Chaos, Cryptology,

and the Coupled Map Lattice

A Senior Research Project in Mathematics

Matt Weeks

April 19, 2010

Abstract

The analogies between chaos and cryptology have led to many proposals of chaotic systems as bases for cryptosystems. The spatiotemporal logistic map lattice, the most commonly studied chaotic coupled map lattice, seems to provide a well-suited basis for a cryptosystem as a pseudorandom bit sequence generator. However, this system is not chaotic under many conditions as previously thought, and in many cases, the lattice elements are closely related to each other. Furthermore, the generator directly derived from this lattice is inherently insecure for cryptographic application as its underlying structure is not out of the reach of the cryptanalyst. This is true over different lattice sizes, coupling ranges, and other parameter variations.

1 Introduction

The study of dynamical system has been dramatically altered in the past 50 years by the appearance of chaos theory. This study demonstrated that many, or even most, models of dynamical systems lead to an unpredictable state known as chaos. What frustrated weather forecasts and predictions of all kinds of systems nevertheless proved promising to another class of problems, that of generating random numbers and cryptography. A wide number of proposals have been created, many of which involve the couplied map lattice, whose behavior will be investigated further.

2 Chaos

2.1 Definition

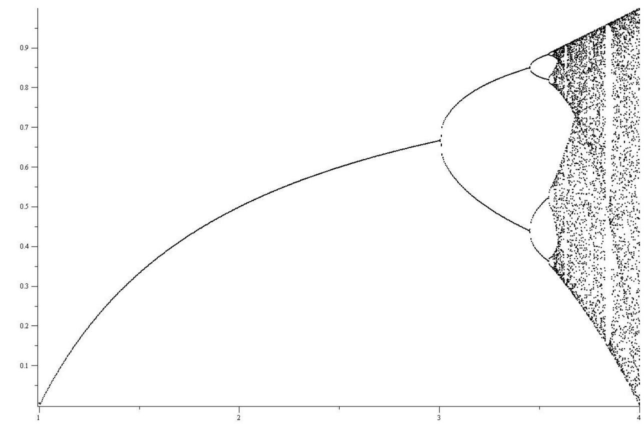
Chaos in popular language refers to any situation which is unpredictable and disorderly or according to the Merriam-Webster dictionary "A state of utter confusion." [1] In mathematical applications, chaos in a dynamical system refers to a system that has sensitivity to initial conditions and nonperiodicity, that is, it is not asymptotically periodic. This type of behavior results in a system that is unpredictable, as a typical orbit does not approach a single point or stable cycle, but remains a chaotic orbit.

2.2 The Logistic Map

The logistic map is one of the most basic systems exhibiting chaotic behavior. It is a simple one-dimensional model of population that predicts the population at time $t + 1$ solely dependent on the population at time t. Population ranges from 0 to 1, and the behavior is dependent on a parameter α that determines the system behavior. The map is defined by the following recurrence relation:

$$x_{t+1} = \alpha x_t \left(1 - x_t\right)$$

This relation was designed to represent how the population is dependent on both how many individuals exist and can reproduce, as well as how overpopulation can limit large populations. The function $f(x) = \alpha x_t \left(1 - x_t\right)$ is positive between 0 to 1. As α is varied, the behavior changes. With $0 \leq \alpha \leq 1$ the population converges to 0. When α is raised above that value, up to about 3, the system first converges to a fixed point. When α is from 3 to about 3.5, it approaches a cycle, and the period doubles from two to four to eight, etc. When α is above that point, the system begins to exhibit chaotic behavior, and at $\alpha = 4$ the system is chaotic across the entire interval $[0, 1]$. This behavior is represented by the bifurcation diagram, which represents possible long-term values of the system over different values of α.



Logistic map bifurcation diagram

The Lyapunov exponent is the standard measure of how neighboring values of the system are attracted to or diverge from an orbit. It can be thought of as the exponential rate of separation of nearby orbits. For a one-dimensional map defined by $x_{n+1} = f(x_n)$, the Lyapunov exponent is defined as

$$\lim_{n\to\infty} \frac{1}{n} \left(|f'(x_1)| + |f'(x_2)| + ... + |f'(x_n)|\right)$$

Chaotic systems contain orbits with Lyapunov exponents that are greater than 0. A Lyapunov exponent less than 0 indicates a stable cycle. The logistic map has a Lyapunov exponent of 2 at $\alpha = 4$. Willeboordse makes use of Lyapunov Spectra, an extention of the Lyapunov exponent for spatially extended systems $x_n = (x_n^1, ..., x_n^N)$ that hold state in a vector of values instead of one variable. [6] The Lyapunov spectra of a system are defined as

$$\lim_{n\to\infty} \frac{1}{n} \ln\left(i\text{th eigenvalue of } J_{n-1}J_{n-2}...J_0\right)$$

where J_n is defined by

$$(J_n)_{i,j} = \frac{\partial x^i_{n+1}}{\partial x^j_n}$$

3 Cryptology

3.1 Introduction

A brief introduction to cryptology would not be complete without a short discussion of the terms involved. Cryptography is "The science and art of making codes and ciphers" or codemaking. Cryptanalysis is "The conversion of encrypted messages into plain text without having the initial knowledge of the key used in encryption" or codebreaking. Cryptology incorporates both. [2] A cryptosystem is a system, usually a cipher, to encrypt and decrypt messages. Plaintext is the original text of a message, and ciphertext is the encrypted text of a message. A cryptosystem is considered broken if there exists an algorithm to decrypt the encrypted message faster than a brute-force search. A practical break must also fit within realistic time constraints on realistic hardware. Cryptanalytic attacks on cryptosystems generally fall within one of three defined categories. The most powerful and desirable for a crypanalyst is the ciphertext-only attack. These attacks result in decryption given only the ciphertext of a message. They are the most difficult to find, but always able to be applied since the cipertext is always available. The next most desirable is the known plaintext attack. Given only part of the message or messages encrypted with a key, a known plaintext attack will decrypt the remaining message or messages.

Most known-plaintext attacks are also easily applicable to modern encrypted communications. (see appendix B) Next is the chosen plaintext attack. In this attack, which is easier to find but usually not applicable to most real-life situations, the attacker chooses part of the message which will let him find out the rest of the message. Other categories include chosen ciphertext and related key attacks, which imply the ability to choose a ciphertext or some knowledge about the key.

3.2 Chaos and Cryptology

Chaos and cryptography hold many similarities that form connections between the two. Each property of chaos is analogous to a cryptographic principle. Sensitivity to initial conditions, a chaotic principle, is very important to cryptography. Even a key that is very close to the original key must create a ciphertext that bears no resemblance to the original so that the attacker cannot judge whether a guess was close. He must try every possible key to break the system. This absolute dependence on initial conditions is known as the avalanche effect. In the same manner, a chaotic system does not approach a periodic orbit, which would make it become predictable. Likewise, different keys must result in completely different ciphertexts, even over long messages. To summarize, both must be pseudorandom. There is a deterministic procedure that cannot easily be discovered by the outsider; it looks random. There are some differences; cryptosystems prefer integer formulae and results, which can be exactly represented numerically and easier to work with when encoding digital data.

Most chaotic systems, however, rely on a continuous state. Nevertheless, due to the similarities, chaotic systems have been proposed as the basis of numerous cryptosystems, usually as the basis of a one-time pad-inspired cryptosystem.

3.3 One-time Pad

The one-time pad cryptosystem (OTP) is one of the simplest cryptosystems and the only cryptosystem proven impossible to break. In the digital form of OTP, the plaintext is a string of n bits (each of which is either 0 or 1). The key is a string of n perfectly random bits as long as the plaintext. The ith bit of the ciphertext is formed by applying the XOR operation to the ith bit of the plaintext and the ith bit of the key. $C_i = P_i \oplus K_i$ The operation XOR is a function $XOR : B \times B \rightarrow B$ that is equal to 1 if the two input bits are different, or 0 if they are the same. It produces the same result as addition modulo 2. Therefore, the ciphertext can be decrypted by the same process $P_i = C_i \oplus K_i$ since $(x \oplus y) \oplus y = x$ for any bits x, y. This cryptosystem is perfectly secure in that for any binary string c of length n and any plaintext p of length n, there exists a key k such that the ciphertext produced by encrypting p with k is c. [5] Therefore, the ciphertext gives no information about the message content to an attacker aside from the message length. OTP is not frequently used because of the difficulties with generating truly random bit sequences and the difficulty of distributing long keys. Many cryptosystems are nevertheless based on OTP, but with the use of a pseudorandom bit sequence generator (PRBSG), a type of pseudorandom number generator (PRNG), to generate each K_i. In this case, the parameters used to "seed" the generator, the initial conditions, are regarded as the key.

4 The Coupled Map Lattice

4.1 Form

The coupled logistic map lattice (or coupled map lattice) is an extention of the logistic map that has been widely used in literature to represent spatiotemporal chaos. [4, 6] The single state value x at time t is replaced with a lattice of elements; x_t^1 through x_t^L where L determines how many spatial elements are in the lattice. The coupled map lattice uses the same function

$$f(x) = \alpha x(1-x)$$

from the logistic map, and the value of α remains fixed for a lattice.

The value of the lattice at time $t+1$ is defined by applying f to each element of the lattice at time t, and then mixing or coupling the value of the lattice element with the values of neighboring elements according to this relationship:

$$x_{t+1}^i = (1-\epsilon) f\left(x_t^i\right) + \frac{\epsilon}{2r} \sum_{k=1}^{r} \left(f\left(x_t^{i-k}\right) + f\left(x_t^{i+k}\right)\right)$$

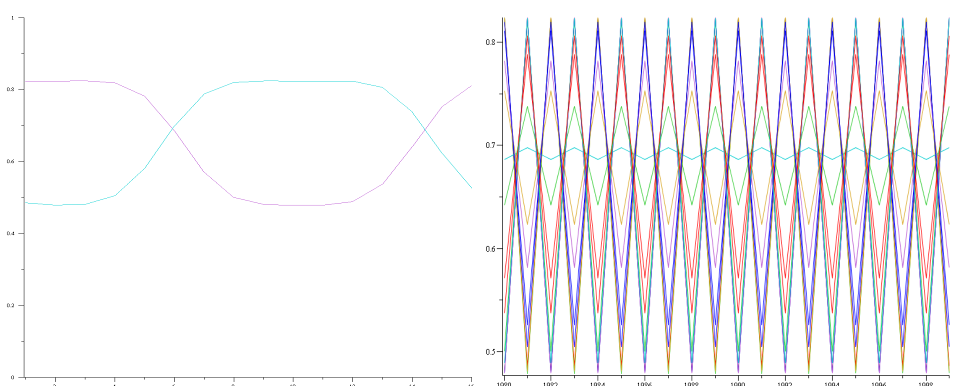
[6] The extent of the coupling is determined by the value of ϵ and the value of r. This research focused on variations of a proposed chaotic coupled map lattice in [4] with parameters $L = 16$, $\epsilon = 0.5$, $r = 1$, and $\alpha = 4$, so unless

otherwise noted, all lattices used in examples or graphs in this paper will use these parameters by default.

4.2 Behavior

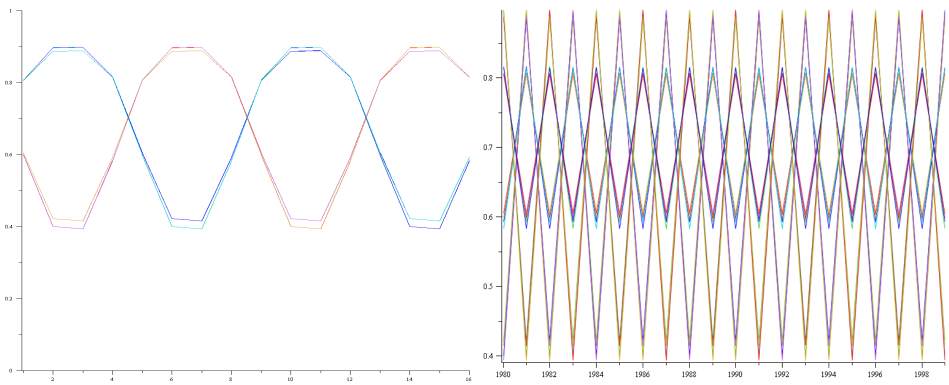
The coupled map lattice generates a sequence of patterns as the parameter α is varied up to 4. For the following graphs, the lattice was seeded with random values between 0 and 1 with Maple's random number generator, and iterated for 2000 iterations, displaying the last 30 overlaid in a spatial view (elements displayed horizontally) and temporal view (elements overlaid, with the horizontal axis time). At a low α, every element of the lattice converges to the same fixed point as the uncoupled logistic map.

When α is between about 3.1 and 3.8, the lattice displays a repeating sequence, an orbit of period two or four, etc. visible in the temporal direction characterized by oscillations in the spatial dimension.



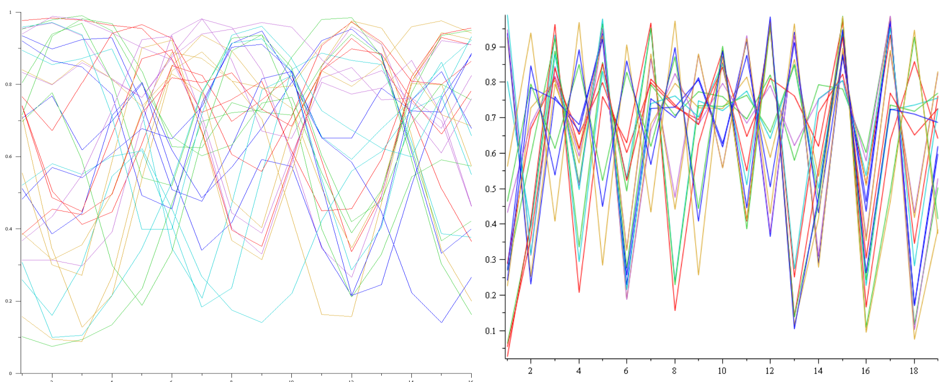
Spatial and temporal views of coupled map lattice with $\alpha = 3.3$

As α is increased, these increase in frequency.



Spatial and temporal views of coupled map lattice with $\alpha = 3.7$

At high values of α, spatiotemporal chaos sets in. [6]

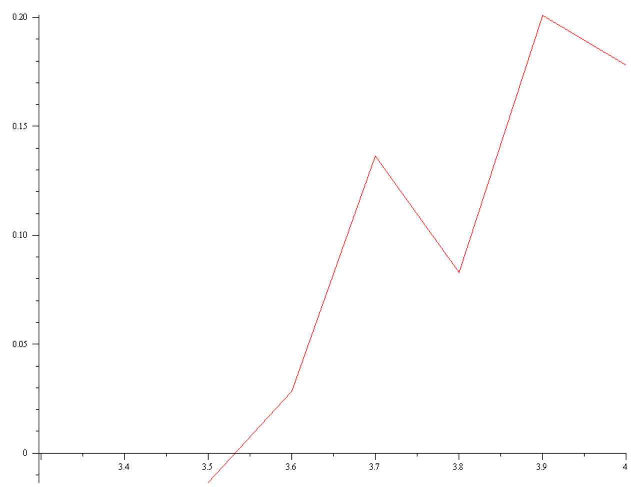


Spatial and temporal views of coupled map lattice with $\alpha = 4.0$

To create a chaotic system, α was fixed at 4 in the proposals in the papers I researched.

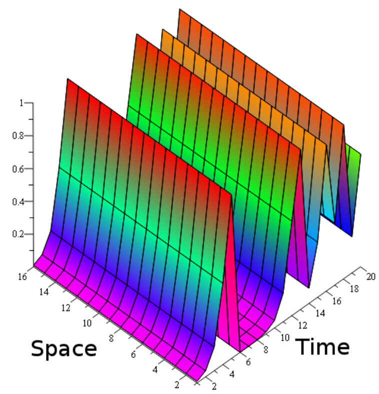
4.3 Properties

The Lyapunov spectra of the coupled map lattice (once again with $L = 16$, $\epsilon = 0.5$, and $r = 1$) are shown for values of α from 3.3 to 4; demonstrating the sensitivity to initial conditions of the system.



Average Lyapunov spectra by α

The Lyapunov spectra, however, do not always reflect the effects of coupling. For example, when the coupling parameter r is increased to 7, even with $\alpha = 4$, the lattice elements are drawn together, and the system becomes a lattice of identical simple logistic maps.

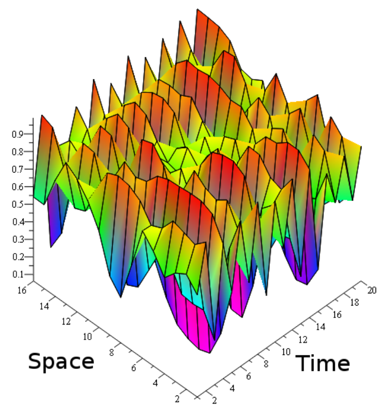


$r = 7$; average Lyapunov spectra = .5; $\rho = 1$

Linear correlation is another way of quantifying the relationship between elements of the coupled map lattice, to measure the spatial correlation. The sample linear correlation coefficient between $x_n(1)$ and $x_n(2)$ is here displayed as ρ.

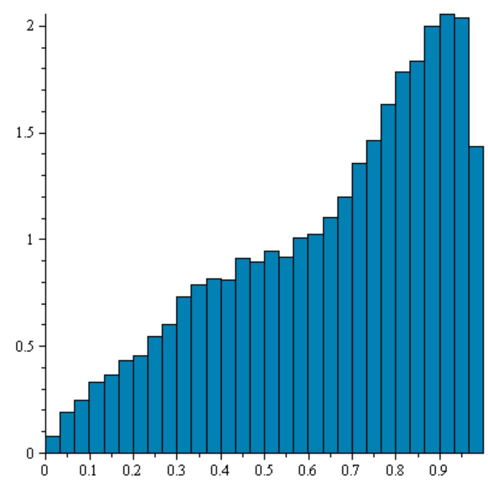
$$\rho = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{(n-1)s_x s_y}$$

With $r = 7$, $\rho = 1$, as there is very strong correlation between neighboring elements, and likewise between any elements. With $r = 1$, neighboring elements have a statistically significant correlation, (about 0.7) while elements significantly removed from each other have little correlation, (about 0.3) when simulated to 2000 time steps.



$r = 1$; average Lyapunov spectra $= .2$; $\rho = .7$

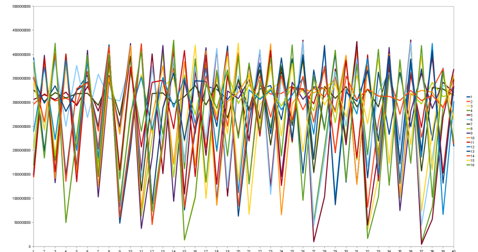
While chaotic, the $r = 1$ system does not produce a uniform distribution. The following distribution was generated from 2000 values of a lattice with $L = 16$ elements, $\epsilon = 0.5$, and $r = 1$.



Histogram of coupled map lattice element values

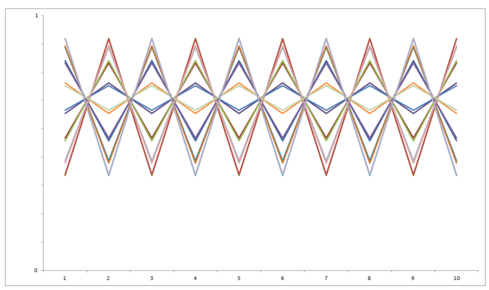
But longer simulations reveals that the system is not chaotic as originally thought.

Shorter term chaotic behavior results in Lyapunov spectra of 0.1-0.3.



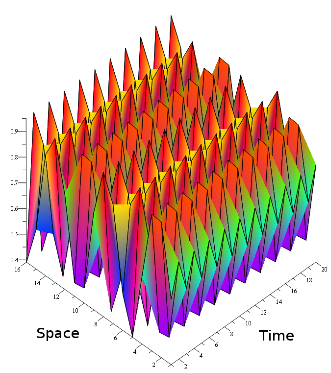
Chaotic behavior at $n = 2000$

But when the system is run for longer periods, the long-term behavior is periodic, and the calculated Lyapunov spectra approaches $-\infty$.



Periodic behavior at $n = 65530$

This periodic behavior for $L = 16$ consists of a period-2 cycle without a clear pattern spatially.



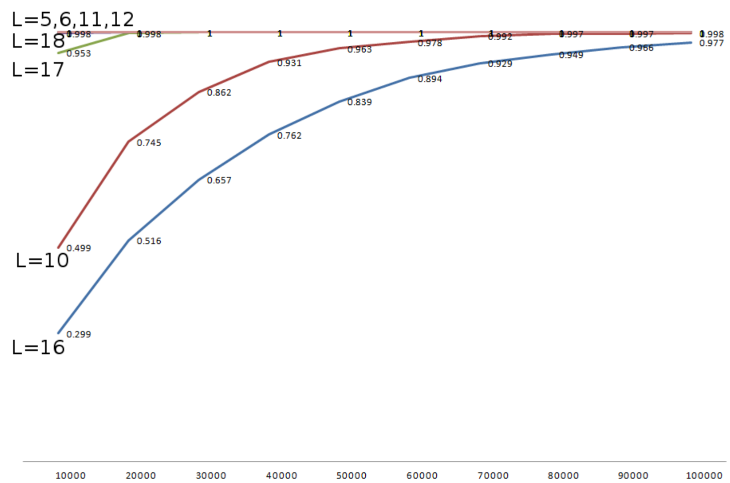
Period-2 behavior at $n = 10000$

The periodic behavior is also heavily dependent on L. Not all L display periodic behavior, and of the values of L that do display asymptotically periodic behavior, the cycle length is dependent on L.

L	1	2	3	4	5	6	7	8	9	10	11	12	13
cycle length	-	-	-	-	2	4	-	-	-	2	2	4	-
L	14	15	16	17	18	19	20	21	22	23	24	25	26
cycle length	-	-	2	2	4	4	-	-	2	2	4	4	-

"-" means not apparently periodic after 100000 time steps. These values of L may not converge, or may converge to cycles with period longer than 40 or may only converge after 100000 time steps.

With the values of L that did produce periodic behavior, some initial conditions converged faster than others. The following graph shows the probability of convergence to periodic behavior by computing, for each value of L that apparently produced periodic behavior, 100000 time steps with 1000 different random initial conditions from the Sun Java cryptographically secure random number generator.



Probability of periodic behavior - threshold $2^{-6} = 0.015625$.

5 Cryptography

5.1 Nanjing Cryptosystem

The coupled map lattice has been proposed as the basis of a practical cryptosystem by Mao, et al. [4] at the Nanjing University of Science and Technology. To distinguish this cryptosystem from another, it will be referred to as the Nanjing cryptosystem. In their paper, the map lattice is used as the basis of a pseudo-random bit sequence generator, which then forms the basis of their one-time pad inspired cryptosystem. To find use as a practical system, it was implemented on a circuit. The lattice proposed used $L = 16$ elements, with coupling parameters $\epsilon = 0.5$ and $r = 1$, and $\alpha = 4$ (our default values). Each lattice element was represented as a 32-bit integer. The random bits generated by the lattice at time t were the least significant 16 bits of each lattice element, 256 in total.

5.2 Cryptanalysis

The proposed pseudorandom bit sequence generator, although more than sufficient for the casual random bit sequence generator, is not secure against a determined adversary. There is no problem with the period, or the distribution of the generated bits, and the generator has an internal state of 512 bits, which is more than secure against a naive brute-force attempt. However, with a known plaintext attack, the lower 16 bits of each lattice element may be directly computed by reversing the one-time pad. This results in 256 bits becoming known leaving 256 bits unknown of the generator's internal state, which is still secure against a naive brute force. (even the AES algorithm, the only public block cipher approved by the National Security Agency for top-secret data encryption only has keys up to 256 bits in length) The primary problem lies with the coupling, which is weak by cryptographic standards. A change of one element of the lattice x_n^i only affects x_{n+1}^{i-1}, x_{n+1}^i, and x_{n+1}^{i+1} since the coupling parameter r is set at 1. Therefore, x_{n+1}^i is completely determined by x_n^{i-1}, x_n^i, and x_n^{i+1}. This allows the cryptanalyst to break the encryption apart piece by piece, and avoid dealing with all of the 512-bit state at once. The cipher was broken by taking advantage of this weakness. It relied on knowledge of 96 consecutive bytes of plaintext for the break demonstrated, or only 64 bytes for a slightly slower break that might end up with multiple possibilities for the key. The full break took about 8 hours on a network of standard workstations.

A more complete description of the break can be found in Appendix A, and the full source code can be provided upon request. From a mathematical perspective, we can see that although the lattice element $x_{t+1}(i)$ is sensitive to the initial conditions of $x_t(i)$, it is not immediately sensitive to the initial conditions of $x_t(i+2)$ or most other lattice elements. Only at later time steps does one lattice element affect all the others.

5.3 Chaotic Alternatives

To resist an attack against the system by a piecewise brute force algorithm, the coupling parameter r can be increased. To avoid a piecewise break, the parameter would have to be increased to $L/2$ so all lattice elements at time t would be included in the calculation of each element's value at time $t+1$. However, increasing r neutralizes the chaotic system. With a large r, the lattice elements converge to the same value, as we have seen.

An attempt to strengthen the system could be made by increasing L, the number of elements in the lattice. However, once the first five elements have been found, each successive element is broken in constant time by trying each possibility for the unknown upper 16 bits, eliminating those that do not generate the correct value for the next two time steps. Even without using parallel optimizations, this takes a fraction of a second per element. The overall time to break a 64 element lattice would be only a few seconds longer than the 16 element lattice, even though it uses a huge 2048 bit internal state. The cryptosystem could alternately be strengthened by only extracting bits from the lattice at times t, $t+8$, $t+16$ ... so that all lattice elements at time t would be included in the calculation of the next block of random bits without losing the chaotic properties of the system. However, this would make the generation process take eight times as long, and prohibit its use encrypting a high-speed data stream. The system would still be weaker than anticipated due to the uneven distribution of element values, and the correlation between neighboring elements.

6 Tianjin Cryptosystem

The coupled map lattice was also proposed as the basis of a cryptosystem by Hui, et al. from Nankai University in Tianjin, China, with more of an eye toward security. This system also created a pseudorandom bit sequence generator that formed the keystream of an OTP based encryption. In this system, a coupled map lattice of $L = 64$ elements was seeded with a separate pseudorandom number generator for each block. The lattice was then iterated 116 time steps before extracting the most significant bit from each element. This produced a block of 64 random bits. The number of time steps was chosen so that an attacker would have to guess the initial conditions exactly in 63 elements and within closer than 0.01 in the last to have a noticeably better than even chance of guessing an output bit correctly. [3] Therefore, a brute force attack would require significantly more than 100^{64} tries as a first step, which is reliably secure against brute-force attacks. However, this system is impractical for actual use, since generating each 64 bit block requires a number of operations equal to the number of operations to generate the 64 seed values from the other random number generator, plus the number of operations to calculate the coupled map lattice state over 116 steps on a lattice of 64 elements, which is $64 \cdot 116 \cdot 20$ since about 20 operations are required to calculate f and the coupling. This results in over 128000 operations to encrypt each block of 8 bytes.

7 Conclusion

The coupled map lattice provides an example of a multivariate spatiotemporal chaotic system, but only with certain parameter values. In cryptography, the coupled map lattice is very difficult to use as the basis for a practical, yet secure cryptosystem. Furthermore, its use as a random number generator is curtailed by the presence of an uneven distribution, linear correlation between elements, and long-term periodic behavior, since, surprisingly, coupling with many lattice sizes results in a stable system.

References

[1] Merriam Webster Online Dictionary. Mar 2010.

[2] National Security Agency Terms and Acronyms. Mar 2010.

[3] MA Hui, ZHU Kai-En, and CHEN Tian-Lun. A Cryptographic Scheme Based on Spatiotemporal Chaos of Coupled Map Lattices. Communications in Theoretical Physics, 45(3):477–482, 2006.

[4] Yaobin Mao, Liu Cao, and Wenbo Liu. Design and FPGA Implementation of a Pseudo-Random Bit Sequence Generator Using Spatiotemoral Chaos. In Communications, Circuits and Systems Proceedings, 2006 International Conference on.

[5] C. E. Shannon. Communication theory of secrecy systems. Bell Systems Technical Journal, 28:656–715, 1949.

[6] F. H. Willeboordse. The Spatial Logistic Map as a Simple Prototype for Spatiotemporal Chaos. Chaos, 13, 2003.

Appendix A. Details of Break and Code

.1 Selected Code Listing

The following code contains examples from the algorithm implementation in C++. First the algorithm parameters were defined. These were hardcoded for each run to allow better compiler optimization. The code also defined f to mirror the FPGA implementation and used 32 and 64 bit integer arithmetic to calculate $f(x)$ accurately for each 32-bit x. [4]

```
//ALGORITHM CONSTANTS
const int l = 16;
const int v = 16;
const int m = 32;
const unsigned long long mod = 1LL<<m;
const unsigned int vmod = 1<<v;
const unsigned long upperMod = 1 << (m-v);

//See "Design and FPGA Implementation of a Pseudo-Random Bit Sequence Generator Using Spatiotemporal Chaos"
inline unsigned long f(unsigned long x){
  unsigned long minus = mod-x;
  unsigned long long product = minus*(unsigned long long)x;
  return (product >> (m-2))%mod;
}
```

The algorithm assumed knowledge of 96 consecutive bytes of the plaintext and the corresponding ciphertext. These allowed us to know the least significant

16 bits of each element of the lattice at x_1, x_2, and x_3. The first major step the algorithm performed was to find all possible values of the unknown upper 16 bits of x_1^1, x_1^2, and x_1^3, that is, those values that would produce the correct (known) lower 16 bits for x_2^2. To calculate this step we used the formula

$$x_{t+1}^i = (1-\epsilon) f\left(x_t^i\right) + \frac{\epsilon}{2}\left(f\left(x_t^{i-1}\right) + f\left(x_t^{i+1}\right)\right)$$

with $\epsilon = 1/2$. This step took the longest time of any step, but eliminated the vast majority of possibilities for x_1^{1-3}. (only 1 out of about each 2^{16} remained) In this process, the function created a set of the possibilities of x_1^2 and x_1^3, eliminating those that could not produce the correct value for x_2^2. (which, as we will see, reduced the search space for x_1^{2-4})

```
//main brute force function for triples. Returns the unique doubles in last two elements, and writes a file with the results
DoubleInfo breakThreeParallel(unsigned int** targets) {
  unsigned int x1=targets[0][0],
        x2=targets[0][1],
        x3=targets[0][2],
        target=targets[1][1];
  size_t total=0,numDoubles=0;
  int nthreads;

  //for return data
  DoubleInfo returnData;
  unsigned int bitmapSize=(1LL<<(2*(m-v)-3));//1 bit for each of 2^2(m-v) possibilities  2^(m-v) for first, and  2^(m-v) for second
  cout<< "allocating bitmap of size "<<bitmapSize<<endl;
  BYTE* bitmap = new BYTE[bitmapSize];
  memset(bitmap,0,bitmapSize);

  {
    size_t threadTotal=0;
    nthreads = NUM_PROCESSORS;
    int numThreadsLocal=nthreads;
    int thId = processorNumber;
    char name[20];
    sprintf_s(name,20,"breakTmp%d",thId);
    HANDLE ftemp = CreateFileA(name,GENERIC_WRITE,FILE_SHARE_READ,NULL,CREATE_ALWAYS,FILE_ATTRIBUTE_TEMPORARY,NULL);
    if(ftemp == INVALID_HANDLE_VALUE)
      die("Cannot make file in breakThreeParallel");
    Triple* outputTriples = new Triple[TRIPLE_BUFFER_SIZE];

    //blocked instead of interleaved (x=thId; x<upperMod; x+=numThreadsLocal) to keep sort order
    for(unsigned int x=thId*upperMod/numThreadsLocal; x<(thId+1)*upperMod/numThreadsLocal; x++){
      //get new value to try for x_n(valuesIterated) setting first m-v bits to the bits of x
      x1= (x1 % vmod) + (x<<v);
      //apply f to x_n(valuesIterated)
      unsigned int f1= f(x1);

      //X2
      for(unsigned int y=0; y<upperMod; y++){
        x2= (x2 % vmod) + (y<<v);
        unsigned int f2= f(x2);

        //X3
        for(unsigned int z=0; z<upperMod; z++){
          x3= (x3 % vmod) + (z<<v);
          unsigned int f3= f(x3);
          if((f2 / 2 + f1 / 4 + f3 / 4)//x_n+1
              % vmod == target % vmod){         // last v bits = seen
```

```
          unsigned int indx = threadTotal % TRIPLE_BUFFER_SIZE;
          outputTriples[indx].first=x1 >>(m-v);
          outputTriples[indx].second=x2 >>(m-v);
          outputTriples[indx].third=x3 >>(m-v);
          threadTotal++;
        {
          //doubles
          unsigned int index=((y<<(m-v))+z)/8; //find index of bitmap with the bits of first and x3 (except for the last 3)
          if((bitmap[index] & (1 << (z & 0x7)))==0){//test bit of bitmap[index] corresponding to least significant 3 bits of x3
            numDoubles++;
            bitmap[index] |= 1 << (z & 0x7);//set bit of bitmap[index] corresponding to least significant 3 bits of x3
          }
        }
          if(indx == TRIPLE_BUFFER_SIZE-1){ //Write buffered triples
            DWORD written;
            unsigned int toWrite=sizeof(Triple)*TRIPLE_BUFFER_SIZE;
            if(!WriteFile(ftemp,outputTriples,toWrite,&written,NULL) || written != toWrite)
              cout << "ERROR file write failed. " << written << " bytes written in thread " << processorNumber<<endl;
          }
        }
      }

    }//x2
  }//x1
  DWORD written;
  DWORD toWrite=sizeof(Triple)*(threadTotal % TRIPLE_BUFFER_SIZE);
  if(!WriteFile(ftemp,outputTriples,toWrite,&written,NULL) || written != toWrite)
    cout << "ERROR file write failed. " << name << "  " << written << " bytes written " << endl;
  CloseHandle(ftemp);
  total += threadTotal;
 }
 returnData.size=numDoubles;
 //return bitmap if it will use less memory than the list
 if(numDoubles*sizeof(DoubleShort)>bitmapSize){
   returnData.data=bitmap;
   returnData.type=BITMAP;
   return returnData;
 }

 //collect results in an array
 DoubleShort* compactedDoubles = new DoubleShort[numDoubles];
 size_t n=0;
 for(unsigned int i=0; i < 1<<(m-v); i++){
   for(unsigned int j=0; j < 1<<(m-v); j++){
     unsigned int index=((i<<(m-v))+j)/8; //find index of bitmap with the bits of first and second (except for the last 3)
     if(bitmap[index] & (1 << (j & 0x7))){//test bit of bitmap[index] corresponding to least significant 3 bits of second
       compactedDoubles[n].first=i;
       compactedDoubles[n].second=j;
       n++;
     }
   }
 }
 delete [] bitmap;
 returnData.data=compactedDoubles;
 returnData.type=DOUBLESHORT_ARRAY;
 return returnData;
}
```

The next function was used to find the possible values of x_1^i using only the possible values of x_1^{i-1} and x_1^{i-2}. For example, starting with each possibility of x_1^2 and x_1^3, the program tested each value of the upper bits of x_1^4, and saved the results. Since we had reduced the number of possibilities of x_1^2 and x_1^3, this step took less time than trying all possibilities of x_1^{1-3}.

As this process proceeded iteratively, the possibilities for the unknown parts of each x_1^i and each x_1^{i+1} were reduced, and finding the possibilities for the next elements proceeded more quickly.

```
size_t breakThird(unsigned int x1, unsigned int x2, unsigned int x3, unsigned int target,
      unsigned int f1,unsigned int f2, Triple* outputTriples, size_t base, HANDLE ftemp) {
  size_t total=0;
  bool alert=false;
  for(unsigned int x=0; x<upperMod; x++){
    //get new value to try for x_n(valuesIterated) setting first m-v bits to the bits of x
    x3= (x3 % vmod) + (x<<v);
    //apply f to x_n(valuesIterated)
    unsigned int f3= f(x3);
    //Try all possibilities for last element
    if((f2 / 2 + f1 / 4 + f3 / 4)//x_n+1
        % vmod == target % vmod){         // last v bits = seen
      unsigned int indx = (total+base)%TRIPLE_BUFFER_SIZE;
      outputTriples[indx].first=x1 >>(m-v);
      outputTriples[indx].second=x2 >>(m-v);
      outputTriples[indx].third=x3 >>(m-v);
      total++;
      if(indx == TRIPLE_BUFFER_SIZE-1){
        DWORD written;
        unsigned int toWrite=sizeof(Triple)*TRIPLE_BUFFER_SIZE;
        if(!WriteFile(ftemp,outputTriples,toWrite,&written,NULL) || written != toWrite)
          cout << "ERROR file write failed. " << written << " bytes written in thread " << processorNumber<<endl;
      }
    }
  }
  return total;
}
```

During testing, it was made clear that this process, if continued, could generally reduce the number of possibilities to only a handful. However, finding an exact solution was made faster using the knowledge of x_3 and looking at the possibilities in groups of five. The next function formed the core of the algorithm to find all possibilities of x_1^{1-5}. It eliminated possibilities for groups of five as it tested whether the tested lattice elements produced the correct answer for x_2^{2-4} and whether those calculated values produced the correct answer for x_3^3.

```
//Using four known (in baseQuint), and some calculated values, breaks fifth. Returns number found. Uses stored triple files.
//Check for duplicate f(x) or identical lsb f(x) values
size_t breakFiveWithFour(Quint baseQuint, int start3,unsigned int f2,unsigned int f3, unsigned int target, unsigned int targetx35,
      unsigned int x4,unsigned int x5,unsigned int fx1_3, set<Quint>& possibilities, bool insert = true){
  unsigned int f4,x2_4,fx2_4,f5,x3_5,fx3_5;
  size_t total=0;
  set<unsigned int> fFives;//to check if this value is an equivalent value to one we have checked already
  //set fx2_4
  x4 = (x4 % vmod) + (baseQuint.fourth << v);
  f4=f(x4);
  x2_4=(f3 / 2 + f2 / 4 + f4 / 4);
  fx2_4=f(x2_4);

  //brute force last element
    for(int x=0; x<upperMod; x++){
        x5= (x5 % vmod) + (x<<v);
    f5=f(x5);
    x3_5=(f4 / 2 + f3 / 4 + f5 / 4);
    fx3_5=f(x3_5);
        //Try all possibilities for last element
    if(x3_5 % vmod == targetx35 % vmod//check against triples
        && (fx1_3 / 4 + fx2_4 / 2 + fx3_5 / 4) % vmod == target % vmod){//and quints
          {
      if(fFives.count(f5 | 0x1)==0){//If already in, (ignoring lsb) this is effectively a duplicate.
        fFives.insert(f5 | 0x1);
        if(insert){
          baseQuint.fifth=x;
           possibilities.insert(baseQuint);
        }
```

```
            total++;
          }
            }
        }
      }
      return total;
    }
```

.2 Performance

The algorithm was implemented in two forms; a simple version to be run on a single system (single or multicore), and a distributed version, which was run for the final break. The distributed version consisted of a client that ran in a loop, querying a server for a work item to be performed, until no work remained. It performed the same procedure as the simple version, except only trying 1/10000th of the possibilities for the first triple in each work item. If no possible results were found at any stage; the algorithm stopped, sent its results back, and queried the server for another work item. The server continued serving out work items until each of the 10000 items was completed. Then the clients were instructed to halt. On average, only half of the items will have to be tried until a break is found. The break was achieved in about 10 hours, but a meaningful benchmark cannot be achieved from the run of the full break, since systems were added and removed throughout the process, and the program was run as a background low priority task to avoid interfering with user's activity. However, the single version was run in controlled conditions on a four-core processor over reduced problem sizes of $M = 10, 11, 12, 13, 14$ and $V = L = M - V$. These can be extrapolated to get an idea of the amount of work that the full break would take. The following graph shows the time that a break takes in seconds on a Intel Q6600 processor for each of the problem sizes. Fitting an exponential curve to the data produces the equation $T = 9.36/10^9 7.2512^L$ where T is time in seconds, which extrapolates to an estimate of 546547 seconds for the full break (6.32 days).

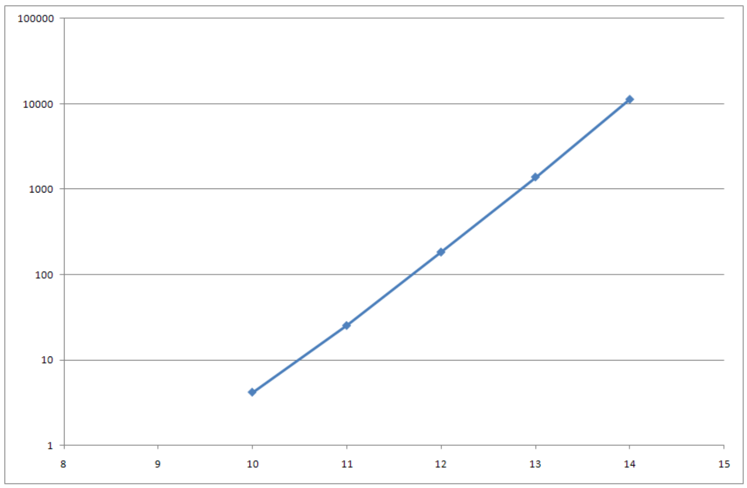


Figure 1: Break times for V=L=M-V

Much higher performance could be achieved by the use of the modern SSE instruction sets to perform operations on multiple pieces of data at once, or implementing the break in harware, such as the FPGA the proposal used.

Appendix B. Known Plaintext Attack Justification

A known-plaintext attack is usually justified if the known plaintext is not required to be large; such attacks were used against the enigma code in World War II. Today, most encrypted transmission of information is over the HTTPS protocol. Usually the header will include at least 130 consecutive bytes of known plaintext. For example, in each request, the Firefox browser sends the following consecutive headers as some of the first lines of the conversation. The actual body of the message, and any valuable information it contains, such as usernames, passwords, authentication tokens, etc., follows these headers.

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
```

These headers comprise 187 bytes and always more than sufficiently provide for 96 bytes of known plaintext aligned on a 32 bit block that is desired for the known plaintext attack implemented against the proposed cryptosystem in [4].