5-1-1996

The use of non-commutative algebra in cryptographically secure pseudo-random number generators

Brian M. McKeever
University of Richmond

Mathematics and
Computer Science
Mck

The Use of Non-Commutative Algebra in Cryptographically Secure Pseudo-Random Number Generators

Brian M. McKeever
Honors thesis[1]
Department of Mathematics & Computer Science
University of Richmond

May 1, 1996

[1]Under the direction of Dr. Gary R. Greenfield

Abstract

This thesis begins with a general overview of pseudo-random number generators and some of their applications. This thesis then describes their applications to cryptography, and some additional requirements imposed by cryptography. This thesis then provides an introduction to the ring of quaternions, and discusses how they can be included in pseudo-random number generators. Finally, this thesis provides a description of the performance of these generators.

This paper is part of the requirements for honors in mathematics. The signatures below, by the advisor, a departmental reader, and a representative of the departmental honors committee, demonstrate that Brian McKeever has met all the requirements needed to receive honors in mathematics.

Gary R. Gruenfeld

(advisor)

Kathy Hoke

(reader)

[illegible]

(honors committee representative)

I. Introduction

In many computer applications, it is desirable to be able to produce a random number. In general, these are applications where the designer has tried to model the randomness of the real world. A simulated card game in which the user is dealt the same hand every time is not very interesting. One solution is to build specialized hardware that connects the computer to the outside world. The computer could then choose random numbers based on radioactive decay rates, or on the static between radio stations. A very different solution is to produce a sequence of "random looking" numbers from an initial seed.

One of the first attempts at this was by Von Neuman [5]. His method is known as the Middle-Squares Generator:

- Start with a four digit integer
- Square it
- Pull out the middle four digits
- Repeat as needed

Because these types of algorithms are deterministic systems trying to emulate randomness, we call them *pseudo-random number generators* or PRNGs.

As an example of using the Middle-Squares Generator, if we choose to start with 5604, we square it to get 31404816, so our new number is 4048. The next number in the sequence would be 3863, since $4048 \times 4048 = 16386304$. The sequence continues 9227, 1375, 8906 .... This type of generator has the following advantages: the numbers certainly *appear* to be random, the next number can be easily produced, and the only knowledge we need (beyond the algorithm) is the present number. This generator does have its flaws. For one, the numbers can stop appearing random. If we continue the previous example long enough, we find the sequence 8441, 2504, 2700, 2900 .... Every successive iteration will produce a multiple of 100. Worse than this, the sequence will degenerate to 4100, 8100, 6100, 2100, 4100 ...— we have

come upon a short cycle.
This highlights the main drawback to the Middle Squares Generator — we would like to avoid falling into traps like this, but the algorithm is sufficiently complex that it is difficult to say which innocuous-looking starting numbers will produce undesirable results. The solution to this last problem is to use an algorithm that is easier to analyze.

The Middle-Squares Generator demonstrates one of the areas in which a sequence of numbers based on a generator using an algebraic expression is vastly different from a sequence of truly random numbers. At some point, the generator must repeat itself. To minimize the impact that this has, we want to design generators that will go as long as possible without repeating.

The Linear Congruence Generator has the form:

$$x_{n+1} = ax_n + c \bmod m.$$

In addition to the strengths of the Middle-Squares Generator, this system has the added advantage that we can give conditions on our choices of a,c, and m so that we have maximum period before repeating. As stated in [5], the necessary and sufficient criteria are

- c and m are relatively prime
- $a - 1$ is a multiple of p, for all $p \mid m$
- $4 \mid (a - 1)$ if $4 \mid m$

A linear congruence generator satisfying these requirements will cycle through all m numbers between 0 and $m - 1$ before repeating. For implementations of this generator, m is generally taken to be the largest number that can be held in the computer's register, so the reduction modulo m is taken care of by overflow. The Linear Congruence Generator is the PRNG most commonly used for simulation purposes, since it is fast and has long period.

II. Random Numbers and Cryptography

The cipher systems used to encrypt data fall into two broad classes: stream ciphers and block ciphers. The main distinction is in how much plaintext material they encrypt at a time. Block ciphers, such as DES or IDEA [8], first divide the plaintext into blocks (of typically sixty-four bits) and then individually encrypt these blocks with a common key. Stream ciphers, on the other hand, encrypt plaintext in smaller units — at most a few bits at a time. The two types of ciphers draw their security from different sources. Block ciphers have at their heart a complicated, non-invertible function that operates on the plaintext. Also, block ciphers may shuffle around the plaintext bits. Stream ciphers are built around a cryptographically secure pseudorandom number generator, or CSPRNG. The generator is used to produce a key bit for each plaintext bit, and the two are added modulo 2.

The Vernam Cipher is an encryption scheme that offers provably unbreakable security. All that is needed is a random key that is as long as the plaintext. The i^{th} ciphertext bit C_i is produced by:

$$C_i = K_i + P_i \bmod 2,$$

where P_i is the i^{th} plaintext bit and K_i is the i^{th} key bit. As long as the key is truly random, an attacker can never find the plaintext from the ciphertext. If the key is truly random, every possible key is equally likely, so every possible plaintext could have produced a given ciphertext with equal probability, which means that an attacker who has intercepted the ciphertext can gain no knowledge about the original message. The primary drawback of this system is that it requires as much key as plaintext. That means that in addition to transmitting the ciphertext, the sender must get an equal amount of key material to the intended recipient.

In essence, stream ciphers represent an attempt to implement the Vernam Cipher with a shorter key. Instead of having equal amounts of key material and message, as required by the Vernam Cipher, a stream cipher requires only a small amount of key, and from that produces an arbitrary length output. This output can then be used as key material in the manner described above. Because of the determinism of a PRNG, the output is completely *non-random*, and we lose the unbreakability of the Vernam Cipher. Any regularity in the output of a CSPRNG is a potential weakness to be exploited by an attacker.

The primary feature that separates a pseudo-random number generator from a cryptographically secure pseudo-random number generator is predictability. Ideally, we would like it to be the case that no matter how many previous outputs an attacker has access to, he has no advantage to predicting the next output. In practice, this can never be the case. Since these algorithms are to be run on finite state machines, their corresponding periods are also finite, which allows an attacker to eventually be able to predict the sequence flawlessly. However, it is often possible to "break" a generator long before it has repeated itself. For example, suppose we are watching the output of a linear congruence generator, and we know the value of m, but not a or c. We can solve for a after seeing only three consecutive outputs: Let the outputs be x_0, x_1, and x_2. Then we know that $x_1 = ax_0 + c \bmod m$ and $x_2 = ax_1 + c \bmod m$. Then $x_2 - x_1 = a(x_1 - x_0) \bmod m$, from which we get the value of a, and can substitute to find c. The process of breaking the generator is more difficult if we don't know m, but it can still be done with relative ease.

In light of these sorts of attacks, we do two things. First, we make the period very long, so that someone cannot see every output. Also, we strive to make it more difficult to set up and solve equations to find the "parameters" of our generator.

There are certain advantages to using a stream cipher over a block cipher. If we imagine that we own a bank that has a branch office, we want to be able to communicate securely between the two, in order to authorize transactions. If we use a block cipher, we will be responsible and change the key fairly often. An attacker can go to our branch office and make a deposit to his account. The branch office will then send an encrypted record of the transaction to the main bank. If the attacker can listen in on the line between the branch office and the main headquarters, he can record the message sent. Then he can send the message again later. If he does so before we change the key, the main office will decrypt the message, and thinking it is a record of a new transaction, the bank will credit another deposit to his account without a second transaction having taken place.

If, on the other hand, we were using a stream cipher, this type of attack (a "replay" attack) would not work. If the attacker sent the encypted communication that he had recorded, which had been valid earlier, the message would come across as nonsense since the message would have been encrypted with a different set of key stream bits. The price we pay is that the CSPRNGs at the two offices would now be out of synch with each other — any subsequent real messages we send would also come out as nonsense, for essentially the same reason.

In practice, we could adjust our protocol to make this sort of attack against a block cipher infeasible. Remedies include adding a time-stamp or a serial number to the message. But this example does illustrate that there are some strengths inherent to stream ciphers.

The most generalized stream cipher consists of a "next-state function" f_s, and an "output function" f_o. A machine is then set into an initial state, σ_0. The machine produces an output $z_0 = f_o(\sigma_0)$, and proceeds to a new state $\sigma_1 = f_s(\sigma_0)$. The first plaintext bit is then encrypted with z_0. This cycle is repeated until all the plaintext has been encrypted. In the most general case, f_s and f_o can both be keyed, and key can be introduced into the internal state at any time. In practice, key is used only to set σ_0 and f_s.

Often, it is enlightening to imagine the different internal states as points in space. We can create paths between them in accordance with the next-state function, so that there is a path connecting σ_i to σ_j if and only if $\sigma_j = f_s(\sigma_i)$. Any point, then, will lie on either an arc or a cycle, and every arc will lead to a cycle. It would seem difficult, however, to determine whether a given point is on an arc or a cycle. The most obvious approach is to iterate the generator from the given point, and store every state the generator has been in. If the first repeated state is the original one, then the original state is on a cycle. Otherwise, the state is on an arc. Unfortunately, this approach requires that we compare the current state with every previous state, which will be a time-consuming process.

In [7], Ritter suggests a clever way to make this determination without having to store every previous state. His method consists of initializing two instances of the same generator, $G1$ and $G2$, to the same state σ_0. Then, we clock $G2$ twice as fast as $G1$, and at each step we compare the internal states of the two generators. If $G1$ and $G2$ are ever in the same state, σ_k, then we know that the current state lies on a cycle. In addition, we can find the period of the cycle by counting the number of iterations of $G1$ it takes before $G1$ is again in state σ_k. Furthermore, if we continue to iterate $G1$ and it reaches state σ_k again before reaching σ_0, then we know that σ_1 was on an arc, and we can calculate the arc length between σ_0 and the cycle containing σ_k.

III. Non-Commutative Algebra

This treatment is based on the one found in [4].

The ring of quaternions was invented by Hamilton as a generalization of the complex numbers from two to four dimensions. Recall that the complex number $\alpha = a + bi$ can be represented as

$$\alpha = \begin{pmatrix} a & b \\ -b & a \end{pmatrix},$$

and that this representation defines an isomorphism between the complex numbers and a subring of the two-by-two matrices with real entries. Similarly, the quaternion number $\alpha = a + bi + cj + dk$ can be represented as

$$\alpha = \begin{pmatrix} a+bi & c+di \\ -c+di & a-bi \end{pmatrix}.$$

Using this defining relation, we can determine how multiplication of quaternions works: The quaternion

$$0 + i + 0j + 0k = \begin{pmatrix} i & 0 \\ 0 & -i \end{pmatrix}$$

satisfies

$$\begin{aligned} i^2 &= (0+i+0j+0k)(0+i+0j+0k) \\ &= \begin{pmatrix} i & 0 \\ 0 & -i \end{pmatrix}\begin{pmatrix} i & 0 \\ 0 & -i \end{pmatrix} = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}, \end{aligned}$$

which is equivalent to the quaternion $-1+0i+0j+0k = -1$. This is the same relation we observe with the complex numbers when *defined* by $i = \sqrt{-1}$. In fact, the complex numbers are a subring of the ring of quaternions, so that these i's are one and the same. Similarly, we can use the fact that

$$j = 0 + 0i + j + 0k = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix},$$

and

$$k = 0 + 0i + 0j + k = \begin{pmatrix} 0 & i \\ i & 0 \end{pmatrix},$$

to show that $j^2 = k^2 = -1$. Additionally, we find that

$$\begin{aligned} ij &= (0 + i + 0j + 0k)(0 + 0i + j + 0k) \\ &= \begin{pmatrix} i & 0 \\ 0 & -i \end{pmatrix}\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & i \\ i & 0 \end{pmatrix}, \end{aligned}$$

which is $0 + 0i + 0j + k = k$, but that

$$\begin{aligned} ji &= (0 + 0i + j + 0k)(0 + i + 0j + 0k) \\ &= \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} i & 0 \\ 0 & -i \end{pmatrix} = \begin{pmatrix} 0 & -i \\ -i & 0 \end{pmatrix}, \end{aligned}$$

which is $0 + 0i + 0j - k = -k$. Similarly, we can show that $jk = i$, $kj = -i$, $ki = j$, and $ik = -j$. This demonstrates the characteristic of the quaternions that we will be exploiting: in general, multiplication is non-commutative. Before we show that the non-zero quaternions form a non-commutative group under multiplication, we introduce two more terms:

We define the *conjugate* of $\alpha = a+bi+cj+dk$ to be $\overline{\alpha} = a-bi-cj-dk$. Then we define the *norm* of α to be $N(\alpha) = \alpha\overline{\alpha} = (a+bi+cj+dk)(a-bi-cj-dk) = a^2+b^2+c^2+d^2$. Now we would like to show that this has the usual norm property: $N(\alpha\beta) = N(\alpha)N(\beta)$. To do this, we first notice that the determinant of $\alpha = \begin{pmatrix} a+bi & c+di \\ -c+di & a-bi \end{pmatrix}$ is

$$\begin{aligned} (a+bi)(a-bi) - (-c+di)(c+di) &= (a+bi)(a-bi) + (c+di)(c-di) \\ &= (a^2+b^2) + (c^2+d^2) \\ &= a^2+b^2+c^2+d^2. \end{aligned}$$

Then we recall that for square matrices A and B, $det(AB) = det(A)det(B)$, so that $N(\alpha\beta) = det(\alpha\beta) = det(\alpha)det(\beta) = N(\alpha)N(\beta)$.

Theorem 1: The non-zero quaternions form a non-commutative group under multiplication.

Proof: The first property we must demonstrate is closure: Let $\alpha = a+bi+cj+dk$, and $\beta = w+xi+yj+zk$. Then $\alpha\beta = (aw-bx-cy-dz)+(ax+bw+cz-dy)i+(ay-bz+cw+dx)j+(az+by-cx+dw)k$, which is also an element of the ring of quaternions. Then we need to show that every element has a multiplicative inverse: Let α be a non-zero element of the ring of quaternions. Then let $\beta = \overline{\alpha}/N(\alpha)$, so that $\alpha\beta = \alpha\overline{\alpha}/N(\alpha) = N(\alpha)/N(\alpha) = 1$. For an identity, we notice that $1 + 0i + 0j + 0k$ corresponds to the identity matrix, and hence is a multiplicative identity. Finally, quaternion multiplication inherits its associativity from that of matrix multiplication. $\square$

It is natural to ask under what conditions multiplication is commutative, or what the center of the group of non-zero quaternions looks like.

Theorem 2: The quaternion α is central if and only if α is a real number.

Proof: It will be helpful to show that all real numbers are in the center first. Let $r = r + 0i + 0j + 0k$ be a real number. Then

$$r = \begin{pmatrix} r & 0 \\ 0 & r \end{pmatrix} = r\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = rI,$$

where I is the identity matrix. We know that real numbers commute with matrices, and that for any matrix A, AI=IA, so that rI the commutes with A, which means that real numbers are contained in the center.

Now, we must show that if an element is in the center, it is a real number. Let $\alpha = a + bi + cj + dk$ be in the center. Then α must commute with every element of the quaternions. In particular, we know that

$$\alpha i = i\alpha,$$

$$\alpha j = j\alpha,$$

and

$$\alpha k = k\alpha.$$

From the first equation, we have $\alpha i = ai+bii+cji+dki = -b+ai-dj+ck$, and $i\alpha = ai+ibi+icj+idk = ai+bii+cij+dik = -b+ai-dj+ck$. We can perform this middle step since a, b, c and d are real numbers. Equating the two gives us $d = -d$ and $c = -c$, so that $c = d = 0$. Next, we stipulate that $\alpha j = j\alpha$. This gives us $\alpha j = aj + bij + cjj + dkj = -c - di + aj + bk$, and $j\alpha = ja + jbi + jcj + jdk = -c + di + aj - bk$. Equating these two gives us that $d = -d$ and $b = -b$, so that $b = d = 0$. Using the third restriction gives us that $b = c = 0$. From this, we know that α is of the form $a + 0i + 0j + 0k$, and hence is a real number. □

For our purposes, we will be interested in a homomorphic image of a subring of H — the quaternions with *rational* coefficients — called H_p, where p is a prime.

Definition: Let p be an odd prime. Then α is an element of H_p if and only if $\alpha = a\zeta + bi + cj + dk$, for some a, b, c, $d \in Z_p$, where ζ is defined as $(1+i+j+k)/2$, and Z_p is the ring of integers modulo p.

For convenience, we identify $a\zeta + bi + cj + dk$ with (a,b,c,d). Addition is then defined in the obvious way, so that $(a,b,c,d) + (a',b',c',d') = (a+a', b+b', c+c', d+d')$, with the components reduced modulo p. If we follow the rules for quaternion multiplication, denoted $\times$, it is a matter of simple computation to show that

$$\begin{aligned}(a,b,c,d) \times (a',b',c',d') = (&-aa' - 2bb' - 2cc' - 2dd' - a'(b+c+d) - a(b'+c'+d'),\\ &aa' + bb' + cc' + dd' + a'(b+c) + a(b'+d') + cd' - c'd,\\ &aa' + bb' + cc' + dd' + a'(c+d) + a(b'+c') + b'd - bd',\\ &aa' + bb' + cc' + dd' + a'(b+d) + a(c'+d') + bc' - b'c).\end{aligned}$$

Lemma: If $\alpha \in H_p$ then $\overline{\alpha} \in H_p$.

Proof: Let $\alpha \in H_p$. Then $\alpha = (a,b,c,d)$ for some a,b,c, and $d \in Z_p$. Then $\alpha = a\zeta + bi + cj + dk = a/2 + (a/2+b)i + (a/2+c)j + (a/2+d)k$, so $\overline{\alpha} = a/2 - (a/2+b)i - (a/2+c)j - (a/2+d)k = a\zeta + (-b-a)i + (-c-a)j + (-d-a)k = (a, -b-a, -c-a, -d-a) \in H_p$.$\square$.

The norm of an element in H_p is defined the same way it is for the quaternions: $N(\alpha) = \alpha\overline{\alpha}$. So, if $\alpha = (a,b,c,d)$, then

$$\begin{aligned}
N(\alpha) &= N((a,b,c,d)) \\
&= N(a(1+i+j+k)/2 + bi + cj + dk) \\
&= N(a/2 + (b+a/2)i + (c+a/2)j + (d+a/2)k) \\
&= (a/2)^2 + (b+a/2)^2 + (c+a/2)^2 + (d+a/2)^2 \\
&= a^2/4 + b^2 + ab + a^2/4 + c^2 + ac + a^2/4 + d^2 + ad + a^2/4 \\
&= a^2 + b^2 + c^2 + d^2 + a(b+c+d).
\end{aligned}$$

Theorem 3: In H_p, α is a zero-divisor if and only if $N(\alpha) = 0$.

Proof: Let α be a zero-divisor, but with $N(\alpha) \neq 0$. Then there exists a $\beta \neq 0$ such that $\alpha\beta = 0$. We know that there exists a $k > 0$ such that $N(\alpha)^k = 1$ in Z_p. This means that $(\alpha\overline{\alpha})^k = 1$ in Z_p, so that $\overline{\alpha}^k\alpha^k = 1$ in H_p. Then $\beta = 1\beta = \overline{\alpha}^k\alpha^k\beta = \overline{\alpha}^k\alpha^{k-1}(\alpha\beta) = \overline{\alpha}^k\alpha^{k-1}0 = 0$, a contradiction.

Let $N(\alpha) = 0$. Then $\alpha\overline{\alpha} = N(\alpha) = 0$, so α is a zero-divisor. $\square$

IV. Some Traditional Generators

Over the years, many different types of PRNGs have been suggested and analyzed for their cryptographic properties. One of the more interesting generators is the linear feedback shift register, or LFSR. Some researchers, Bruce Schneier for one, do not recommend their use. On the other hand, he reports that LFSRs are currently used in the US military's field ciphers [8].

An LFSR consists of n cells which each hold one bit. We will denote their contents by $b_0, b_1, \ldots, b_{n-1}$. The next-state function is determined by n keyed constants $c_0, c_1, \ldots, c_{n-1}$. In order to go from one state to the next, we set the new b_{n-1} equal to $\sum_{j=0}^{n-1} b_j c_j \bmod 2$, and the new b_j equal to the old b_{j+1} for $j = 0, 1, \ldots, n-2$. The output is then b_0.

It is clear that this generator can be stuck in a very short loop. If each of the b_j's is zero, then the state will not be changed by application of the next-state function. On the other hand, if we initialize the LFSR to some non-zero state, and if we choose our c_j's so that $x^n + \sum_{j=0}^{n-1} c_j x^j$ is a primitive, irreducible polynomial in x, then the LFSR will loop through all $2^n - 1$ nonzero states before returning to the original state. In addition, the output sequence has some desirable properties:

- Of the $2^n - 1$ outputs, we will have almost an even number of zeros and ones (2^{n-1} versus $2^{n-1} - 1$)

- Every string of outputs n bits long will occur exactly once, with the exception of the all-zero string.

Countless ways have been devised to combine multiple LFSRs to produce more complicated outputs. As a consequence, we talk about the *linear complexity* of a system of LFSRs. The linear complexity is the length of the shortest single LFSR necessary to replicate the output of the system. It turns out that if n is the linear complexity of our generator, then with $2n$ consecutive output bits, we can break the generator. The method is based on the fact that the first n bits to come out form the initial state of the generator, which give us σ_0. We then set up n linear equations for the feedback coefficients $c_0, c_1, \ldots, c_{n-1}$, which define f_s. This set of equations can easily be solved by Gaussian elimination or by the Berlekamp-Massey algorithm [6]. Berlekamp-Massey takes advantage of the special form of the equations to more efficiently solve for the coefficients. Gaussian elimination requires $O(n^3/3)$ operations compared to $O(n^2)$ for Berlekamp-Massey. In either case, the $2n$ bits needed is very short compared to the period of $2^n - 1$ of the generator.

The $1/p$ generator is another PRNG that has very good output sequences, but which also is broken without much difficulty. If we choose a prime p, we can produce a sequence of b-ary digits by expanding $1/p$ in some base b. This sequence will have the property that every sequence of fewer than $\log_b p$ digits will occur in every period, and a sequence of $\log_b p$ will occur at most once. In their paper [2], Blum, Blum, and Shub describe a method to efficiently recover p given only $\log_b(2p^2)$ consecutive output digits.

Despite the ease with which the previous generators can be broken, there do exist random number generators that are very difficult to break. These generally have one of the famous "hard" cryptography problems at their core. We can design generators whose difficulty in breaking is based on factoring, discrete logs, or quadratic residuosity. The "power generator" has its next-state function given by $x_{i+1} = x_i^d \bmod N$. If we take $d = 2$ and $N = pq$, for p and q primes both congruent to 3 mod 4, then we have the BBS generator, which is considered one of the most secure PRNGs. In their paper [2], Blum, Blum, and Shub argue that an ability to predict the parity of next output given the previous output, with success any better than flipping a fair coin, can be used to guess quadratic residuosity. A similar CSPRNG has the form

$$x_{i+1} = g^{x_i} \bmod N,$$

where g is a primitive element of Z_N. This generator draws its strength from the difficulty of taking discrete logarithms. These last two generators have something else in common, in addition to their security. They both require exponentiation, and are therefore rather slow to implement.

Steven Wolfram devised a family of generators based on Cellular Automata [8]. In some sense, they are similar to LFSRs, since they too are composed of a number of cells which each hold a bit. However, the next-state function is very different. The n registers are updated in parallel, according to a constant rule, which can be key-dependent. One of the rules suggested by Wolfram updates each *cell* by

$$a_k^{i+1} = a_{k-1}^i \; XOR \; (a_k^i \; OR \; a_{k+1}^i),$$

where the subscripts are taken modulo n, and the superscripts emphasize that the cells are updated in parallel. The output is taken to be the sequential states of one of the cells. There is a paper [1] which purports to show that the outputs of cellular automata are equivalent to those of LFSRs. This is surprising, given that one may make the next-state equations as non-linear as one likes. For some reason, the paper seems to consider only certain rules, all of which are linear.

V. Some Non-Commutative Generators

The motivation behind looking at non-commutative versions of established PRNGs is that one hopes to be able to use the existing theory to analyze the properties of a generator, while at the same time making the generators more resistant to attack. For example, an obvious adaptation of the linear congruence generator has the form

$$x_{i+1} = ax_ib + c,$$

where $a, b, c \in H_p$. If we then try to break the generator in the same manner as we did earlier, we can set up the equation $a(x_1 - x_0)b = (x_2 - x_1)$. The beauty is that we are prevented from solving for either a or b in closed form, as we were able to do so simply in the commutative case. This does not mean that it is impossible to solve for a and b — we will discuss a simple technique later in the paper — but it is encouraging that we have transformed the most trivially-broken of all PRNGs into something far from trivial.

As we said above, in addition to requiring a CSPRNG to be difficult to predict, we also require that the output have "good statistics". Since our sequences are pseudorandom, there exist statistical tests that will be able to distinguish between any of the sequences we produce and sequences coming from uniform random distributions. Instead of trying to design statistical tests to defeat the generators we created, we set down two benchmark tests that should give us some indication for how good the generators are. We chose $p = 7$, so that all of the generators we tested produced output in the form of a string of elements of Z_7. The first test looks at the distribution of symbols, and the second test examines the spacing between consecutive appearances of each symbol. For example, one of the things we count is the number of times consecutive fives are separated by three non-fives. The results of these two tests are compared to what we would expect if the numbers were drawn randomly from a uniform distribution. So, for the first test, we compare the distribution produced by the generator to a uniform distribution, and for the second test we compare the distribution of spacings to a geometric distribution.

One of the immediate problems of trying to create non-commutative generators from the PRNGs discussed above is that the theory that has been developed is not always readily adaptable to the non-commutative case. For instance, we know that for an LFSR to have full period, the feedback polynomial must be irreducible and primitive. It is understood how to satisfy this when the polynomial is in $Z_2[x]$, but what happens when we take the coefficients from H_p?

There is also a dearth of information regarding methods of producing general solutions to even low order equations. This would suggest promise for generators like

$$x_{i+1} = ax_i bx_i c + dx_i e + f,$$

whose commutative counterparts are cryptographically weak. Similarly, we can foil the Berlekamp-Massey algorithm by mixing left and right multipliers in our non-commutative LFSR.

VI. What Are We Able to Say About These Generators?

Multiplying elements in H_p is a computationally more expensive operation than multiplying integers. Since speed of implementation is certainly one criterion by which to judge CSPRNGs, we limited ourselves to using as few multiplication operations as we could. For this reason, the non-commutative generators we concentrated on were:

- a linear congruence generator
- a two-term LFSR generator
- a five-cell cellular automata generator.

Of these three, the only one to show "good statistics" on a consistent basis was the cellular automata generator.

The non-commutative Linear Congruence Generator has the next-state function given by $x_{i+1} = ax_i b + c$, where neither a nor b is a zero-divisor. For the first two generators, we "tapped" the sequence using $f_o(a, b, c, d) = c$ as the output of the PRNG output. That is, the output digit is the coefficient of j. The output of a typical run of seven hundred iterations can be summarized as:

output digit	number of occurences
0	116
1	116
2	116
3	58
4	117
5	59
6	118

Worse than this is the spacing between outputs. For each output digit (across the top), the column below it shows the distribution of spacing of sequential occurences of that digit. So the first number under 2, for example, shows how often a 2 appeared immediately following the appearance of another 2.

	0	1	2	3	4	5	6
0	58	58	0	0	0	0	0
1	0	0	58	0	0	0	59
2	0	0	0	0	0	0	1
3	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0
5	0	0	0	0	116	1	0
6	0	0	0	0	0	0	0
7	0	0	0	0	1	0	0
8	0	1	0	0	0	0	0
9	0	0	57	0	0	0	58
10	58	57	0	0	0	0	0
11	0	0	0	57	0	58	0
12	0	0	0	0	0	0	0
> 12	0	0	1	1	0	0	0

If we look down column four, the LCG never produced two fours in a row, nor were there ever two fours separated by one non-four. In fact, with one exception, every time the LCG output a four, the next four appeared exactly six iterations later. It is clear that this in no way resembles the geometric distribution to which we are comparing it.

For the non-commutative LFSR, we used the next-state equation

$$x_{i+1} = ax_i + x_i b,$$

where again neither a nor b is a zero-divisor. The results for this generator were not good either. A typical output of the symbol frequencies for seven hundred iterations is

output digit	number of occurences
0	300
1	66
2	67
3	68
4	65
5	67
6	67

and for the spacing is:

	0	1	2	3	4	5	6
0	66	0	0	17	16	0	0
1	68	0	17	0	0	17	0
2	166	17	0	17	16	0	17
3	0	0	0	0	0	0	0
4	0	16	0	0	0	0	17
5	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0
7	0	0	0	0	1	0	0
8	0	0	0	0	0	0	0
9	0	0	16	17	16	17	0
10	0	0	0	0	0	0	0
11	0	0	0	57	0	0	0
12	0	0	0	0	0	0	0
> 12	0	33	34	16	16	33	33

For our non-commutative Cellular Automata, we used five cells, and the next-state function was given by:

$$a_k^{i+1} = a_{k-1}^i \times a_{k+1}^i.$$

Because this next-state function involves only multiplication, it is important that the cells not contain zero-divisors. In every instance we looked at where one of the cells initially contained a zero-divisor, the generator quickly degenerated to the fixed point of $a_1 = a_2 = a_3 = a_4 = a_5 = (0, 0, 0, 0)$.

Because of this, we need more information about the zero-divisors. Specifically, we have two questions: how many zero-divisors are there, and how are the different elements of Z_p distributed among the components of the zero-divisors? Knowing the answer to the first question would allow us to determine the size of the phase-space, while knowing the answer to the second one would let us know more about the population from which these numbers are coming. We know there are p^4 elements in H_p and there are just as many of the form $(0, b, c, d)$ as there are $(1, b, c, d)$. This is why, for the LFSR and the LCG, we compare the sequence of output digits to a *uniform* random distribution. But for the CA, excluding the zero-divisors may change the "mother population". We can still use the same two statistical tests, but we have to be careful — the distribution of occurences may no longer be uniform, which would also affect the gap distribution.

To answer the first question, we recall that the zero-divisors have norm congruent to zero. So to find the number of zero-divisors, we must count the number of solutions to $N(a, b, c, d) = a^2+b^2+c^2+d^2+a(b+c+d) \equiv 0 \bmod p$, with a, b, c, $d \in Z_p$. We conjecture that there are p^3+p^2-p solutions to this equation. Through an exhaustive count, this has been verified for $p \leq 67$. In addition, it has shown to be true for the $p \equiv 1 \bmod 4$ case [3]. It is worth noting that we can show that there are as many solutions to this equation as there are solutions to $w^2 + x^2 + y^2 + z^2 \equiv 0 \bmod p$, with w, x, y, $z \in Z_p$, which is likely easier to prove.

Theorem 4: The set of solutions $(a, b, c, d) \in H_p$ for the equation $a^2 + b^2 + c^2 + d^2 + a(b + c + d) \equiv 0 \bmod p$ has the same cardinality as the set of solutions for $w^2 + x^2 + y^2 + z^2 \equiv 0$.

Proof: Let w, x, y, $z \in Z_p$ be a solution to $w^2 + x^2 + y^2 + z^2 \equiv 0 \bmod p$. Then we set $a = 2w$, $b = x - w$, $c = y - w$, $d = z - w$, so that $a^2 + b^2 + c^2 + d^2 + a(b + c + d) = 4w^2 + x^2 - 2wx + w^2 + y^2 - 2wy + w^2 + z^2 - 2wz + w^2 + 2w(x - w + y - w + z - w) = 7w^2 + x^2 + y^2 + z^2 - 2w(x + y + z) + 2w(x + y + z - 3w) = w^2 + x^2 + y^2 + z^2 = 0$.

Since this mapping is one-to-one and onto, the sets have the same number of elements. □

We have verified that our conjecture is correct for $p = 7$, so we know that each cell can take on any of $p^4 - p^3 - p^2 + p$ different states. The five cell CA with $p = 7$ will have about 3.3×10^{13} possible states. For larger values of p the correction due to eliminating the zero-divisors becomes less and less significant, so that the total number of five-cell states approaches p^{20}.

As for the second question, we note that if (a, b, c, d) is a solution to $N(a, b, c, d) \equiv 0$, then $N(ga, gb, gc, gd) \equiv (ga)^2 + (gb)^2 + (gc)^2 + (gd)^2 + ga(gb + gc + gd) \equiv g^2(a^2 + b^2 + c^2 + d^2 + a(b + c + d)) \equiv g^2(0) \equiv 0 \bmod p$ is also a solution, for any $g \in Z_p$. In particular, if we have a zero-divisor (a, b, c, d), with, say $b \not\equiv 0$, then b will generate the additive group of Z_p, so that if we let g take on the values $1, 2, \ldots, p - 1$, we will have $p - 1$ zero-divisors, and the digits $1, 2 \ldots, p - 1$ will each appear once in the second position. Because of this, the non-zero elements of Z_p will be uniformly distributed as coefficients in the group of units.

Since this argument does not extend to the appearance of zeros as coefficients of zero-divisors, we choose to ignore them whenever they are the output digit of the CA generator, so that we can compare the remaining digits to a random uniform distribution. This has the added advantage that we are now considering an even number of output symbols, so that there is an obvious way to map to Z_2, which was lacking when the output symbols were all p elements of Z_p.

With this out of the way, we can analyze the output of a CA generator. For a run of 1400 iterations, with the output digit taken as the j coefficient, there were 1172 non-zero outputs (zero was the output digit 228 times):

output digit	number of occurences	expected number of occurences
1	198	195.33
2	193	195.33
3	208	195.33
4	186	195.33
5	204	195.33
6	183	195.33

The chi-squared sum for this is 2.49488, which corresponds to a *p*-value of approximately .22.

For the spacing:

	1	2	3	4	5	6	
0	33	27	34	28	41	28	30.500
1	28	34	31	22	26	29	25.417
2	26	19	24	23	32	17	21.181
3	16	18	21	28	24	24	17.650
4	13	15	17	16	8	12	14.709
5	16	15	14	12	10	10	12.257
6	17	11	17	10	11	11	10.214
7	8	7	9	6	10	7	8.512
8	10	7	9	5	12	5	7.093
9	5	9	1	6	5	7	5.911
10	1	6	4	4	2	3	4.926
11	0	6	6	2	3	6	4.105
12	4	3	5	3	2	2	3.421
13	0	2	5	1	0	3	2.851
> 14	21	14	11	20	18	19	14.253

The last column represents the expected values for the spacing of the six column. That is, if these numbers were truly random, we would expect to see adjacent sixes 30.5 times, while we observed them 28 times.

These produce the following chi-squared values:
Chi-squared for 1 spacing is 12.31882
Chi-squared for 2 spacing is 15.00070

Chi-squared for 3 spacing is 14.71301
Chi-squared for 4 spacing is 4.06404
Chi-squared for 5 spacing is 11.16476
Chi-squared for 6 spacing is 14.76778

The p-values for these range between .005 and .6.

As we stated earlier, having good statistics is only one requirement of a CSPRNG. A CSPRNG must also have cycles with long periods. For the generators we examined, we could not make any *a priori* statements about the lengths of the arcs. Because the LCG and the LFSR performs so poorly on the statistical tests, we did not even investigate their arc lengths or cycle periods. However, we used Ritter's technique a number of times for the CA generator, and were surprised by the results. The cycles we observed all seemed to have period 144. As for the arc lengths, there were instances where the generator completed five thousand iterations without entering a cycle.

VII. Breaking a Non-Commutative Generator

As we stated earlier, the non-commutivity of the reconstruction equations is a hindrance to their solutions, but this difficulty is not insurmountable. To illustrate one technique to break a generator, we will look at the non-commutative LFSR. Recall that this generator is defined by

$$x_{i+1} = ax_i + x_ib.$$

Let us suppose that we have access to x_0, x_1, and x_2, which are equal to $(x_{0,1}, x_{0,2}, x_{0,3}, x_{0,4})$, $(x_{1,1}, x_{1,2}, x_{1,3}, x_{1,4})$, and $(x_{2,1}, x_{2,2}, x_{2,3}, x_{2,4})$, respectively. Then, if we write a as (a_1, a_2, a_3, a_4) and b as (b_1, b_2, b_3, b_4), then we can set up eight linear equations in the eight unknowns b_i and a_i. The first equation would set the ζ coefficient of x_1 equal to the ζ coefficient of ax_0+x_0b, and would look like:

$$\begin{aligned} x_{1,1} = & - a_1x_{0,1} - (a_2 + a_3 + a_4)x_{0,1} - b_1x_{0,1} \\ & - (b_2 + b_3 + b_4)x_{0,1} - 2a_2x_{0,2} - 2b_2x_{0,2} - 2a_3x_{0,3} \end{aligned}$$

$$- \quad 2b_3x_{0,3} - 2a_4x_{0,4} - 2b_4x_{0,4}$$
$$- \quad a_1(x_{0,2} + x_{0,3} + x_{0,4}) - b_1(x_{0,2} + x_{0,3} + x_{0,4}).$$

The remaining seven equations would give us enough to be able to solve for a and b. Knowledge of these gives us the next-state equation, which would allow us to reconstruct the output in its entirety.

While an adaptation of this method of attack can be employed against any of the generators we considered, it loses its efficacy when used against the Cellular Automata generator. The reason for this is that the CA generator is *non-linear*, which makes the equations far more difficult to solve simultaneously. In fact, given the values of the five cells of a state, the computer algebra package *Mathematica* was unable to solve for the previous state, even after twelve hours of running! This would suggest promise for the CA generator as a one-way function, in addition to its qualities as a CSPRNG.

VIII. Conclusion

While we examined only a sampling of generators, it is clear that the non-commutative versions are not always superior to their commutative cousins. There is one notable exception, the CA generator. It produced good statistics, and breaking it seems to be difficult even with $p = 7$ and five cells. A larger value of p and more cells could only make it more secure. Also, while these generators may be somewhat slow in software, there is hardware support for 4×4 matrix multiplication, which could be used to multiply elements of H_p.

Acknowledgements
I would like to thank Dr. Greenfield for countless hours spent in discussion, of this subject and others, and for his sound advice throughout the entire process.

References

[1] Bardell, Paul, Analysis of cellular automata used as pseudorandom pattern generators, 1990 *International Test Conference Proceedings*, 762–

768.

[2] Blum, Blum, and Shub, A Simple Unpredictable Pseudo-Random Number Generator, *SIAM Journal of Computing*, Volume 15, Number 2, May 1986, 364–383.

[3] Greenfield, Gary, personal communication, March 1996.

[4] Hillman and Alexanderson, *A First Undergraduate Course in Abstract Algebra, 3^{rd} Edition*, Wadsworth Publishing Company, Belmont California, 1983.

[5] Knuth, Donald, *The Art of Computer Programming*, Addison-Wesley, Reading Mass, 1973.

[6] Lidl and Niederreiter, *Finite Fields*, Addison-Wesley Publishing Company, Reading, Mass, 1983.

[7] Ritter, Terry, Efficient Generation of Cryptographic Confusion Sequences, *Cryptologia*, Volume 15, Number 2, pages .

[8] Schneier, Bruce, *Applied Cryptography*, John Wiley and Sons, Inc., New York, 1994.