. However, certain differences can arise in practical applications of reversing a PRNG as not all mathematical operations have an equivalent inverse. In addition to this, going through the entire period is infeasible as [6] states "storing full paths from the constraint equation for re-use in the adjoint equation becomes infeasible due to memory limitations".

For the scope of this paper I plan on reversing the six different PRNGs available in SPRNG (FSU's Scalable Pseudo-Random Number Generator) [7]. This includes a Linear Congruential Generator (LCG), a 64-bit LCG referred to as LCG64, a Lagged Fibonacci Generator (LFG), a Parallel multiplicative Lagged Fibonacci Generator (MLFG), a Combined Multiple Recursive Generator (CMRG), and a Prime Modulus Linear Congruential Generator (PMLCG). I further planned to reverse them in such a way that would not modify how they already operate, IE: if the given seed returns a set value after x iterations, my modifications will allow for the same value to be returned with the same seed after the same x iterations, so long as the machine has not been reversed.

The applications for the reversal of a random number generator would mean that on an application that does not require extremely intensive use of a single information stream of random numbers an application could be run in double the speed from running a single stream in both forward and reverse. According to [1] this would at most double the speed it was previously generating at without significant additional overhead in memory. This is further stated in [8] "Bi-directional (forward and reverse) execution find use in fault-tolerant high-performance computing, rollback-based optimistic parallel simulations, large-scale debugging, and other areas". Emphasizing the usefulness of the reversal of PRNGS.

II. Reversing The Generators

As mentioned before RNG, or rather PRNGs are not entirely random. They are "Pseudo-random" in other words if they are not random, and they run through a machine, a computer, a deterministic system, then the same inputs will always lead to the exact same outputs. As a result. When a computer generates a random number doing some sort of mathematical operation on the previous number to get the next number. The next value will be theoretically a product of some function X, and some value or set of values gained from the previous equation. We will call this set S. We can simplify this by stating the following equation:

$$S_n = X(S_{n-1}) \tag{1}$$

Using this equation, we can determine the previous value by taking the inverse function of X, and applying it to S. This inverse function will be referred to as X' and can be represented in the following equation:

$$S_{n-1} = X'(S_n) \tag{2}$$

The only difficulty now is finding each different PRNG, for each different function that X can take, what is the inverse of X? What is X'?

A. LCG

An LCG or a Linear Congruential Generator, first proposed by [5] is a generator that has values A and C and exists within modulus M in addition to some starting seed or value

which we refer to as "seed" in the example code. This simple Generator takes the previously generated number or if it is the first number to generate, the seed, and multiplies it by the multiplier value A, adds the offset value C, and then reduces the number (if necessary) by putting it into the modulus of M. This is shown as an equation of

$$X_{n+1} = X_n \times a + c \mod m \tag{3}$$

This then continues with the new number acting as the seed for every consequent number generated. Three of the five generators are some modifications of this generator here.

Listing 1. Example LCG code in python

```
def lcg(m:int,a:int,c:int,seed:int)
-> Generator[int, int, int]:
    while True:
        seed = (a * seed + c) % m
        yield seed
```

In the practical application of reversing an LCG. The equation itself needs to change.

$$X_{n-1} = (X_n - c) \div a \mod m \tag{4}$$

In this reversed function we can see how now the generator can work in reverse if it wasn't for the $\mod m$. in the series of numbers inside modulus m. all numbers must be contained in $\mathbb{Z}$ such that each number $n < m$. To do this, instead of dividing by the multiplier a we must multiply by something that will create an equivalent value given the modulus. Known as the multiplicative inverse of a referred to as $a - 1$. We can calculate the multiplicative inverse of a given modulus m. [4] provides a useful algorithm for doing so, the Extended Euclidean Algorithm. This can be represented through the following python code:

Listing 2. Extended Euclidean Algorithm in Python

```
def moduloInv(a : int, m : int) -> int:
    r, q = m % a, m // a
    t1 = 0
    t2 = 1
    t3 = (t1 - q * t2) % m
    while(r >= 1):
        m = a
        a = r
        r = m % a
        q = m // a
        t1 = t2
        t2 = t3
        t3 = (t1 - q * t2) % m
    return t2
```

With a^{-1} now defined we can then rewrite and apply the inverse equation of the LCG:

$$X_{n-1} = (X_n - c) \times a^{-1} \mod m \tag{5}$$

This application works well for LCGs because a-1 remains constant. It does not have to recompute a new multiplier and go through the entire extended Euclidean algorithm through every iteration, only through every inversion. Making the reversal operation itself about O(log(N)) with n being the relative sizes of a and m. and any step of the generator keeping the continuous O(1) efficiency just being the same as before.

In the application of reversing the LCG for SPRNG it must be noted that the format of the equation itself has changed, the offset C must be subtracted from the value before anything else can occur. Because of this, There must be two separate paths for the system to take when it is reversed and when it is not. As a result, a small amount of overhead must be added to the generator in the form of a Boolean that determines if the generator is reversed or not. This is a common theme among the generators, we do this with a simple line of c++ code that looks like this:

Listing 3. reversing the mode in c++

```
reverse_mode = !reversemode;
```

For our application of a reversed LCG, the separate path looks like such:

Listing 4. what the reverse mode does in the generator (c++)

```
void LCG::multiply()
{
  if(reverse_mode){
      seed -= prime;
}
  mult_48_64(&seed,&multiplier,&seed);
  //^seed = seed*multiplier
  if(!reverse_mode){
    seed +=  prime;
  }
  seed &= LSB48;
}
```

It must also be important to note that the `LSB48` is a constant macro that is used as a bit-mask in order to create the modulus.

It may also be important to note that the value of `LCG::multiplier` does change depending on the reversal mode, calculating its new value using the extended Euclidean algorithm.

B. LCG64

SPRNG also uses a 64-bit LCG which offers some unique complications and room for improvement. This can be reversed in the same way that the normal LCG can with a few caveats. In order to find the multiplicative inverse of any given number in a modulus, the number must be relatively prime to the number. As [4] highlights, there must not be integers s and t such that if the number x is in modulus m, $xs! = tm$.

This statement can be made simpler by stating that the prime decomposition of x must not overlap with the prime decomposition of m. while we covered this caveat previously through the returning value of 1 in the `LCG::reverse()` function when it could have come up, for the LCG64 there exists an easier way to find an inverse of a number in the modulus of 2^{64}, Newton-Raphson iteration. Because of this different way, we must first check that the values of the prime-decomposed multiplier do not overlap with the prime decomposed modulus. This is rather simple because the primes that make up the modulus of 2^{64} are 2. Given that a number will contain 2 in its prime decomposition IFF it is even, we

can say that, so as long as the multiplier is odd, there exists a number that can be the multiplicative inverse in the given modulus.

Newton-Raphson iteration – a method of finding the multiplicative inverse of a given number inside of a modulus that is a prime power [4]. Given that 2^{64} is a prime power, this method will provide us a much faster way of finding the multiplicative inverse of a given number inside of the modulus. The code for the application of this in SPRNG looks as such:

Listing 5. newton-raphson iteration showcase for multiplicative inverses in prime power moduli

```
int LCG64::reverse() {
#ifdef LONG64
    unsigned LONG64 a = multiplier;

    // Inverse exists iff a is odd
    if ((a & 1) == 0)
        return 1;  // not invertible

    // Newton-Raphson iteration for mod 2^64
    unsigned LONG64 inv = 1; // correct mod 2

    for (int i = 0; i < 6; ++i) {
        inv *= 2 - a * inv;
    }

    multiplier = inv;
    reverse_mode = !reverse_mode;
    return 0;
#endif
}
```

This allows us to very easily and quickly modify the multiplier into being the correct multiplicative inverse for this given modulus.

C. PMLCG

Because of the prime modulus that PMLCG uses. The edge cases of numbers being relatively prime to the PMLCG don't exist. As a result, we don't have to check if the multiplier is relatively prime to the given modulus. In addition to that, due to its prime modulus resulting in seemingly more "random" outputs, there is no offset value required here. Resulting in an equation that looks like such:

$$X_{n+1} = (aX_n) \mod 2^{61} - 1 \tag{6}$$

As a result there is no "`reverse_mode`" that needs to be altered in this equation. All that needs to change is a, and a will always have a multiplicative inverse in the modulus, meaning that this generator will always work regardless of situation. Furthermore, due to the prime modulus, this allows us to use another method of finding the modular multiplicative inverse of the multiplier in the modulus, Fermat's Little Theorem [3].

$$a^{p-1} \equiv 1 \mod p \tag{7}$$

This allows us to easily calculate the multiplicative inverse for the multiplier very easily. Although it should be noted in practice that this requires a 128-bit intermediate conversion to do so effectively. This was accomplished with the following C++ code:

Listing 6. 128-bit intermediate multiplication for pmlcg using fermat's little theorem

```
static unsigned long long mod_pow_u128(
unsigned long long base,
unsigned long long exp,
unsigned long long mod) {

unsigned long long result = 1 % mod;
base %= mod;

while (exp > 0) {
    if (exp & 1) {
        // use 128-bit intermediate for the
            multiplication
        unsigned __int128 tmp = (unsigned
            __int128) result * base;
        result = (unsigned long long) (tmp %
            mod);
    }
    unsigned __int128 tmp2 = (unsigned
        __int128) base * base;
    base = (unsigned long long) (tmp2 % mod);
    exp >>= 1;
}
return result;
}
```

Furthermore do to fermat's Little Theorem we can rearrange the identity of the multiplicative inverse.

$$a \times a^{-1} \equiv 1 \mod p \tag{8}$$

Which will then change into:

$$a^{-1} \times a \equiv a^{p-1} \mod p \tag{9}$$

$$a^{-1} \equiv a^{p-1} \times a^{-1} \mod p \tag{10}$$

$$a^{-1} \equiv a^{p-2} \mod p \tag{11}$$

Lastly we can then use the 128-bit intermediate modular exponentiation function in order to compute the correct modular inverse.

Listing 7. fermat's little theorem identity in practice

```
// compute inverse using Fermat: a^(p-2) mod p
unsigned long long inv = mod_pow_u128(a, MOD -
    2ULL, MOD);

// store canonical inverse back to mult
mult = inv;
```

D. LFG

An LFG or Lagged Fibonacci Generator works slightly differently from an LCG. Instead of using values a and c, the LFG uses values j and k as offsets in a line of randomly generated numbers that must already exist to some extent. An LFG is a simpler basic equation based off the Fibonacci sequence which is defined as

$$X_n = X_{n-1} + X_{n-2} \tag{12}$$

This example of the Fibonacci sequence can be easily modified too, instead of using offsets 1 and 2, using the offsets of j and k. In what is known as an additive lagged Fibonacci

generator it is easy to create an equation that looks structurally like the sequence equation:

$$X_n = X_{n-j} + X_{n-k} \mod m \tag{13}$$

This equation shows a basic additive Lagged Fibonacci generator. However, being that a lagged Fibonacci generator can use any number of operations to combine the two "lagged" values. The values could subtract from one another in a subtractive Lagged Fibonacci generator or multiplied in a Multiplicative Lagged Fibonacci Generator as shown in SPRNG. However, the type of operation used here heavily impacts what must happen to reverse it later. We can also demonstrate the Lagged Fibonacci Generator in python code.

Listing 8. LFG example generator (assumes J < K < M)

```
def lfg(s:list, j:int, k:int, m:int) ->
    Generator[int, int, int]:
    n = len(s)
    while True:
        s.append((s[n-j] + s[n-k]) % m)
        s.pop(0)
        yield s[n-1]
```

Unlike an LCG, an LFG does not have a "set" operation that always occurs for every LFG generator. The equation for an LFG is as follows:

$$X_n = X_{n-j} * X_{n-k} \mod m \tag{14}$$

The operation here takes two known values X_{n-j} and X_{n-k} and does some operation dictated with $*$ to them. This operation does not necessarily have to be multiplication, it can be addition, subtraction, division, XOR, AND, or any other possible binary operators. The important part for reversing it is, is it reversible? The answer is simple, if there is an inverse of the operand, the generator can be reversed. This can be done by highlighting the different possible inverse operands represented all as $\cdot$. In addition to this assumption, we also must make some other assumptions first that $k > j$. and that all values between X_{n-k} through X_{n-1} are known. We can write out a theoretical one-shoe-fits-all equation for this reversed LFG:

inverse equation:

$$X_{n-k-1} = X_{n-1} \cdot X_{n-j-1} \mod m \tag{15}$$

For any one equation this can be difficult to visualize. There are many possibilities for how this can be implemented. Most simply it would be able to be done with an XOR LFG as such.

1) normal:

$$X_n = X_{n-j} \oplus X_{n-k} \mod m \tag{16}$$

2) reverse:

$$X_{n-k-1} = X_{n-1} \oplus X_{n-j-1} \mod m \tag{17}$$

This is easily done because XOR operations are their own inverse. Alternatively, some operations do have an inverse in LFGs. A notable example would be an additive LFG.

3) normal:

$$X_n = X_{n-j} + X_{n-k} \mod m \tag{18}$$

4) reverse:

$$X_{n-k-1} = X_{n-1} - X_{n-j-1} \mod m \tag{19}$$

Through this example it is apparent how the operation changes for the different LFG uses. However, it is important to look at more complex inverses and how they might be applied. Back to looking at multiplication in this context, it is easy to see how the normal vs reversed generators vastly differ in their operations:

5) normal:

$$X_n = X_{n-j} \times X_{n-k} \mod m \tag{20}$$

6) reverse:

$$X_{n-k-1} = X_{n-1} \times X_{n-j-1}^{-1} \mod m \tag{21}$$

This again must go through the extended Euclidean algorithm to define itself. However, unlike the LCG, this generator must utilize the extended Euclidean algorithm every single reverse generation. For most examples of multiplicative LFGs, it is much slower to generate numbers in reverse than in order.

For applying the use of reversed LFGs, it is much easier to reverse additive LFGs for the fact that they are both simpler and faster. The following python code highlights this:

Listing 9. function to create an inverse generator of an LFG utilizing the list of numbers, the modulus, and the offsets.

```
def inverseLFG(s:list, j:int, k:int, m:int) ->
     Generator[int, int, int]:
    n = len(s)
    while True:
        s.insert(0, (s[n - 1] - s[n - j - 1])
            % m)
        s.pop(n)
        yield s[n - 1]
        n = len(s)
```

In practice the LFG used in SPRNG utilizes two separate, parallel generators in order to prevent the higher and lower order bits from becoming "not random enough". This results in the operation in practice requiring twice as many steps and looks something like this:

Listing 10. forward parrallel lagged fibonacci generator in SPRNG

```
if(!reverse_stream) {
      //generate value at H
      r0_local[hptr_local] = INT_MOD_MASK & (
          r0_local[hptr_local] + r0_local[lptr
          ]);
      r1_local[hptr_local] = INT_MOD_MASK & (
          r1_local[hptr_local] + r1_local[lptr
          ]);

      //create value to return
      new_val = (r1_local[hptr_local] & (~1))
          ^ (r0_local[hptr_local] >> 1);

      //progress sequence
      if (--hptr_local < 0)
```

```
            hptr_local = lv - 1;
    /* skip an element in the sequence */

        if (--lptr < 0)
            lptr = lv - 1;

        //assign extra value to skip
        r0_local[hptr_local] = INT_MOD_MASK&(
            r0_local[hptr_local] + r0_local[lptr
            ]);
        r1_local[hptr_local] = INT_MOD_MASK&(
            r1_local[hptr_local] + r1_local[lptr
            ]);

        //commit *hp to the correct value
        *hp = (--hptr_local < 0) ? lv-1 :
            hptr_local;
}
```

It may be important to clarify the values used in this function that `hptr_local` = j, `lptr` = k, and `r0_local` and `r1_local` are the vectors storing the numbers in the parallel sequence, at least for our purposes.

Because the Generators utilized here are simple XOR generators, the operation between the numbers becomes its own inverse, the only problem and editing comes into how to edit the vectors used to store the numbers to store them in reverse, which becomes a rather simple operation.

```
else{

  //regress sequence
  if(++hptr_local > lv - 1)
      hptr_local = 0;

  if (++lptr > lv -1)
      lptr = 0;

  //generate value at J
  r0_local[hptr_local] = INT_MOD_MASK & (
      r0_local[hptr_local] - r0_local[lptr]);
  r1_local[hptr_local] = INT_MOD_MASK & (
      r1_local[hptr_local] - r1_local[lptr]);

  //commit *hp to the correct value and
      regress
  *hp = (++hptr_local > lv-1) ? 0 : hptr_local
      ;
  if (++lptr > lv -1)
      lptr = 0;

  //create value to return here?:
  new_val = (r1_local[hptr_local] & (~1)) ^ (
      r0_local[hptr_local] >> 1);

  //generate new value at j+1
  r0_local[hptr_local] = INT_MOD_MASK & (
      r0_local[hptr_local] - r0_local[lptr]);
  r1_local[hptr_local] = INT_MOD_MASK & (
      r1_local[hptr_local] - r1_local[lptr]);
}
```

E. MLFG

The MLFG or Multiplicative Lagged Fibonacci Generator works very similarly to the LFG. except for one small change, the operator. While the LFG uses an XOR operator between its lagged components, the MLFG utilizes a multiplication operator. This operation looks simply like:

$$X_n = X_{n-k} \times X_{n-j} \mod m \tag{22}$$

This simple equation also becomes simple in code.

Listing 11. Forward transversing c++ code in SPRNG for MLFG

```
// Forward step
hptr--;
if (hptr < 0) hptr = lv - 1;

int lptr = hptr + kv;
if (lptr >= lv) lptr -= lv;

multiply(lags[hptr], lags[lptr], lags[hptr]);
```

`hptr` here represents the j and `lptr` represents the k. the multiply equation simply takes the first two values here, multiplies them together inside of the modulus and returns the product into the last value.

The reversal of this equation can become more complicated however. Due to the nature of modulus multiplication and division, we must alter one of the values to use its multiplicative inverse in the modulus in order to properly go backwards in the generator. The equation looks as such:

$$X_{n-k} = X_n \times X_{n-l}^{-1} \mod m \tag{23}$$

In practice this can be very slow depending on the modulus, as finding the multiplicative inverse for a given number in a modulus can take a large amount of computing power. However, we must note that this Generator is created in a 2^{64} bit environment. That means, because 2^{64} is a prime power modulus, We can use Newton-Raphson iteration in order to reverse the operator in a set 4 operations every time [2]. In practice, the implementation looks like such:

Listing 12. MLFG for SPRNG (reversed) utilizing Newton Raphson iteration to reverse.

```
int lptr = hptr + kv;
if (lptr >= lv) lptr -= lv;

uint64 x = lags[hptr];      // current value to
    invert
uint64 y = lags[lptr];      // multiplier from
    forward step


uint64 inv = y;
inv = inv * (2 - y * inv);  // 1st iteration
inv = inv * (2 - y * inv);  // 2nd iteration
inv = inv * (2 - y * inv);  // 3rd iteration
inv = inv * (2 - y * inv);  // 4th iteration
// 4 iterations (Newton-Raphson)

inv &= INT_MOD_MASK;        // reduce mod 2^
    BITS

// Undo the forward multiplication
multiply(x, inv, lags[hptr]);

// Move pointer backward
hptr++;
if (hptr >= lv) hptr = 0;
```

F. CMRG

The CMRG or Combined Multiple Recursive Generator is different from the other generators so far, in the sense that it, by its definition is two different generators that combine. The first generator is simple, a 64 bit LCG which will be referred to as X. the second that SPRNG uses is a Prime Modulus Multiple Recursive Generator (PMMRG), referred to as Y. This sequence does not change between generators, the differences caused by the seed is only through the changes seen in the LCG. The equation for the CMRG and PMMRG are as follows

$$Y(n) = a \times Y(n-1) + b \times Y(n-5) \mod m \quad (24)$$

where $a = 107374182$ and $b = 104480$ and $m = 2147483647$

$$Z(n) = X(n) + Y(n) \times 2^{32} \mod 2^{64} \quad (25)$$

In theory this equation can be reversed as it is a very simple generator combined with an LCG. in practice this looks like:

$$Y(n-5) = b^{-1} \times (Y(n) - a \times Y(n-1)) \mod m \quad (26)$$

*we assume Y(n-4...n) is still known

$$Z(n-1) = X(n-1) + Y(n-1) \times 2^{32} \mod 2^{64} \quad (27)$$

However in practice not every operation has an inverse. That is true for SPRNG's implementation of the CMRG. In the generator there exists these lines:

Listing 13. Mersenne Fold in c++

```
p = (p>>31) + (p&0x7fffffff);
if(p&0x80000000) p = (p+1)&0x7fffffff;
```

What is happening here is known as a Mersenne fold. This is typically used as a tool to obscure values by reducing them into a given modulus. However, The Mersenne fold implements reduction modulo $2^p - 1$ using the identity $2^p \equiv 1 \mod 2^p - 1$. It is a lossy operation that removes information due to its many-to-one nature. Due to this loss of information, it becomes impossible to reverse without significant memory overhead or computing the entire period of the generator.

III. Discussion

Ultimately most of the generators in SPRNG are shown to be reversible without altering their current generation period. However, It becomes important to note the speed, memory, and reliability of reversing the generators in SPRNG.

All of these generators are PRNGs, meaning they follow a set structure that will generate the same numbers in the same order so long as they are initialized with the same seed. So it is safe to say aside from the non-reversable CMRG, The generators can be reliably reversed.

However, as more code becomes added into the generator, memory can begin to become an issue. Thankfully for most of the reversible generators, they only operate with one more boolean in constant memory per stream. The PMLCG has the greatest memory of all of these however because it does not require even that due to the single multiplier that is used in the PMLCG.

Lastly, the speed becomes important. For the Linear Congruential Generators in SPRNG, the speed of the generator should not change significantly as the operation to reverse the generators happens once on the reversal, inverting the multiplier and changing the equation slightly. However, that equation does not add any significant amount of operations when going in reverse.

The same can be said about the LFG. Due to its simple and easily reversible XOR operators, it does not add significant time complexity to the generation.

Lastly is the MLFG. The MLFG utilizes mutliplication in constantly changing numbers. This can cause issues because the MLFG needs to find a multiplicative inverse in a given modulus constantly. as a result, it adds a not-insignificant amount of operations to the generation of the previous number in sequence. However, in SPRNG this still isn't terrible. Due to the modulus being a prime power, the inversion of the multiplier into its inverse can be done in 4 lines of code with around 4 different binary operators per line. This lessens the impact that the MLFG has on the time complexity.

The implications of this however could mean that an MLFG could be faster and reversible with changes to its modulus. If the MLFG was in a prime modulus, Fermat's Little Theorem could easily be applied and the generator would only use one extra line in "reversed mode".

IV. Conclusion

The implications for these findings in reversible PRNGs could mean that parrallel consistent generation over a period could be easily accomplished. This would result in Use cases in Scientific Computin, Random Number Testing, and High-performance Computing.

In random number testing, a single generation stream that might have taken hours to traverse the period in would likely be able to be traversed in half the time.

In Scientific computing, traversing over a single-seeded stream and using that stream in a Monte-Carlo method could produce the same result of a normal generator in half the time.

Lastly in High-performance computing, Performing Monte-Carlo method calculations in half the time would be beneficial.

V. References

References

[1] Gene M. Amdahl. “Validity of the single-processor approach to achieve large scale computing capabilities”. In: *AFIPS Joint Spring Conference Proceedings*. Vol. 30. Atlantic City, NJ, USA: AFIPS Press, 1967, pp. 483–485.

[2] Jean-Guillaume Dumas. “On Newton–Raphson Iteration for Multiplicative Inverses Modulo Prime Powers”. In: *IEEE Transactions on Computers* 63.8 (2014), pp. 2106–2109. DOI: 10.1109/TC.2013.94.

[3] Sushil Jajodia, Pierangela Samarati, and Moti Yung, eds. *Encyclopedia of Cryptography, Security and Privacy*. Springer Nature, 2025.

[4] Donald E. Knuth. *The Art of Computer Programming, Volume 2*. Addison-Wesley Professional, 2014.

[5] D. H. Lehmer. “Mathematical methods in large-scale computing units”. In: *Proceedings of the 2nd Symposium on Large-Scale Digital Calculating Machinery*. Cambridge, MA, USA: Harvard University Press, 1949, pp. 141–146.

[6] E. Løvbåk et al. “Reversible Random Number Generation for Adjoint Monte Carlo Simulation of the Heat Equation”. In: *Springer Proceedings in Mathematics & Statistics*. Springer, 2024, pp. 451–468. DOI: 10.1007/978-3-031-59762-6_22.

[7] M. Mascagni and A. Srinivasan. “Algorithm 806: SPRNG”. In: *ACM Transactions on Mathematical Software* 26.3 (2000), pp. 436–461. DOI: 10.1145/358407.358427.

[8] S. B. Yoginath and K. S. Perumalla. “Efficient reversible uniform and non-uniform random number generation in UNU.RAN”. In: *Proceedings of the Annual Simulation Symposium (ANSS ’18)*. San Diego, CA, USA: Society for Computer Simulation International, 2018, pp. 1–10.