A new class of scalable parallel pseudorandom number generators based on Pohlig-Hellman exponentiation ciphers

Paul D. Beale*
University of Colorado Boulder
(Dated: December 9, 2014)

We propose a new class of pseudorandom number generators based on Pohlig-Hellman exponentiation ciphers. The method generates uniform pseudorandom streams by encrypting simple sequences of short integer *messages* into *ciphertexts* by exponentiation modulo prime numbers. The advantages of the method are: the method is trivially parallelizable by parameterization with each pseudorandom number generator derived from an independent prime modulus, the method is fully scalable on massively parallel computing clusters due to the large number of primes available for each implementation, the seeding and initialization of the independent streams is simple, the method requires only a few integer multiply-mod operations per pseudorandom number, the state of each instance is defined by only a few integer values, the period of each instance is different, and the method passes a battery of intrastream and interstream correlation tests using up to 10^{13} pseudorandom numbers per test. We propose an implementation using 32-bit prime moduli with small exponents that require only a few 64-bit multiply-mod operations that can be executed directly in hardware. The 32-bit implementation we propose has millions of possible instances, all with periods greater than 10^{18}. A 64-bit implementation depends on 128-bit arithmetic, but would have more than 10^{15} possible instances and periods greater than 10^{37}.

PACS numbers: 05.10.-a, 05.10.Ln, 07.05.Tp

I. INTRODUCTION

We propose a new class of pseudorandom number generators based on Pohlig-Hellman exponentiation ciphers.[1,2] The method creates a pseudorandom stream by encrypting a simple sequence of short integer *messages*, or *plaintexts*, m_k into *ciphertexts* c_k using the transformation

$$c_k = m_k^e \bmod n, \tag{1}$$

where each independent generator is based on a prime modulus n, and an exponent e that is co-prime to $n-1$. Here and throughout, $x = y \bmod z$ means x is the remainder of y upon division by z, with $0 \le x < z$. A sequence of messages m_k with $k = 0, 1, 2, \ldots$ is chosen from the set $[0\,..\,n-1]$ using some simple pattern that uniformly samples the set. The exponentiation step eqn. (1) then gives a sequence of ciphertexts c_k that is uniformly distributed in the same range. Uniformly distributed double precision floating point values R_k on the open real interval $(0\,..\,1)$ can be formed with a floating point divide: $R_k = (c_k+1)/(n+1)$.

Most pseudorandom number generators generate the next pseudorandom integer from either the previous pseudorandom integer in the sequence, or by combining two or more pseudorandom integers from earlier in the sequence. This method is different in that the pseudorandom sequence arises from the encryption of a sequence of integer messages. In this way, it is similar to pseudorandom number generators based on block ciphers.[3] The quality of the pseudorandom sequence from our method is based on modular exponentiation being a good one-way cryptographic function that results in both the high-order bits and the low-order bits of the ciphertext being simultaneously *hard*,[2,4,5] meaning that it is computationally difficult to ascertain the message from the ciphertext. The algorithm we propose here produces excellent, long-period pseudorandom sequences even for moduli as short as 32-bits. The period of the 32-bit implementation we propose is greater than 10^{18} and has millions of possible instances. Sixty-four bit implementations have periods in excess of 10^{37}, with more than 10^{15} possible instances.

Two qualitatively different schemes have been used to create scalable systems of pseudorandom number generators for use on massively parallel supercomputers: parameter splitting and stream splitting.[6] Stream splitting is based on a single pseudorandom number generator with an extremely long period. Parallelization is accomplished by subdividing the full period into independent non-overlapping subsequences. This method requires careful seeding to ensure that no two subsequences will overlap for any feasible-length set of processes. Parameter splitting uses a single algorithm, but produces independent pseudorandom sequences by using different parameters in each process. The SPRNG web site[7,8] gives examples of parallel generators of both classes. For example, power-of-two congruential generators of the form $s_k = (a s_{k-1} + b) \bmod p$ can be parallelized by parameter splitting.[7–10] The parameters are: p is a power of 2, typically 2^{48} or 2^{64}, a is a well-tested multiplier that ensures a maximal period, and b is an odd number in $[1\,..\,p-1]$. Each process uses the same values of p and a, but the value of b is chosen to be different for each process.[11] Another widely used class of parallel generators are based on the lagged Fibonacci method.[7,8,12–14] They are usually of the form $s_k = (s_{k-q} \otimes s_{k-r}) \bmod p$ where $\otimes$ is one of the operations addition, subtraction, multiplication, or wordwise exclusive or, p is a power of two, and $q < r$ are integer parameters chosen based on primitive polynomials modulo 2.[2,15] Lagged Fibonacci generators have periods of the order of 2^r, with r being of the order of several hundred to several thousand. The state of the generator is defined by a table of r integers or r bits, which represent the most recent pseudorandom numbers in the sequence. Parallel implementation of these algorithms can be accomplished by either stream splitting or parameter splitting.[7]

The parallel pseudorandom number generator class we propose here is based on parameter splitting. In our algorithm every independent process is assigned a different prime modulus n, which produces an independent pseudorandom sequence. The pseudorandom numbers produced by the method are effectively uncorrelated within a single stream and between streams, and the period of every stream is different. The number of independent streams depends only on the number of prime numbers in the range defined by the implementation. For example in a 32-bit implementation, the prime moduli can be chosen from the set of 98,182,656 primes in the range $[2^{31} \mathinner{.\,.} 2^{32}]$.[16]

To demonstrate the relation to encryption and explain some of the useful properties of the method, the message m can be decrypted from the ciphertext c using the decryption exponent $d = e^{-1} \bmod (n-1)$, i.e. $de = 1 + q(n-1)$ for some q.[1,2,17,18] The decryption exponent d exists and is unique if $\gcd(e, n-1) = 1$, and can be determined using the extended Euclidean algorithm.[2,17,18] The decryption is based on Fermat's little theorem:[17,18] for any prime n and for all $0 < m < n$, $m^{n-1} \bmod n = 1$. Therefore

$$c^d \bmod n = m^{de} \bmod n = m^{1+q(n-1)} \bmod n = (m^1 (m^{n-1})^q) \bmod n = m \bmod n = m. \tag{2}$$

For any exponent e with $\gcd(e, n-1) = 1$, the encryption maps $[0 \mathinner{.\,.} n-1]$ onto a permutation of the same set. Therefore, any message sequence that uniformly samples $[0 \mathinner{.\,.} n-1]$ will produce ciphertexts that uniformly sample the same set. The Pohlig-Hellman algorithm is closely related to the more widely used RSA public key encryption method[19] which uses moduli that are products of two large primes. Using a composite modulus $n = pq$ creates a serous weakness for this application since every message m that is a multiple of p or q gives a ciphertext c that is also a multiple of p or q, respectively.

In the Pohlig-Hellman cryptographic scheme, the key consisting of the prime n and exponent e must be known only to the sender and receiver. Cryptography theory suggests using safe primes for the moduli, i.e. primes n for which $(n-1)/2$ is also prime.[1,2,5,20] This helps assure eavesdroppers will find it computationally difficult to take the discrete logarithm to recover the message from the ciphertext. While our empirical tests do not give noticeably better statistics for safe primes than for primes in general, the number of safe primes does not seriously limit the scaling with 32-bit moduli since there are 3,060,794 safe primes in the range $[2^{31} \mathinner{.\,.} 2^{32}]$.[21,22]

While the algorithm presented here is based on an exponentiation cipher, a cryptographically secure pseudorandom number generator requires primes with hundreds of digits.[1,4,5] Rather, we propose fast pseudorandom number generator implementations based on 32-bit or 64-bit primes. If n is chosen to be a prime number less than 2^{32}, each multiply–mod operation can be performed in hardware on a 64-bit processor with a single 64-bit multiply, and a single 64-bit mod. Multiply-mod operations with 64-bit moduli require an implementation using 128-bit arithmetic.Our algorithm leads to a new class of scalable, fast, and effective pseudorandom generators based on parameter splitting. There are several advantages to this pseudorandom number generation method.

- The algorithm is based on elementary number theory and cryptography.
- The method is trivially parallelizable by parameterization, with each instance derived from an independent prime modulus.
- Pseudorandom sequences that result from different prime moduli are independent, and have different periods.
- The method is fast since it requires only a few integer multiply-mod operations per pseudorandom number.
- The method is fully scalable on massively parallel computing clusters due to the millions of available 32-bit primes.
- By using message pattern skips derived from prime number linear congruential pseudorandom number generators, the periods of the cryptosequences are greatly extended.
- The seeding and initialization of the independent streams is simple, and it is possible to initialize each process without needing any information about the states of any of the other processes. The state of each generator is defined by only seven 32-bit numbers: m, e, n, and c, and integer pseudorandom skip parameters s, a and p defined below. All of these values can be stored in local memory, and require no global storage. The values n, e, p and a are fixed in each instance. Only the three values m, s, and c change during calls to the generator.

- The algorithm allows one to jump to forward or backward directly to any point in the pseudorandom sequence.
- The method does not require combining generators[23] or shuffling[24] to remove correlations.
- The method is fast, and the code is compact enough to allow the generator to be implemented as an in-line function for efficiency.
- The algorithm can be implemented in parallel on vector computers and GPUs.
- The method passes a battery of intrastream and interstream correlation tests using up to 10^{13} pseudorandom numbers per test.

II. MESSAGE SKIPPING ALGORITHMS

The selection of messages to be encrypted for a given modulus n can be accomplished in many different ways, but they can all be expressed in terms of an integer skip sequence $\{s_k\}$, with the skips s_k chosen from the range $[0\,..\,n-1]$:

$$m_k = (m_{k-1} + s_k) \bmod n, \tag{3a}$$
$$c_k = m_k^e \bmod n. \tag{3b}$$

Note that if the skip sequence values s_k are chosen uniformly and *randomly* (not pseudorandomly) from $[0\,..\,n-1]$, then the sequence of messages m_k forms a uniform random distribution on the same set. This is, in effect, a one-time pad cipher.[2] Since the set of ciphertexts is one-to-one with the set of messages, the sequence of ciphertexts c_k also forms a uniformly distributed random sequence. It is our goal to use a pseudorandom skip sequence that mimics a random sequence and, over the full period of the generator, uniformly samples the messages in $[0\,..\,n-1]$. But first, let's first examine the properties of cryptosequences derived from simpler skip patterns.

In encryption, it is essential to avoid or mitigate *cribs*, i.e. messages that result in easily decoded ciphertexts. For example, the messages $m = 0, 1, n-1$ are cribs for all allowed exponents e since $m^e = 0, 1, n-1$, respectively. Also, messages with $m^e \lesssim n$ and $(n-m)^e \lesssim n$ give predictable ciphertext values, so exponents $e \lesssim \log_2 n$ result in additional cribs. The problem with cribs in our context is not their appearance, but rather their *correlated* appearance. Calculational efficiency considerations make it desirable to use small exponents to reduce the number of multiply-mod operations needed to generate each pseudorandom number. Cryptographic applications often use small exponents, with cribs being avoided by padding messages to ensure no messages are close to 0 or n. For example, one might only allow messages in the range $\lfloor n/4 \rfloor \leq m \leq (n-1)/2$ by appropriately setting the leading bits. Note, however, that eliminating cribs from the message stream biases the distribution due to the elimination of ciphertexts formed from messages with $m^e < n$ and $(n-m)^e < n$, resulting in a slight but systematic under-sampling of small and large values of c. Even though the effect of this is small, we choose not to implement an algorithm that does not give a uniform distribution of ciphertexts over the full period. Our goal is to select a simple skip pattern that ensures a uniform sampling of the set of all messages, is computationally fast, has a long period, and allows use use of small exponents.

- The simplest skip sequence that uniformly samples $[0\,..\,n-1]$ is the unit skip, i.e. $s_k = 1$ for all k. This gives $m_k = (m_0 + k) \bmod n$. The full period P of the generator is obviously $P = n$. For 32-bit moduli, one can easily test the full period of the cryptosequence. In spite of the cribs near $k = 0$ and $k = n-1$, the pseudorandom sequence $c_k = k^e \bmod n$ with $0 \leqslant k < n$ passes a battery of tests even for exponents as small as $e = 9$. Naturally, the full-period sequence produces a perfectly uniform one-dimensional histogram since after n steps every value c will appear once, and only once, in the cryptosequence. Except for the one-dimensional frequency test, and other tests that are affected by the uniformity of the sampling of numbers in the range $[0\,..\,n-1]$, and in spite of the symmetry noted below, the exponentiation cipher with unit skips passes a battery of statistical correlations tests for exponents $e = 9$ and $e = 17$. By comparison, all linear congruential generators fail *all* d-dimensional correlations tests once a substantial fraction of the period has been exhausted since all d-dimensional histograms become uniform as the period of the generator is approached.[15] The exponential cipher does not suffer from this; see figures 1 and 2 which display the two-dimensional correlation patterns for a prime number linear congruential generator and a Pohlig-Hellman generator using the same prime modulus. As with most pseudorandom number generators, there is a symmetry in the pseudorandom sequence. The exponentiation cipher has the symmetry

$$(n-m)^e \bmod n = n - m^e \bmod n,$$

i.e. the ciphertexts derived from $m \in [1\,..\,(n-1)/2]$ are strongly correlated with the ones derived from $m \in [(n+1)/2\,..\,(n-1)]$, but in reverse order.

- The next simplest skip algorithm is the constant skip: $s_k = b$ with $1 < b < n-1$, so $m_k = (m_0 + kb) \bmod n$. For most values of b, the method removes sequential and closely spaced cribs. This message pattern is especially important for understanding the quality of our proposed pseudorandom skip algorithm discussed next. However, other than breaking up closely spaced cribs, the cryptosequence pattern from constant skips should not be expected to be substantially better than the unit skip cryptosequence since

$$(kb)^e \bmod n = ((b^e \bmod n)(k^e \bmod n)) \bmod n, \tag{4}$$

i.e. this is the same as the unit-skip cryptosequence except for a single constant factor $b^e \bmod n$. Like the unit skip case, there is a symmetry in the cryptosequence since $((n-k)b)^e \bmod n = n - (kb)^e \bmod n$. Empirical tests with 32-bit primes indicate the exponents as small as $e = 7$ pass the full-period tests described above for the unit skip sequence.

- Our recommended skip pattern is a pseudorandom skip produced by a prime number linear congruential pseudorandom number generator:

$$s_k = a s_{k-1} \bmod p = s_0 a^k \bmod p, \tag{5}$$

with prime modulus $p < n$. The multiplier a is chosen to be a primitive root[17,18] $\bmod p$ that delivers a full period, well-tested pseudorandom sequence. This gives pseudorandom skips s_k in the range $[1\,..\,p-1]$, with the period of the skip sequence being $p-1$.[9,15] This gives the following cryptosequence:

$$s_k = s_0 a^k \bmod p, \tag{6a}$$

$$m_k = \left(m_0 + \sum_{j=1}^{k} s_0 a^j \bmod p \right) \bmod n, \tag{6b}$$

$$c_k = m_k^e \bmod n. \tag{6c}$$

Using a pseudorandom skip sequence serves several important purposes:

  - The method provides excellent cryptosequences since the pseudorandom skips effectively eliminates the problem of correlated cribs, even when using small exponents. For reasons given below, we recommend using exponents $e = 9$ or $e = 17$.
  - Using a pseudorandom skip extends the full period of the generator to $P = n(p-1)$.
  - The method provides a uniform sampling of cipher texts over the full period of the generator. Each message $m \in [0\,..\,n-1]$, and hence each ciphertext $c \in [0\,..\,n-1]$, will appear exactly $p-1$ times in the sequence.
  - The state of each generator is defined by only seven integers: $\{m, s, c, n, e, p, a\}$.
  - Implementations using 32-bit primes are fast. They require only a few 64-bit multiply–mod operations that can be implemented in hardware on 64-bit processors.
  - The implementations suggested below pass a battery of statistical tests with up to 10^{13} pseudorandom numbers per test.

First, let's prove that the period of the generator is $P = n(p-1)$. Since a is a primitive root mod p, after $p-1$ pseudorandom skips s_k will have cycled through every value in $[1\,..\,p-1]$, so $s_{p-1} = s_0$ and $m_{k+p-1} = \left(m_k + \sum_{s=1}^{p-1} s\right) \bmod n = (m_k + p(p-1)/2) \bmod n$. This means the sequence of messages separated by multiples of $p-1$ steps in the sequence are derived from a constant skip with $b = p(p-1)/2 \bmod n$. Since p and $(p-1)/2$ are coprime to n, the constant skip b is also co-prime to n. Therefore, the state of the generator after $k = q(p-1)+r$ steps, with $q = \lfloor k/(p-1) \rfloor$ and $r = k \bmod (p-1)$, is given by

$$s_k = s_0 a^r \bmod p, \tag{7a}$$

$$m_k = \left(m_0 + qp(p-1)/2 + \sum_{j=1}^{r} \left(s_0 a^j \bmod p \right) \right) \bmod n, \tag{7b}$$

$$c_k = m_k^e \bmod n. \tag{7c}$$

This demonstrates the mathematical form of the cryptosequence across the full period of the generator. Each subsequence of ciphertexts of length $p-1$ will be different from every other subsequence, and over the full period $P = n(p-1)$, every ciphertext in $[0\,..\,n-1]$ will appear exactly $p-1$ times. Messages within $p-1$ steps of each other are related by the pseudorandom skip sequence, and messages separated by multiples of $p-1$ are related by constant skips.[25] One can use equation (7) to skip k steps forward directly to any point in the pseudorandom sequence, with skips being very fast if k is close to a multiple of $p-1$. Backward skips of k steps are accomplished by replacing a with $a^{-1} \bmod p$, subtracting rather than adding in equation (7b), and reordering equations (7a) and (7b).

Due to the mathematical structure of the sequence given by eqn. (7), one can determine the full-period d-dimensional correlations pattern for small regions without needing to exhaust the generator. The d-dimensional density of points $(c_k, c_{k+1}, \ldots, c_{k+d-1})$ over the full period is $\rho_d = n(p-1)/n^d$. One can select c_k from all points in some small chosen domain. The succeeding points are given by $c_{k+1} = (c_k^d \bmod n + s)^e \bmod n$, $c_{k+1} = (c_k^d \bmod n + s + (as) \bmod p)^e \bmod n$, etc. where the skips s take on all values in $[1\,..\,p-1]$. One checks to see which succeeding points are inside the ranges of interest. A magnified region near the origin of the full-period two-dimensional pattern for safe prime $n = 4294967087$, with $p = 2147483647$, $a = 784588716$, and $e = 9$ is shown in figure 3, with the full-period three-dimensional pattern shown in figure 4.

We tested the quality of 32-bit pseudorandom cryptosequences using a battery of independent statistical tests. We selected thousands of moduli from the set of safe primes between 2^{31} and 2^{32}, and used a single prime number linear congruential pseudorandom number generator recommended by L'Ecuyer:[9] $p = 2^{31} - 1 = 2147483647$ and $a = 784588716$. We tested uniform double precision floating point pseudorandom sequences R_k over the full period of the skip generator for more than ten-thousand different safe primes. Even using exponents as small as $e = 3$, the intrastream pseudorandom sequences pass all of our statistical tests across the period of the skip generator.

We also tested for intrastream correlations among the ciphertexts separated by $p-1$ steps across the full period of the constant skip generator. This cryptosequence is given by $c_{q(p-1)} = (m_0 + qb)^e \bmod n$, with $b = p(p-1)/2 \bmod n$ and $q = 0, 1, \,..n-1$. These constant skip sequences all pass of our the statistical tests for $e \geq 7$ for up to 2^{25} numbers, and almost all tests across the full period. (As noted earlier, all generators fail full-period tests that depend on one-dimensional histograms of the sequence, since one-dimensional histograms becomes uniform over a full period. These are the only tests that the constant skip sequence appears to fail.) Based on this, we recommend using pseudorandom skips with exponents $e = 9$ or $e = 17$, which require only four or five multiply-mod operations, respectively.

Since ciphertexts separated by $k < p-1$ steps demonstrate good intrastream statistics because of the pseudorandom skip, and ciphertexts separated by multiples of $p-1$ demonstrate good statistics with the constant skips $b = p(p-1)/2 \bmod n$, one has good reason to believe that the entire cryptosequence should pass a battery of statistical tests until the length of the pseudorandom sequence approaches the full period $P = n(p-1)$.

To test this, we performed intrastream statistical tests of the pseudorandom 32-bit skip algorithm over as large a fraction of the period, and for as many different moduli, as possible. We tested sequences of 10^{11} pseudorandom numbers using hundreds of different primes, sequences of 10^{12} pseudorandom numbers using dozens of different primes, and sequences of 10^{13} pseudorandom numbers using a few different primes. This latter test corresponds to several thousand periods of the skip sequence. The method passed every test we applied.

We also tested to ensure that the algorithm displayed lack of correlation between streams. Suppose one has N_p processes each with a different prime modulus $n^{(\alpha)}$ with $\alpha = 0, 1, \ldots, N_p - 1$, and pseudorandom sequences $R_0^{(\alpha)}, R_1^{(\alpha)}, \ldots$. Our interstream correlations tests draw the pseudorandom numbers from the N_p streams in the order $R_1^{(0)}, R_1^{(1)}, R_1^{(2)}, \ldots, R_1^{(N_p-1)}, R_2^{(0)}, R_2^{(1)}, \ldots$. We tested groupings of $N_p = 2$, 32, 1024, 32768 and 1048576 different streams, using exponents $e = 5$, 9, and 17. We also included tests using $N_p = 3,060,794$ safe primes, i.e. all of the safe primes in $[2^{31}\,..\,2^{32}]$. To test that seeding coincidences do not create spurious correlations, we performed many of the interstream tests by starting every sequence with exactly the same initial values m_0 and s_0. The interstream correlations passed every test for up to 10^{13} pseudorandom numbers per test.

III. TESTS

We first applied well-established pseudorandom number test suites DIEHARD,[26] NIST[27], and TestU01,[28] to ensure the generator passes a wide variety of tests. We then applied the following ten additional tests that produce histograms to which one can apply a chi-square test.[15] We applied these additional tests to sequences of up to 10^{13} pseudorandom numbers per test.

- One-dimensional frequency test:[15] We distribute the sequence of pseudorandom numbers into a one-dimensional histogram with 2^{20} bins, and compare the histogram to a uniform Poisson distribution.

- Two-dimensional serial test:[15] We distribute pairs of pseudorandom numbers into a two-dimensional histogram with 2^{20} bins, and compare the histogram to a uniform Poisson distribution. This measures sequential pair correlations in the sequence.
- Three-dimensional serial test:[15] We distribute triplets of pseudorandom numbers into a three-dimensional histogram with 10^6 bins, and compare the histogram to a uniform Poisson distribution. This measures sequential triplet correlations in the sequence.
- Four-dimensional serial test:[15] We distribute groups of four pseudorandom numbers into a four-dimensional histogram with 2^{20} bins, and compare the histogram to a uniform Poisson distribution. This measures sequential four-point correlations in the sequence.
- Five dimensional serial test:[15] We distribute groups of five pseudorandom numbers into a five-dimensional histogram with 2^{20} bins, and compare the histogram to a uniform Poisson distribution. This measures sequential five-point correlations in the sequence.
- Poker test:[15] We use groups of five pseudorandom numbers and count the number of pairs, three-of-a-kind etc. formed from five cards and ten denominations, and test the resulting histogram against a Poisson distribution derived from the exact probabilities. This measures a variety of five-point correlations in the sequence.
- Collision test:[15] We distribute 2^{14} pseudorandom numbers into 2^{20} bins and test the histogram of the number of collisions against a Poisson distribution derived from the exact probabilities. This measures long-range correlations in the sequence.
- Runs test:[15] We test the histogram of the length of runs of 0's ($R \leq 0.5$) and 1's ($R > 0.5$) against a Poisson distribution derived from the exact probabilities. This measures correlations of the leading bit in the sequence.
- Fourier test:[7] We use a Fast Fourier Transform to calculate the Fourier coefficients of sequences of $M = 10^{20}$ double precision floating point pseudorandom numbers,

$$\hat{x}_k = \frac{1}{\sqrt{M}} \sum_{j=0}^{M-1} x_j e^{2\pi ijk/M}, \tag{8}$$

  where $x_j = R_j - 0.5$. The real and imaginary parts of the Fourier coefficients $\hat{x}_k$ with $k = 0, \ldots, M/2 - 1$ should each be gaussian distributed about zero with variance $1/24$. Using $N \gg M$ pseudorandom numbers, we histogram the real and imaginary parts of the Fourier coefficients, and test the resulting histogram against a Poisson distribution derived from the exact gaussian distribution. This tests for correlations between pairs of pseudorandom numbers separated by up to M steps in the sequence.
- Two-dimensional Ising model energy distribution test:[29,30] We perform a Wolff algorithm[31] Monte Carlo simulation at the critical point of the two-dimensional Ising model on a 128×128 square lattice, and compare the energy histogram against a Poisson distribution derived from the exact probabilities[29,30] Since the Wolff algorithm is based on stochastically growing fractal critical clusters that can span the system, this tests for long-range correlations in the pseudorandom sequence, and has proven to be very effective at identifying weak generators.[29,30,32] Assigning an independent stream to each of the 32768 bonds in the lattice tests provides an additional test for exposing interstream correlations.

For several of these tests, we used the low-order bits to confirm they do not harbor any hidden correlations. We define passing our tests by there being no $\mathscr{P} < 10^{-8}$ events among the tests. In the three cases in which one generator gave a single $\mathscr{P} < 10^{-6}$ event for one of the tests, we repeatedly retested the same parameters n, e, p and a, with different values of m_0 and s_0 to ensure that no further $\mathscr{P} < 10^{-6}$ events appeared for any of the ten tests. Since there were tens of thousands of independent tests, we also counted the number of $\mathscr{P} < 10^{-4}$ events in our samples to confirm that the number was consistent with the expected number.

IV. IMPLEMENTATION

The algorithm for generating uniform double precision floating point pseudorandom numbers R on the open interval $(0\,..\,1)$ is:

$$
\begin{aligned}
&s := (as) \bmod p && \text{(9a)}\\
&m := (m+s) \bmod n && \text{(9b)}\\
&c := m^e \bmod n && \text{(9c)}\\
&R := (c+1)/(n+1) && \text{(9d)}\\
&\text{RETURN } R && \text{(9e)}
\end{aligned}
$$

In a 32-bit implementation, all of the operations in steps (9a)-(9c) of the the algorithm can be implemented using 64-bit unsigned integer arithmetic, which can be executed in hardware on 64-bit processors. This algorithm generates the sequence given by equations (7). For efficiency, one can precalculate the double precision floating point value $1.0/(n+1)$ and perform a floating point multiplication instead of a floating point divide in step (9d). For small exponents like $e = 9$ or $e = 17$, the code is compact and simple enough to be implemented as an in-line function.

For a multiprocessor environment, each process can be assigned an independent prime using well-established primality tests. The Rabin-Miller test,[33,34] which is the same as Algorithm P in Knuth,[15] provides a simple probabilistic test for primality. Every odd prime $n = 1 + 2^r s$ with s odd satisfies one of the following conditions for every base g in $[2\,..\,n-2]$: either $g^s \bmod n = 1$, or $g^{2^j s} \bmod n = n-1$ for some some j in the range $0 \leq j < r$. A composite number n satisfying these criteria is called a strong pseudoprime to base g. For any odd composite, the number of bases g that will pass a single Rabin-Miller test is less than $n/4$, so if the test is applied repeatedly with M randomly chosen bases in $[2\,..\,n-2]$, the probability that a composite will pass every test is less than 4^{-M}.[15,33,34] Better yet, the Rabin-Miller test can quickly and deterministically identify all primes below 2^{64}. There are no composite numbers below 2^{64} that are strong pseudoprimes to all of the twelve smallest prime bases ($g = 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37$).[35–39] Therefore any number below 2^{64} that passes the Rabin-Miller test for all twelve of these bases is prime. Likewise, any number below 2^{32} that passes the Rabin-Miller test for all of the five smallest prime bases ($g = 2, 3, 5, 7, 11$) is prime. For efficiency, one first checks to see if any small primes divide n before applying the Rabin-Miller test.

One can vary the pseudorandom skip parameters p and a used in different instances, but there is not a scalable number of well-tested prime number linear congruential generators from which to choose. L'Ecuyer[9,23] gives several examples. In all of our tests below, we used $p = 2147483647$ and $a = 784588716$.[9]

Since each process α is assigned an independent prime $n^{(\alpha)}$, the statistical independence of the different streams is not especially sensitive to the selection of the initial values $m_0^{(\alpha)}$ and $s_0^{(\alpha)}$, but one should seed each process with a different initial message, and at a different point in the pseudorandom skip sequence. For example, one could use the system time variable and the process identifier α to construct unique values of $n^{(\alpha)}$, $s_0^{(\alpha)}$, and $m_0^{(\alpha)}$ for each of the N_p processes. One might use equation (7) to skip to a specific starting point in the sequence relative to some starting point such as $m_0 = 0$ and $s_0 = 1$.

Even if the state values for two different prime moduli happen to get synchronized with the same values of m and s, the values for the ciphertexts for different moduli will be different due to the encryption step, and the message synchrony is removed after a few skips, even if the skips remain synchronized. Our interstream correlations tests concentrated on cases where every process was initialized with the same values of m_0 and s_0 to ensure accidental synchronization of the messages and skips does not create observable interstream correlations.

The implementation for 32-bit moduli is easy and fast since all of the multiply-mod operations can be implemented in hardware using standard 64-bit unsigned arithmetic on 64-bit processors. For example in C, the 32-bit multiply-mod $z = xy \bmod n$ is implemented as simply z=(x*y)%n, where x, y, n, and z are unsigned 64-bit integers, which ensures the intermediate value x ∗ y is less than 2^{64}. The quality of the pseudorandom sequences does not appear to be dependent on values of the prime moduli, but we recommend using safe primes selected from $[2^{31}\,..\,2^{32}]$ unless the number of processes exceeds 3,060,794, the number of safe primes in that range. If the implementation requires more than 3,060,794 instances, there are 49,091,941 primes n in $[2^{31}\,..\,2^{32}]$ with $e = 9$ coprime to $n-1$, and there are 92,045,560 primes in that range with $e = 17$ coprime to $n-1$. There are not a scalable number of well-tested 31-bit prime number pseudorandom skip generators, but one can use the handful of ones available in the literature[9,23] to substantially extend the number of possible instances.[40] The full period of each generator is $P = n(p-1) > 2^{62} \simeq 4.6 \times 10^{18}$. This 32-bit implementation passes all of our intrastream and interstream correlations tests for $e = 9$ and $e = 17$, for up to 10^{13} pseudorandom numbers per prime modulus. The exponent $e = 9$ requires only five multiply-mod operations per pseudorandom number, one for the skip and four for the exponentiation, and the exponent $e = 17$ requires only six multiply-mod operations.

If far longer periods or far more instances are needed, one can implement the algorithm using 64-bit primes. There are approximately 3×10^{15} safe primes in $[2^{63} \,..\, 2^{64}]$. L'Ecuyer[9] provides suitable pseudorandom skip parameters near 2^{63} so the period of each process is $P = n(p-1) > 2^{126} \simeq 8.5 \times 10^{37}$. Using 64-bit prime moduli results in a speed penalty with current processors since the multiply–mod operations need to be implemented in using 128-bit unsigned integers. Since a single process is unlikely to exhaust a 63-bit pseudorandom skip, one might consider using smaller encryption exponents for efficiency. Preliminary statistical tests indicate the method works well for 64-bit safe prime moduli with exponents as small as $e = 3$.

V. CONCLUSION

We propose a new class of parallel pseudorandom number generators based on Pohlig-Hellman exponential ciphers. The method creates pseudorandom streams by encrypting simple sequences of integer messages. The method is fully scalable based on parametrization since each process can be assigned a unique prime modulus. By using pseudorandom skips among messages, one can use small exponents and the period is greatly extended. For 32-bit implementations, only a few 64-bit multiply–mod operations in hardware are needed per pseudorandom number. There are millions of possible independent instances, and the period of each instance is is greater than 10^{18}. We have tested thousands of different pseudorandom streams for intrastream and interstream correlations using up to 10^{13} pseudorandom numbers per test, and all pass a battery of statistical tests. A 64-bit implementation would have more than 10^{15} possible instances and periods greater than 10^{37}. Sample C++ code of a 32-bit multi-processor implementation of the Pohlig-Hellman pseudorandom number generator with pseudorandom skip can be found at `http://spot.colorado.edu/~beale/Pohlig-Hellman`.

VI. ACKNOWLEDGEMENTS

We thank Matt Glaser, Rudy Horne, Nick Mousouris, Ethan Neil, Robert Blackwell, John Black, and David Grant for helpful discussions. This work utilized the Janus supercomputer, which is supported by the National Science Foundation (award number CNS-0821794) and the University of Colorado Boulder. The Janus supercomputer is a joint effort of the University of Colorado Boulder, the University of Colorado Denver, and the National Center for Atmospheric Research.

---

[*] Electronic address: paul.beale@colorado.edu
[1] S. Pohlig and M. Hellman, IEEE Transactions on Information Theory **(24)**: 106 (1978).
[2] B. Schneier, *Applied Cryptography*, (Wiley, New York, 1994).
[3] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical Recipes: The Art of Scientific Computing*, (Cambridge, New York, 1992), second edition,
[4] M. Blum and S. Micali, SIAM J. Comput., **13**, 850, (1984).
[5] S. Patel and G.S. Sundaram, CRYPTO98, LNCS **1462**, 304, (1998).
[6] H. Bauke and S. Mertens, Phys. Rev. E, **75**, 066701 (2007).
[7] M. Mascagni and A. Srinivasan, ACM Transactions on Mathematical Software **26**, 436 (2000).
[8] The Scalable Parallel Random Number Generators Library (SPRNG), `http://www.sprng.org`
[9] P. L'Ecuyer, Mathematics of Computation **68**, 249 (1999).
[10] M. Mascagni, Parallel Computing **24**, 923 (1998).
[11] The period of every such generator is p, and the low order bits are highly correlated within each stream, as well as between streams.
[12] M. Mascagni, M. L. Robinson, D. V. Pryor and S. A. Cuccaro *Springer Verlag Lecture Notes in Statistics***106**, 263 (1995).
[13] M. Mascagni, S. A. Cuccaro, D. V. Pryor and M. L. Robinson Journal of Computational Physics **119**, 211 (1995).
[14] M. Matsumoto and T. Nishimura, ACM Transactions on Modeling and Computer Simulation **8(1)**, 3, (1998).
[15] D. Knuth, *The Art of Computer Programming*, vol. 2, (Addison-Wesley, Reading, Massachusetts 1999).
[16] *The On-Line Encyclopedia of Integer Sequences* `http://oeis.org/A036378`
[17] T. Koshy, *Elementary Number Theory and Applications*, (Academic, San Diego, 2002).
[18] J.H. Silverman, *A Friendly Introduction to Number Theory*, (Pearson, New York, 2006).
[19] R. Rivest, A. Shamir, and L. Adelman, Communications of the ACM, **21**, 120 (1978).
[20] This makes $(n-1)/2$ a Sophie-Germain prime.
[21] *The On-Line Encyclopedia of Integer Sequences* `http://oeis.org/A211395`
[22] *The On-Line Encyclopedia of Integer Sequences* `http://oeis.org/A211397`

[23] P. L'Ecuyer, Computations of the ACM **31**, 741 (1988).

[24] C. Bays and S.D. Durham, ACM Transactions on Mathematical Software, **2**, 59, (1976).

[25] Equation (7) also shows why the method requires $p < n$. If $n = p$, one can evaluate the sum in eqn. (7b) in closed form to give $m_k = (m_0 + s_0 a(a-1)^{-1}(a^k - 1)) \bmod p$. Therefore $m_{p-1} = m_0$ and $s_{p-1} = s_0$, so period of the generator becomes $P = p - 1$. Furthermore, the cryptosequence is of the form $c_k = c_0 a^{ek'} \bmod p$, where $k' = k - k_0$ and k_0 is the element where $m_{k_0} = 0$. This is simply the *same* prime number linear congruential generator with modulus p, but skipping ahead e steps at a time. Even if the linear congruential skip generator is of high-quality, skipping ahead will almost certainly yield a lower-quality pseudorandom sequence.[15]

[26] G. Marsaglia, DIEHARD: a battery of tests of randomness, (1996); see `http://stat.fsu.edu/~geo/diehard.html`.

[27] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Van- gel, D. Banks, A. Heckert, J. Dray, and S. Vo, NIST special publication 800-22, National Institute of Standards and Technology (NIST), Gaithersburg, Maryland, USA, 2001; see `http://csrc.nist.gov/rng/`.

[28] U01 Test; see `http://www.iro.umontreal.ca/~simardr/testu01/tu01.html`

[29] P.D. Beale, Phys. Rev. Lett. **76**, 79 (1996).

[30] R.K. Pathria and P.D. Beale, *Statistical Mechanics*, (Academic, Boston, 2011), third edition.

[31] U. Wolff, Phys. Rev. Lett. **62**, 361 (1989).

[32] A.M. Ferrenberg, D.P. Landau and Y.J. Wong, Phys. Rev. Lett. **69**, 3382 (1992).

[33] G.L. Miller, Journal of Computer Systems Science **13**, 300 (1976).

[34] M.O. Rabin, Journal of Number Theory **12**, 128 (1980).

[35] C. Pomerance, J. L. Selfridge and S. S. Wagstaff, Jr., Mathematics of Computation **35**, 1003 (1980).

[36] *The On-Line Encyclopedia of Integer Sequences* `http://oeis.org/A014233`

[37] Zhenxiang Zhang, Math. Comp. **70**, 863 (2001).

[38] Y. Jiang and Y. Deng, `http://arxiv.org/abs/1207.0063v1`

[39] See `http://mathworld.wolfram.com/Rabin-MillerStrongPseudoprimeTest.html`

[40] One might consider using power-of-two generators for the skip generator since a scalable number of them are available,[9] but we have not investigated the effects of the strong correlations in the low-order bits.

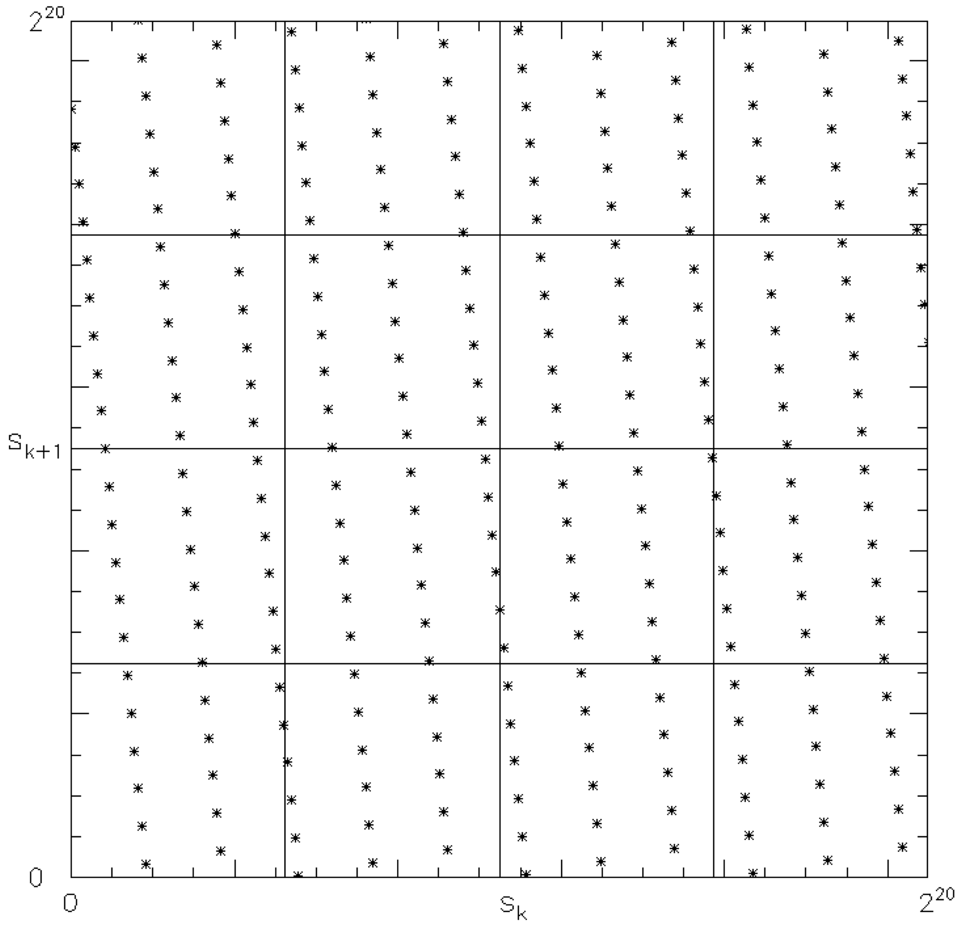

FIG. 1: Magnified region near the origin of the full-period s_{k+1} vs. s_k pattern delivered by a prime number linear congruential pseudorandom number generator (eqn. (5)) with $p = 2^{32} - 5 = 4294967291$ and $a = 279470273$.[9] The axes cover the range $[0 \mathinner{.\,.} 2^{20}]$, i.e. a linear magnification factor of 4096. The 16 subsquares have area $\Delta x^2 = (2^{18})^2$ each. As with all linear congruential generators, every d-dimensional full-period pattern forms a perfect lattice,[15] so the number of points in each cell is almost exactly $N_{cell} = n/\Delta x^d \approx 16$ events in this case. A full-period lattice occurs for all congruential generators, and in all dimensions.

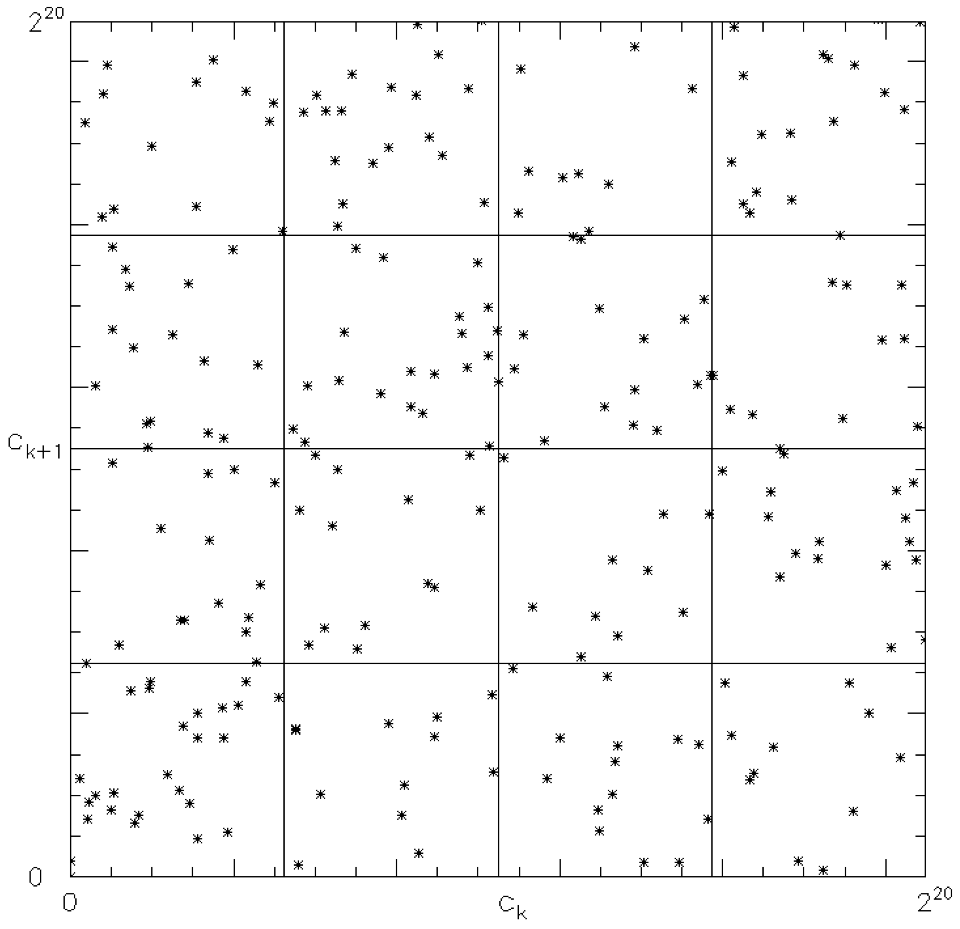

FIG. 2: Magnified region near the origin of the two-dimensional full-period c_{k+1} vs. c_k pattern for a Pohlig-Hellman exponential cipher (eqn. (3)) with $n = 2^{32} - 5 = 4294967291$, $e = 9$ and unit skip, i.e. $m_k = 0, 1, 2, \ldots, n-1$. The axes cover the range $[0\,..\,2^{20}]$, i.e. a linear magnification factor of 4096. The 16 subsquares have area $\Delta x^2 = (2^{18})^2$ each. Unlike linear congruential generators, the distribution of points in the cells approximate a Poisson distribution with $N_{cell} = n/\Delta x^d \pm \sqrt{n/\Delta x^d} \simeq 16 \pm 4$ in this case. The unit-skip correlated cribs $\{(0,1),\ (1, 2^9 = 512),\ (512, 3^9 = 19683),\ (19693, 4^9 = 2^{18} = 262144\}$ appear in the lower left square.

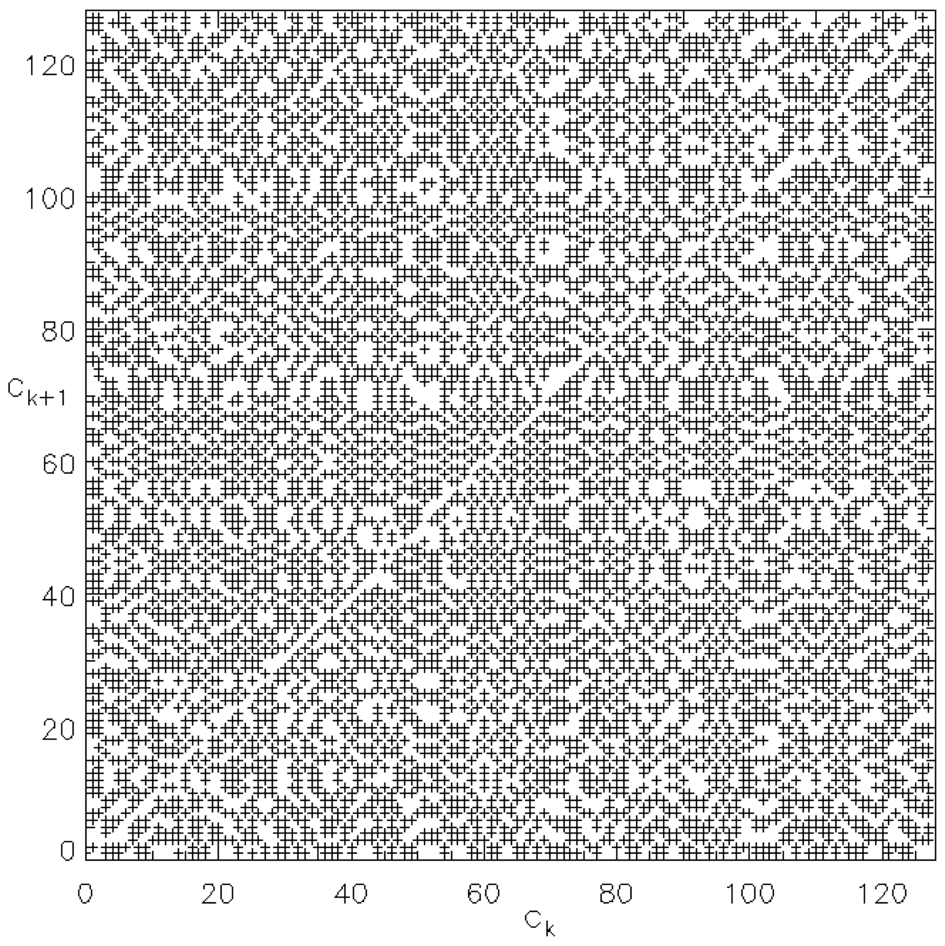

FIG. 3: This figure shows (c_k, c_{k+1}) pairs in a magnified two-dimensional square region over the full period of the generator for a pseudorandom skip with $n = 4294967087$, $p = 2147483647$, $a = 784588716$, and $e = 9$. The region shown is a $2^{20} \times 2^{20}$ square closest to the origin, Every prime n–e–p combination gives a different full-period pattern, but the two-dimensional full-period pattern is independent of the value of primitive root a. The average full-period density of the pairs in two dimensions is $\rho_2 = (p-1)/n \simeq 0.5$. The absence of any points on the diagonal with $c_{k+1} = c_k$ is because $0 < s_k < p$ for all k. Even though the full period is $P = n(p-1) \simeq 4.6 \times 10^{18}$, it is feasible to determine the local full-period pattern without exhausting the generator since $c_{k+1} = ((c_k^d \bmod n) + s_{k+1})^e \bmod n$. One selects only ciphertexts c_k in the chosen domain while testing all skips s_{k+1} in $[1\,..\,p-1]$ to see which ones give c_{k+1}'s in the chosen range. Primes with $p \approx n$ give patterns that fill in nearly every integer pair in the two-dimensional pattern, except those along the diagonal.

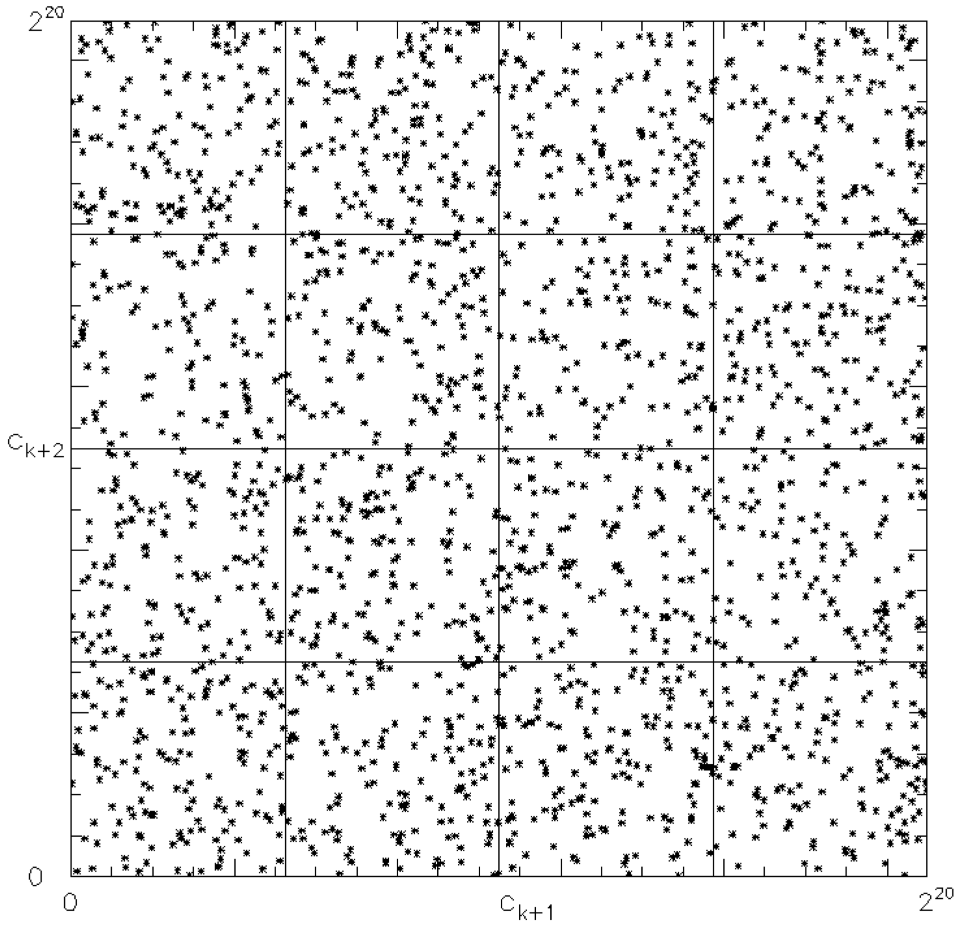

FIG. 4: This figure shows (c_k, c_{k+1}, c_{k+2}) triplets in a magnified rectangular three-dimensional region over the full period of the generator for a pseudorandom skip with $n = 4294967087$, $p = 2147483647$, $a = 784588716$, and $e = 9$. The region shown is a $16 \times 2^{20} \times 2^{20}$ rectangular slab closest to the origin, with the short first dimension projected out. Even though the full period is $P = n(p-1) \simeq 4.6 \times 10^{18}$, it is feasible to determine the local full-period pattern without exhausting the generator since $c_{k+1} = ((c_k^d \bmod n) + s_{k+1})^e \bmod n$ and $c_{k+2} = ((c_k^d \bmod n) + s_{k+1} + (as_{k+1}) \bmod p)^e \bmod n$. One selects only ciphertexts c_k in the chosen domain while testing all skips s_{k+1} in $[1 \,..\, p-1]$ to see which ones give c_{k+1} and c_{k+2} in the chosen range. Since the three-dimensional density of triplets is $\rho_3 = (p-1)/n^2$, each $16 \times 2^{18} \times 2^{18}$ slab, and each $1 \times 2^{20} \times 2^{20}$ slice should follow a Poisson distribution with $N_{cell} = \rho_3 V_{cell} \pm \sqrt{\rho_3 V_{cell}} \simeq 128 \pm 11.3$ points in each region. The actual results here are 127.25 ± 14.0 in the sixteen slabs, and 127.25 ± 10.7 in the sixteen slices. The total number of points in the slab is 2036, consistent with the expected number 2048 ± 45.3.