Parallel Pseudo-Random Number Generation Using ZUC Algorithm

Zeyang Gao
Earlham College
Richmond, Indiana
zgao14@earlham.edu

ABSTRACT

Generating random numbers to meet the needs of Monte Carlo simulations, which are widely used in both natural science and social science for simulation, is a challenge [14]. To address this challenge, this project provides a parallel pseudo-random number generator using ZUC algorithm. The output random streams were tested through NIST test suite to evaluate their randomness. In comparison, output random streams generated by LCG and LFG algorithms from the latest SPRNG library developed in 2016 were also examined by NIST, to demonstrate the level of randomness of ZUC random number generator is as good as algorithms in SPRNG library, under the measurement schema provided by NIST.

KEYWORDS

Random Number Generator, ZUC, NIST test suite

ACM Reference Format:
Zeyang Gao. 2018. Parallel Pseudo-Random Number Generation Using ZUC Algorithm. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, Article 4, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Monte Carlo simulation based on large random sampling is one of the most important research tools not only in its classical fields such as analytic physics and applied mathematics, but also in social sciences and industry [14]. Each year, about one-half of all supercomputer clock cycles are spent on Monte Carlo calculations [2]. Monte Carlo simulations are ubiquitously run in parallel. With N processors, each executing its own Monte Carlo computation, the nested result would achieve a variance N times smaller than a single processor of such calculation within the same amount of time. Although a relatively small variance usually leads to a more accurate result, it is required that the random number stream for each processor to be statistically independent[15]. Therefore, a reliable random number generator that can fit the enormous demands of random numbers for parallel Monte Carlo calculation is vital.
Apart from Monte Carlo simulations, an enormous amount of random numbers are generated as private keys for RSA encryption every day to maintain the confidentiality of personal information since the development of the Internet. A private key is a large randomly generated number that is confidential to its user and should not be predictable for security purpose[16]. Thus, the randomness of the private key is essential for the security of communication systems. Moreover, if multiple users are accessing the same server, the speed of private key generations plays an important role.
To address the above mentioned problems, a random number generator (RNG) must be both fast and secure. Truly random number generators (TRNGs) use a non-deterministic source (entropy source), along with some post-processing to produce randomness. The entropy source typically consists of some physical, unpredictable quantity, such as the noise in an electrical circuit or the timing of user processes (mouse movements on key strokes). Even though these approaches ensure the security, their productions of high-quality random numbers are extremely time consuming, making them undesirable when a large quantity of random numbers is needed. In this case, pseudo-random number generators are preferable.
Pseudo-random number generators (PRNGs) use some deterministic processes to generate bits. They usually involve a seed value, which is the initial input value to the generators. Then, the number sequence they generated is pre-determined by the seed number and their algorithm structure. PRNGs are usually quick and reproducible. However, since randomness cannot be attained by a deterministic process [1], these numbers are said to be pseudo-random. If a pseudo-random sequence is properly constructed, each value in the sequence is produced from the previous value via transformations that appear to introduce additional randomness. A series of such transformations can eliminate statistical auto-correlations between input and output. Thus, the outputs of a PRNG may have better statistical properties and be produced faster than a TRNG [4].
In this paper, we propose the ZUC parallel pseudo-random number generator (Section 4.7), a new PRNG that is based on the ZUC ciphering algorithm. We have evaluated the output random number streams using NIST test suite and compared the result with PRNGs from the Scalable Parallel Random Number Generators Library (SPRNG). Our results (Section 5) have shown that the ZUC random number generator not only produces high quality output under the NIST standard, but also invokes more randomness than any algorithms from the SPRNG library.

2 RELATED WORK

2.1 SPRNG library

Prior work has implemented the classical pseudo-random number generation algorithms (see Section 3.1). The SPRNG library is designed for the purpose of parallel Monte Carlo computations

Conference'17, July 2017, Washington, DC, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

and used parameterized pseudo-random number generators to provide random number streams to parallel processes [14]. SPRNG includes six generators (LCG, LCG-64, PMLCG, LFG, CMRG, LMFG) in C/C++/FORTRAN with MPI interface and an integrated test suite DIEHARD [8]. Although SPRNG is a great success in stochastic computations [5], its underlying pseudo-random number generation algorithms are primitive and simple. Moreover, the DIEHARD test package published in 1998 is not the contemporary standard. The current standard is the NIST test suite (section 3.2). The SPRNG library may still work well in generating random samplings for parallel Monte-Carlo computations, whereas it may not be suitable for an encryption task nowadays.

2.2 ZUC ciphering algorithm

The core of the 3rd Generation Partnership Project (3GPP) encryption standard 128-EEA3 is a stream cipher called ZUC [3]. It was designed by the Chinese Academy of Sciences and proposed for inclusion in the cellular wireless standards called "Long Term Evolution(LTE)" or "4G" in 2010 [17]. ZUC is an LFSR-based cipher uses a 128-bit key and a 128-bit initial vector. It has been proved to be able to counter multiple attacks including timing attacks, chosen IV attacks, Guess-and-Determine attacks, weak key attacks and SAT based attacks [10] [21]. Several researches from 2011 till now tried to either optimize the structure of ZUC to improve the efficiency or modify the algorithm itself to empower the security [12][20][9]. ZUC has the flexibility and potential to get better performance and higher throughput due to these improvements. In this paper, we migrated the ZUC ciphering algorithm, making it the core PRNG algorithm, to produce a new pseudo-random number generator.

3 BACKGROUND

In this section, two classic algorithms (LCG and LFG) that are responsible for the entropy in the SPRNG library are introduced. Their performances were evaluated in section 5 to compare with the ZUC random number generator.

3.1 Existing PRNG algorithms

In this section, we first introduce two classic PRNG algorithms implemented in the SPRNG library. They are the fundamental algorithms in SPRNG library that are responsible for the entropy source. Their performances are compared with ZUC generator in section 5. Additionally, we included the NIST test suite, since it is used for testing in section 5.

3.1.1 *Linear Congruential Generator.* (LCG)

LCG has been the most commonly used generator for pseudo-random number generation [14]. It was first introduced by Lehmer in 1949 [11]. LCG is driven by the linear recursion:

$$x_n = ax_{n-1} + b \pmod m \tag{1}$$

to generate random number sequence x_n with user declarations of multiplier a, additive constant b, and modulus m. However, the choice of $\{a, b, m\}$ together needs to make a purely periodic sequence a_n. Therefore, m is usually selected to be either a large prime or 2^k, a power-of-two. If m is a prime, then the period of a_n is $m - 1$. If $m = 2^k$, then the period of a_n is 2^k. For both cases, a uniform pseudo-random number in [0,1] can be produced using equation:

$$z_n = x_n/(m - 1) \tag{2}$$

$$z_n = x_n/2^k \tag{3}$$

and the initial input of the LCG, x_0, is often called the seed value. In this algorithm, either the exposure of seed value x_0 or the exposure of parameters $\{a, b, m\}$ can be fatal to the predictability of LCG output sequence (usually called output stream) x_n. This problem is inevitable to every PRNG algorithms [7].

3.1.2 *Lagged Fibonacci Generator.* (LFG)

LFG is a variation of the classic Fibonacci sequence to generate random numbers. It is described by:

$$x_n = x_{n-j} + x_{n-k} \pmod{2^m}, \quad j < k \tag{4}$$

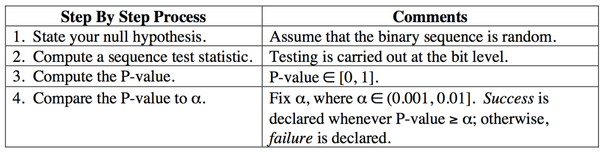
which is called ALFG (Additive LFG), with a user declarations of step size j, k, and modulus power m. In recent years, ALFG has become a popular generator for multiple reasons: (1) machines handle additions much faster than multiplications [6].(2) it is easy and cheap to implement [14]. A reasonable variation of ALFG is MLFG (Multiplicative LFG). It is described by:

$$x_n = x_{n-j} \cdot x_{n-k} \pmod{2^m}, \quad j < k \tag{5}$$

which is proved to generate streams that have superior empirical properties than ALFG streams[4]. However, MLFG runs much slower than ALFG.

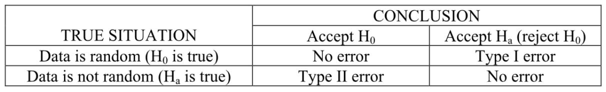
3.2 NIST test suite

The National Institute of Standard and Technology (NIST) maintains a standard randomness test package called NIST Statistical Test Suite. Like many other randomness tests [8], NIST is based on hypothesis testing. A hypothesis test is a procedure for determining if an assertion about a characteristic of a population is reasonable. In this case, the test involves determining whether or not the input stream is random. The complete evaluation process is given in Figure 1. The hypothesis summary for NIST test suite is given in Figure 2. Each test is designed to test on specific characteristics of truly random number streams. A summary of tests and the characteristics they are testing on is given in Table 1.



Step By Step Process	Comments
1. State your null hypothesis.	Assume that the binary sequence is random.
2. Compute a sequence test statistic.	Testing is carried out at the bit level.
3. Compute the P-value.	P-value $\in$ [0, 1].
4. Compare the P-value to α.	Fix α, where $\alpha \in (0.001, 0.01]$. *Success* is declared whenever P-value $\geq \alpha$; otherwise, *failure* is declared.

Figure 1: NIST Evaluation Process [18]



TRUE SITUATION	CONCLUSION	
	Accept H_0	Accept H_a (reject H_0)
Data is random (H_0 is true)	No error	Type I error
Data is not random (H_a is true)	Type II error	No error

Figure 2: Hypothesis summary for NIST test suite [18]

Table 1: NIST test summary

Statistical Test	Defect Detected
1. Frequency	Too many zeroes or ones
2. Frequency with block	Same as above
3. Cumulative Sums	Too many zeroes or ones at the beginning of the sequence
4. Longest Runs	Deviation of the distribution of long runs of ones
5. Runs	Large (small) total number of runs indicates that the oscillation in the bit-stream is too fast(too slow)
6. Rank	Deviation of the rank distribution from a corresponding random sequence, due to periodicity
7. Spectral	Periodic features in a bit-stream
8. Non-overlapping Template	Too many occurrences of non-periodic templates
9. Overlapping Template	Too many occurrences of m-bit runs of ones
10. Universal	Compressibility (regularity)
11. Random Excursions	Deviation from the distribution of the number of visits of a random walk to a certain state.
12. Random Excursion Variant	Deviation from the distribution of the total number of visits (across many random walks) to a certain state
13. Approximate Entropy	Non-uniform distribution of m-length words. Small values of ApEn(m) imply strong regularity
14. Serial	Non-uniform distribution of m-length words
15. Linear Complexity	Deviation from the distribution of the linear complexity for finite length (sub)strings

4 METHODOLOGY

In this section, we describe the three major aspects of ZUC random number generator: architecture of ZUC encryption algorithm (section 4.1-4.5), how we used ZUC as a random number generator (section 4.7) and how we parallelized it (section 4.7).

4.1 ZUC architecture

ZUC algorithm is a word-oriented stream cipher. It takes a 128-bit initial unencrypted key and a 128-bit initial vector (IV) as its input, and outputs a key-stream of 32-bit words. Each word in the key-stream is called a key-word and can be used as encrypted key.
There are two stages during the execution of ZUC: initialization stage and working stage. During the initialization stage, a key/IV initialization is performed, then the working stage will produce a

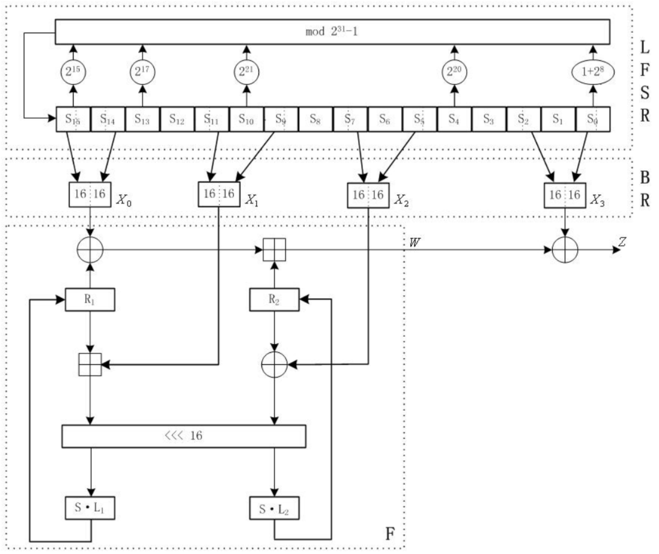


Figure 3: ZUC architecture with 3 logical layers [3]

32-bit word of output. The working stage can be executed multiple times for a single ciphering process.
ZUC has three logical layers. The top layer is a linear feedback shift register (LFSR) of 16 stages, the middle layer is for bit-reorganization (BR), and the bottom layer is a nonlinear function F.

4.2 LFSR

The linear feedback shift register (LFSR) has sixteen 31-bit registers (s0, s1,..., s15). Each register is restricted to take values from the set $\{1, 2, 3, \ldots, 2^{31} - 1\}$. During the initialization stage, the unencrypted key is loaded in these registers through evenly cutting. For example, the input 128-bit key k and 128-bit vector v is divided into:

$$k = k_0 k_1 k_2 \ldots k_{15}$$

$$v = v_0 v_1 v_2 \ldots v_{15}$$

where each k_i and v_i are 8-bit long. For the i-th register, the value $k_i d_i v_i$ will be loaded where the 15-bit long d_i can be found in Appendix A.

4.3 BR

The middle layer of the algorithm is the bit-reorganization. It extracts 128 bits from the registers of the LFSR and forms four new 32-bit words. The first three words will be used by the nonlinear function F in the bottom layer, and the last word will be involved in producing the key-stream. Let $s0, s2, s5, s7, s9, s11, s14, s15$ be the 8 registers of LFSR as in section 3.2. Then the bit-reorganization will generate four 32-bit words X_0, X_1, X_2, X_3 by:
1. $X_0 = s15_{R16} \cdot s14_{L16}$
2. $X_1 = s11_{L16} \cdot s9_{R16}$
3. $X_2 = s7_{L16} \cdot s5_{R16}$
4. $X_3 = s2_{L16} \cdot s0_{R16}$
where "$\cdot$" is the operation of bit concatenation and $s15_{L16}$ stands

for the leftmost 16 bits of $s15$ and $s14_{R16}$ stands for the rightmost 16 bits of $s14$.

4.4 F

The nonlinear function F has two 32-bit memory cells $R1$ and $R2$. Suppose the inputs of F are X_0, X_1 and X_2, each comes after the bit-reorganization, then the function F will output a 32-bit word w. The detailed process of F is as follows:
1. $w = (X_0 \text{ XOR } R1)+R2 \pmod{2^{32}}$ ($R1$, $R2 = 0$ if they are empty)
2. $w1 = R1 + X_1 \pmod{2^{32}}$
3. $w2 = R2 + X_2 \pmod{2^{32}}$
4. $R1 = S(L1(w1_{L16} \cdot w2_{R16}))$
5. $R2 = S(L2(w2_{L16} \cdot w1_{R16}))$
Step 4 and 5 concatenate half of $w1$ and half of $w2$ to generate a new 32-bit number as the input of L1, L2 function (See Appendix B). Finally, the output of L1 and L2 are passed into the S-Box to finalize the working stage.

4.5 S-box

The 32×32 S-box is composed of four 16×16 S-boxes $\{S0, S1, S2, S3\}$, where $S0 = S2$ and $S1 = S3$. The definitions of $S0$ and $S1$ can be found in Appendix C respectively.
Let x be an 8-bit input to $S0$ (or $S1$). Rewrite x into two hexadecimal digits as $x = h \cdot l$, then the entry at the intersection of the h-th row and the l-th column is the output of $S0$ (or $S1$).

4.6 ZUC execution

During the initialization stage, the algorithm will load the 128-bit initial key k and the 128-bit initial vector v into the LFSR, and set the 32-bit memory cells R1 and R2 to be all 0. Then the cipher runs the following operations 32 times:
1. BR, output:$\{X_0, X_1, X_2, X_3\}$
2. $F(X_0, X_1, X_2)$, output: W
3. $v = 2^{15}s15 + 2^{17}s13 + 2^{21}s10 + 2^{20}s4 + (1 + 2^8)s0 \pmod{2^{31} - 1}$
4. $u = W \gg 1$
5. $s16 = (v + u) \pmod{2^{31} - 1}$
6. If $s16 = 0$, then set $s16 = 2^{31} - 1$
7. Right shift all registers (Assign $s1$ to $s0$, $s2$ to $s1$, etc.)
When the initialization stage is finished, there has been enough entropy introduced into the system. ZUC can now enter its working stage by:
1. BR, output:$\{X_0, X_1, X_2, X_3\}$
2. $Z = F(X_0, X_1, X_2) + X_3 \pmod{2^{32}}$, output: Z
3. $s16 = 2^{15}s15 + 2^{17}s13 + 2^{21}s10 + 2^{20}s4 + (1 + 2^8)s0 \pmod{2^{31} - 1}$
4. If $s16 = 0$, then set $s16 = 2^{31} - 1$
5. Right shift all registers (Assign $s1$ to $s0$, $s2$ to $s1$, etc.)
A key-stream can be produced by collecting Z from step 2 and repeat the working stage. Denote Z_0 to be the output after the first execution of working stage, Z_1 the output after the second execution, etc. Then $Z = Z_0Z_1Z_2\ldots$ is the key-stream obtained from ZUC algorithm.

4.7 ZUC-RNG

ZUC random number generator is designed to take a 128-bit seed value and 128-bit initial vector as its input, these two inputs are passed into ZUC algorithm. After the fourth run of the working stage, the output stream $Z = Z_0Z_1Z_2Z_3$ is collected as the first random number N_0 generated. Then, N_0 is passed as a new key to ZUC with the same initial vector to produce the next random number. The algorithm is the following:
1. ZUC(seed, initial vector), output: $Z = Z_0Z_1Z_2Z_3$
2. $N_0 = Z$
3. ZUC(N_i, initial vector), output: N_{i+1}
The output stream will be $N_0N_1N_2\ldots$ with each N_i to be a 128-bit number. If 128-bit is too long, one can always bring the length down through $N_i \pmod{2^{32} - 1}$ which will produce a 32-bit number.
For parallelization, we let each thread executes its own ZUC system. With N processors, exactly N random numbers are generated until these numbers are sent back as keys to the LFSR. We first collect them as the first N random numbers $\{a_0, a_1, \ldots a_{N-1}\}$ generated. Then, these N numbers are passed back to each processor in their original order (see section 6 for a variation) for the next iteration of execution.

5 RESULT

In this section, we have evaluated the ZUC random number generator on multitudes of performance metrics. First, we have evaluated the output quality of ZUC using NIST test suite. Then, we provided the NIST evaluations for LCG and LFG from the SPRNG library for comparison (section 5.2). Next, we delved deeply into analyzing the impact of window length m on the a particular property of RNGs, called non-overlapping property (section 5.3). Finally, we have evaluated the impact of parallelization on the run-time of the algorithm (see section 5.4).

5.1 Evaluation Plan

Table 2: Experiment parameters set up

parameter	value
significance level	$\alpha = 0.01$
number of streams	50
length of each number	32-bits
random numbers in each stream	10 million

Table 2 provides the NIST evaluation for the ZUC random number generator. Table 3 and 4 show the results for LCG and LFG from the SPRNG library respectively.

5.2 Comparison Analysis

For ZUC, under the significance level $\alpha = 0.01$, all p-values are above α. Therefore, there is not enough evidence to reject the null hypothesis, which means that the ZUC random number generator produces random streams under the NIST standard. Moreover, the ZUC random number generator, if compared with LCG and LFG from SPRNG, is more reliable. That is because the 50 random streams generated by ZUC fail less tests than streams of LCG or LFG. Additionally, all p-values from the ZUC are even greater than

Table 3: NIST result for ZUC 10 million output stream

test	p-value	stream passed
Frequency	0.472934	50/50
Block Frequency	0.666461	50/50
Cumulative Sums	0.494930	50/50
Runs	0.698499	50/50
Longest Run	0.302109	50/50
Rank	0.666882	50/50
Fast Fourier Transform	0.437274	50/50
Non-Overlapping	0.251930	48/50
Overlapping	0.219074	50/50
Universal	0.739918	50/50
Approximate Entropy	0.338818	50/50
Random Excursion	0.112173	49/50
Random Excursion Variant	0.454981	50/50
Serial	0.534146	50/50
Linear Complexity	0.429142	50/50

Table 4: NIST result for LCG 10 million output stream

test	p-value	stream passed
Frequency	0.616305	50/50
Block Frequency	0.319084	50/50
Cumulative Sums	0.657933	50/50
Runs	0.419021	49/50
Longest Run	0.616305	50/50
Rank	0.040108	48/50
Fast Fourier Transform	0.191687	49/50
Non-Overlapping	0.035174	47/50
Overlapping	0.213309	50/50
Universal	0.058984	50/50
Approximate Entropy	0.699313	50/50
Random Excursion	0.568055	50/50
Random Excursion Variant	0.090936	50/50
Serial	0.699313	50/50
Linear Complexity	0.955835	50/50

0.1. If some certain applications require really high quality of random numbers, they might require a bigger significance level α, to make it more difficult to accept the null hypothesis and less likely to encounter a Type II error. The ZUC random number generator can even handle a significance level of $\alpha = 0.1$, whereas LCG and LFG are only capable of handling $\alpha = 0.03$ and $\alpha = 0.008$ respectively.

5.3 Non-overlapping property analysis

From the previous result, we can see that the ZUC random number generator produces high quality random streams. However, the ZUC random streams fail the Non-Overlapping test 4% of the time. In comparison, the 50 random streams generated by LCG fail 6%

Table 5: NIST result for LFG 10 million output stream

test	p-value	stream passed
Frequency	0.137282	50/50
Block Frequency	0.816537	50/50
Cumulative Sums	0.153763	49/50
Runs	0.236810	49/50
Longest Run	0.779188	50/50
Rank	0.300806	50/50
Fast Fourier Transform	0.455937	50/50
Non-Overlapping	0.008879	47/50
Overlapping	0.574903	50/50
Universal	0.267908	50/50
Approximate Entropy	0.098418	48/50
Random Excursion	0.112173	49/50
Random Excursion Variant	0.500193	50/50
Serial	0.739918	50/50
Linear Complexity	0.383827	50/50

of the time and its p-value 0.035174 is only slightly above the significance level $\alpha = 0.01$. The 50 LFG random streams completely fail the Non-overlapping test since its p-value 0.008879 is less than the significance level $\alpha = 0.01$. Therefore, it is necessary to perform further analysis over the Non-Overlapping testing for random streams.

Non-overlapping test is a test which focuses on detecting random number generators that produce too many occurrences of a given non-periodic pattern. It involves an m-bit window, used to search for a specific m-bit pattern. If the pattern is not found, the window slides one bit position. If the pattern is found, the window is reset to the bit after the found pattern, and the search resumes. Currently, the default value of m is set to be 9. However, we have $32 \cdot 10000000 = 320000000$ bits to be tested, which is extremely possible to encounter many repetitions of 9-bit strings, not because of non-randomness, but as a result of its length. Therefore, it is necessary to adjust the window length m. NIST recommended that m should be within the range of [9, 15] to maintain the validity of p-value [4]. Thus, in the following experiment, m has been set to be 10, 11, 12 to demonstrate the impact of different window length (see Table 6, 7, 8).

As the window length m gets bigger, the test results of ZUC and LCG are improved. The LFG is indifferent to the change of window length. We can still see the advantage of randomness that ZUC random number generator over other algorithms.

5.4 Run-time performance

We have collected the average run-time for 3 streams, with each stream generating 10 million 32-bits random numbers. The machine running this experiment has four 1.4 GHz Intel Core i7 processors. "-p" indicates a parallel execution with two processors involved. The result is shown in Table 9.

From the chart, we can see the time difference between ZUC and

Table 6: Non-Overlapping testing with m=10

trial	ZUC	LCG	LFG
1	0.171867, 50/50	0.236810, 47/50	0.155835, 48/50
2	0.534146, 50/50	0.657933, 50/50	0.574903, 47/50
3	0.657933, 48/50	0.739918, 49/50	0.058984, 50/50
4	0.350485, 50/50	0.851383, 49/50	0.153763, 50/50
5	0.096578, 49/50	0.350485, 48/50	0.419021, 49/50
6	0.350485, 49/50	0.574903, 48/50	0.911413, 48/50
7	0.096578, 49/50	0.171867, 49/50	0.236810, 49/50
8	0.255835, 49/50	0.657933, 49/50	0.616305, 49/50
9	0.262249, 50/50	0.350485, 48/50	0.236810, 49/50
10	0.335716, 48/50	0.791468, 50/50	0.171699, 47/50
total	492/500	488/500	486/500

Table 7: Non-Overlapping testing with m=11

trial	ZUC	LCG	LFG
1	0.911413, 50/50	0.058984, 50/50	0.350485, 46/50
2	0.574903, 49/50	0.191687, 50/50	0.494392, 50/50
3	0.122325, 50/50	0.289667, 50/50	0.191687, 47/50
4	0.319084, 50/50	0.262249, 49/50	0.851383, 47/50
5	0.699313, 49/50	0.534146, 48/50	0.319084, 50/50
6	0.040108, 49/50	0.574903, 49/50	0.383827, 48/50
7	0.657933, 50/50	0.883171, 48/50	0.494392, 48/50
8	0.213309, 49/50	0.236810, 50/50	0.534146, 50/50
9	0.455937, 49/50	0.419021, 48/50	0.075719, 45/50
10	0.534146, 50/50	0.15376, 49/50	0.171867, 49/50
total	495/500	491/500	480/500

Table 8: Non-Overlapping testing with m=12

trial	ZUC	LCG	LFG
1	0.657933, 49/50	0.657933, 49/50	0.016661, 49/50
2	0.534146, 50/50	0.350485, 49/50	0.058984, 48/50
3	0.779188, 49/50	0.122325, 48/50	0.262249, 47/50
4	0.122325, 50/50	0.008879, 48/50	0.030806, 45/50
5	0.153763, 50/50	0.419021, 50/50	0.108791, 48/50
6	0.350485, 50/50	0.574903, 50/50	0.111413, 49/50
7	0.911413, 50/50	0.213309, 49/50	0.085587, 50/50
8	0.534146, 49/50	0.616305, 50/50	0.096578, 50/50
9	0.616305, 49/50	0.075719, 50/50	0.289667, 49/50
10	0.016661, 50/50	0.494392, 48/50	0.319084, 49/50
total	496/500	491/500	484/500

SPRNG algorithms are about two to three seconds. After parallelization, the difference is further reduced.

Table 9: run-time comparison

algorithm	serial (s)	parallel (s)
LCG	22.81	13.82
LFG	21.87	12.57
ZUC	24.81	15.09

6 CONCLUSION AND FUTURE WORK

In this paper, we proposed and implemented the ZUC pseudo-random number generator. The underlying algorithm that is responsible for its randomness is the ZUC ciphering algorithm. ZUC random number generator produces random number streams with high quality, tested by NIST test suite. Moreover, we have compared the ZUC random number generator with the existing popular random number generation library SPRNG. ZUC generates better quality random numbers, compared with LCG and LFG in SPRNG but only slightly slower than them.

In the future, we could further optimize the ZUC random number generator run-time through running ZUC random number generator on GPU to continue minimizing the time difference. Moreover, there is another opportunity to introduce extra entropy during the key feedback phase (see section 4.7). We can make some reasonable permutations or bijective mappings of the output from each processor first before sending them back to the LFSR. This is an extra opportunity to introduce entropy to the system, which is possible to further extend ZUC's security. Finally, since Monte Carlo calculations and many other scientific researches also requires random samples of different distributions such as Gaussian distribution or gamma distribution [19][13], one can also add a post-processing package to ZUC random number generator to meet their need.

REFERENCES

[1] 2011. Home. (2011). https://engineering.mit.edu/engage/ask-an-engineer/can-a-computer-generate-a-truly-random-number/

[2] 2014. SPRNG: Scalable Parallel Pseudo Random Number Generators Library. (2014). http://www.sprng.org/

[3] 3GPP. 2011. *Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 128-EIA3. Document 2: ZUC Specification.* Technical Specification (TS) 36.101. 3rd Generation Partnership Project (3GPP). Version 1.6.

[4] Lawrence E. Bassham, III, Andrew L. Rukhin, Juan Soto, James R. Nechvatal, Miles E. Smid, Elaine B. Barker, Stefan D. Leigh, Mark Levenson, Mark Vangel, David L. Banks, Nathanael Alan Heckert, James F. Dray, and San Vo. 2010. *SP 800-22 Rev. 1a. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications.* Technical Report. Gaithersburg, MD, United States.

[5] Yu Bi, Gregory D Peterson, G. Lee Warren, and Robert J. Harrison. 2006. A Reconfigurable Supercomputing Library for Accelerated Parallel lagged-Fibonacci Pseudorandom Number Generation. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06).* ACM, New York, NY, USA, Article 169. https://doi.org/10.1145/1188455.1188630

[6] IBM Knowledge Center. 2011. fmadd or fma (Floating Multiply-Add) instruction. (2011). https://www.ibm.com/support/knowledgecenter/ssw_aix_61/com.ibm.aix.alangref/idalangref_fmadd_instrs.htm

[7] Yevgeniy Dodis, David Pointcheval, Sylvain Ruhault, Damien Vergniaud, and Daniel Wichs. 2013. Security Analysis of Pseudo-random Number Generators with Input: /Dev/Random is Not Robust. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security (CCS '13).* ACM, New York, NY, USA, 647–658. https://doi.org/10.1145/2508859.2516653

[8] Donald E. Knuth. 1997. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[9] Shri Ramtej Kondamuri, Nitish Kumar Gupta, and Rakesh Sharma. 2014. Modified EEA3 algorithm with improved throughput performance. (07 2014), 890–894.

[10] Frédéric Lafitte, Olivier Markowitch, and Dirk Van Heule. 2013. SAT Based Analysis of LTE Stream Cipher ZUC. In *Proceedings of the 6th International Conference on Security of Information and Networks (SIN '13)*. ACM, New York, NY, USA, 110–116. https://doi.org/10.1145/2523514.2523533

[11] Derrick H. Lehmer. 1951. Mathematical Methods in Large-scale Computing Units. In *Proceedings of the Second Symposium on Large Scale Digital Computing Machinery*. Harvard University Press, Cambridge, United Kingdom, 141–146.

[12] Zongbin Liu, Qinglong Zhang, Cunqing Ma, Changting Li, and Jiwu Jing. 2016. HPAZ: a High-throughput Pipeline Architecture of ZUC in Hardware. (01 2016), 269–272.

[13] George Marsaglia and Wai Wan Tsang. 2000. A Simple Method for Generating Gamma Variables. *ACM Trans. Math. Softw.* 26, 3 (Sept. 2000), 363–372. https://doi.org/10.1145/358407.358414

[14] Michael Mascagni. 1999. Sprng: A Scalable Library For Pseudorandom Number Generation. *Recent Advances in Numerical Methods and Applications II* (1999). https://doi.org/10.1142/9789814291071_0027

[15] Daniel V. Pryor, Steven A. Cuccaro, Michael Mascagni, and M. L. Robinson. 1994. Implementation of a Portable and Reproducible Parallel Pseudorandom Number Generator. In *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing (Supercomputing '94)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 311–319. http://dl.acm.org/citation.cfm?id=602770.602829

[16] R. L. Rivest, A. Shamir, and L. Adleman. 1978. A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Commun. ACM* 21, 2 (Feb. 1978), 120–126. https://doi.org/10.1145/359340.359342

[17] Gautham Sekar. 2011. The Stream Cipher Core of the 3GPP Encryption Standard 128-EEA3: Timing Attacks and Countermeasures. *IACR Cryptology ePrint Archive* 2012 (2011), 425.

[18] Juan Soto. 1999. Statistical Testing of Random Number Generators. In *In: Proceedings of the 22nd National Information Systems Security Conference.*

[19] David B. Thomas, Wayne Luk, Philip H.W. Leong, and John D. Villasenor. 2007. Gaussian Random Number Generators. *ACM Comput. Surv.* 39, 4, Article 11 (Nov. 2007). https://doi.org/10.1145/1287620.1287622

[20] Lei Wang, Jiwu Jing, Zongbin Liu, Lingchen Zhang, and Wuqiong Pan. 2011. Evaluating Optimized Implementations of Stream Cipher ZUC Algorithm on FPGA. In *Information and Communications Security*, Sihan Qing, Willy Susilo, Guilin Wang, and Dongmei Liu (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 202–215.

[21] Hongjun Wu, Tao Huang, Phuong Ha Nguyen, Huaxiong Wang, and San Ling. 2012. Differential Attacks against Stream Cipher ZUC. In *Advances in Cryptology – ASIACRYPT 2012*, Xiaoyun Wang and Kazue Sako (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 262–277.

APPENDIX A AUXILIARY BITS DURING KEY LOADING

$$d_0 = 100010011010111_2$$

$$d_1 = 010011010111100_2$$

$$d_2 = 110001001101011_2$$

$$d_3 = 001001101011110_2$$

$$d_4 = 101011110001001_2$$

$$d_5 = 011010111100010_2$$

$$d_6 = 111000100110101_2$$

$$d_7 = 000100110101111_2$$

$$d_8 = 100110101111000_2$$

$$d_9 = 010111100010011_2$$

$$d_{10} = 110101111000100_2$$

$$d_{11} = 001101011110001_2$$

$$d_{12} = 101111000100110_2$$

$$d_{13} = 011110001001101_2$$

$$d_{14} = 111100010011010_2$$

$$d_{15} = 100011110101100_2$$

APPENDIX B L1 AND L2

Both L1 and L2 take a 32-bit word X as input and produce a new 32-bit word. They are defined as:
$L1(X) = X$ XOR(X«2)XOR(X«10)XOR(X«18)XOR(X«24)
$L2(X) = X$ XOR(X«8)XOR(X«14)XOR(X«22)XOR(X«30)
where X«k is the word obtained after left cyclic shift k bit to X.

APPENDIX C S-BOX

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	3E	72	5B	47	CA	E0	00	33	04	D1	54	98	09	B9	6D	CB
1	7B	1B	F9	32	AF	9D	6A	A5	B8	2D	FC	1D	08	53	03	90
2	4D	4E	84	99	E4	CE	D9	91	DD	B6	85	48	8B	29	6E	AC
3	CD	C1	F8	1E	73	43	69	C6	B5	BD	FD	39	63	20	D4	38
4	76	7D	B2	A7	CF	ED	57	C5	F3	2C	BB	14	21	06	55	9B
5	E3	EF	5E	31	4F	7F	5A	A4	0D	82	51	49	5F	BA	58	1C
6	4A	16	D5	17	A8	92	24	1F	8C	FF	D8	AE	2E	01	D3	AD
7	3B	4B	DA	46	EB	C9	DE	9A	8F	87	D7	3A	80	6F	2F	C8
8	B1	B4	37	F7	0A	22	13	28	7C	CC	3C	89	C7	C3	96	56
9	07	BF	7E	F0	0B	2B	97	52	35	41	79	61	A6	4C	10	FE
A	BC	26	95	88	8A	B0	A3	FB	C0	18	94	F2	E1	E5	E9	5D
B	D0	DC	11	66	64	5C	EC	59	42	75	12	F5	74	9C	AA	23
C	0E	86	AB	BE	2A	02	E7	67	E6	44	A2	6C	C2	93	9F	F1
D	F6	FA	36	D2	50	68	9E	62	71	15	3D	D6	40	C4	E2	0F
E	8E	83	77	6B	25	05	3F	0C	30	EA	70	B7	A1	E8	A9	65
F	8D	27	1A	DB	81	B3	A0	F4	45	7A	19	DF	EE	78	34	60

Figure 4: S0[3]

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	55	C2	63	71	3B	C8	47	86	9F	3C	DA	5B	29	AA	FD	77
1	8C	C5	94	0C	A6	1A	13	00	E3	A8	16	72	40	F9	F8	42
2	44	26	68	96	81	D9	45	3E	10	76	C6	A7	8B	39	43	E1
3	3A	B5	56	2A	C0	6D	B3	05	22	66	BF	DC	0B	FA	62	48
4	DD	20	11	06	36	C9	C1	CF	F6	27	52	BB	69	F5	D4	87
5	7F	84	4C	D2	9C	57	A4	BC	4F	9A	DF	FE	D6	8D	7A	EB
6	2B	53	D8	5C	A1	14	17	FB	23	D5	7D	30	67	73	08	09
7	EE	B7	70	3F	61	B2	19	8E	4E	E5	4B	93	8F	5D	DB	A9
8	AD	F1	AE	2E	CB	0D	FC	F4	2D	46	6E	1D	97	E8	D1	E9
9	4D	37	A5	75	5E	83	9E	AB	82	9D	B9	1C	E0	CD	49	89
A	01	B6	BD	58	24	A2	5F	38	78	99	15	90	50	B8	95	E4
B	D0	91	C7	CE	ED	0F	B4	6F	A0	CC	F0	02	4A	79	C3	DE
C	A3	EF	EA	51	E6	6B	18	EC	1B	2C	80	F7	74	E7	FF	21
D	5A	6A	54	1E	41	31	92	35	C4	33	07	0A	BA	7E	0E	34
E	88	B1	98	7C	F3	3D	60	6C	7B	CA	D3	1F	32	65	04	28
F	64	BE	85	9B	2F	59	8A	D7	B0	25	AC	AF	12	03	E2	F2

Figure 5: S1[3]