Cryptography with Chaotic Functions

John Stewart

December 4, 2006

Abstract

In this paper I will examine a couple of different ways in which chaotic functions can be used to create cryptographic schemes. That is, ways in which information can be encoded using a chaotic function, creating a cipher-text which is difficult to decode without knowing exactly all of the parameters used in creating the cipher-text. It will be convenient to first cover some of the basics of non-chaotic cryptography for the purpose of comparison with chaotic methods.

The chaotic encryption schemes dealt with here are symmetric key encryption schemes, as opposed to public key encryption schemes. That is, in order for the crypto-system to be used, it is first necessary for the encryption key to be transmitted from one party to the other, since the same key that is used to encode plain-text into cipher-text is also used to recover the plain-text from the cipher-text. Typically, this initial exchange is performed using a public-key cryptographic scheme.

Summary of Theory

The chaos-based cryptography schemes that I will be looking at face a fundamental difficulty. Because computers work with finite memory, it is impossible for them to represent all values in an infinite set - specifically, all values in a continuous interval of the real numbers. Because it is well understood by me, and it is a frequently used example in most literature, the function $f(x) = 4x(1-x)$ will be used exclusively throughout this paper. This

is of course a member of the quadratic parameterized family, often called the logistic map, ($f(x) = \mu x(1-x)$) which we examined in class, with parameter chosen as $\mu = 4$, the value for which the bifurcation diagram has a single bound, and shows chaotic behaviour. In general, any function which exhibits chaotic behaviour would be an acceptable substitute for f.

Chaotic functions have the following properties that make them well-suited to cryptographic applications:

1. Extreme sensitivity to initial conditions
    - This will ensure that, given even very slightly wrong parameters for key generation, significantly different keys will be produced.
2. Determinism
    - Because chaotic functions are completely deterministic, despite their random-seeming behaviour, proper computations of the precise parameters for key generation will always yield the same result, allowing two parties to independently generate the same key.
3. Statistically resilient
    - Because of the seemingly random behaviour of chaotic orbits, these sequences of numbers are not linked together in any obvious statistical way, meaning that various statistical methods will not be immediately effective on chaotic crypto-systems involving numerous keys related by chaotic orbits. It should be noted, however, that while this is currently the case, chaos-based cryptography has not proven itself against such attacks. Rather, because of the obscurity of chaotic encryption it has not come under sufficient scrutiny, or

resisted significant numbers of statistical attacks to say with firm certainty that such attacks would not work. What can be accurately claimed is that such attacks do not yet work, and developing attacks which do work is expected to be difficult.

Overview of Non-Chaotic Cryptography

Cryptography relies on the fact that some mathematical operations are difficult to undo without knowing the exact way in which they were done, but fairly easy to undo with complete information. For example, many cryptographic schemes rely on the fact that it is very difficult to factor a large number which is the product of two prime numbers (this is even more difficult to do in some groups, but that is not the topic of discussion here), but if one of the two prime numbers is known then the factorization is trivially easy.

In most applications, symmetric key cryptography is used for encoding all transmitted data, while while public key crypto-schemes are used to transmit the symmetric key from one party to the other, or information about how to create the symmetric key. This is because public-key schemes sometimes require hundreds of thousands of times more computations to both encrypt and decrypt data than symmetric-key schemes do.

In practice, symmetric keys are generated by a random number generator. For greater security, it is necessary to change the symmetric key frequently, so that statistical analysis of cipher-text will not yield the symmetric key. So a problem arises in that either a large number of symmetric-keys must be generated by one party and transmitted to the other, or there must be a way for each party to deterministically compute a next key given the previous one, in such a way that if one key is compromised the entire set of keys is not. This is specifically

where it is easy to see how chaotic functions could apply.

Finally, a symmetric key is typically a binary string of some length (a power of 2, for computational reasons). A key encrypts a plain-binary-string of the same length through an XOR operation to produce a cipher-binary-string of the same length. To recover the plain-binary-string, another XOR is performed with the cipher-binary-string and the key.

Chaos Cryptography

Chaotic Functions in Discrete Domains:

The first thing to be considered is that, as these crypto-system is destined for use on computers, it must be formulated to work on the computers finite memory. That is, we can no longer treat chaotic functions in the familiar setting of continuous domains.

A first observation to be made is that, on a discrete domain of size *n*, it is impossible for an orbit to have any greater than *n* values in it. This is fairly obvious from the fact that a single application of f will map from one discrete value to another, and once a value iterates to a value it has already reached, all subsequent iterates will have already been reached. It is also of interest to note that almost every orbit will in fact have many less than *n* elements in it. If the chaotic function is thought of as always choosing the next iterate perfectly at random, then once an orbit has half of the domain in it, each subsequent iteration is more than 50% likely to already be in the orbit, and halt the orbits growth. So we see that, as *n* becomes large, the probability of an orbit covering the entire domain falls near zero. So we can say that every point is either periodic of some order, or eventually periodic.

It is important to recognize the fact that there are likely to be periodic orbits of a wide range of sizes, from very small to very large. One of the things we will need to be wary of is

that the starting point we will choose for iterating is not either periodic of a small order, or eventually periodic of a small order, for reasons which will be evident later.

Our "chosen" *n* will correspond directly to the number of decimal points of precision that are carried through in computing iterations of f. It is certainly more relevant at this point to think of this less abstractly: with a given decimal precision, the continuous interval of interest, (0,1) (0 being fixed, and 1 being eventually fixed), is transformed into a discrete domain whose elements are all of the numbers in (0,1) expressible with the given precision. So, if *d* decimal places are carried through computations, then $n = 10^d$. It is a fact that, due to the chaotic nature of our calculations, after a finite number of iterative computations which carry more than *d* decimal places we will begin to see drastic deviations between the more and less precise computations. One related fact to note is that for a given key length, there is a minimum *d* which must be used, so that all possible keys of that size are available. For binary key length *k*, we would like to choose d such that the relation $10^d \geq 2^k$ holds, giving us the maximum number of keys. This gives a lower bound on desired *d* values, but increasing *d* will provide better security at the cost of more computations. Simple computations show that, to get all 64-bit keys, at least 20 decimal places should be carried. For 128-bit keys, at least 39 decimal places should be carried.

Algorithm 1: Simple Chaos Encryption

This algorithm will provide a method to create a symmetric key of some length using a chaotic function. Because only one key is used, this scheme is very vulnerable to attack. Indeed, only 2^k keys must be checked to produce the plain-text by brute-force.

The following information must be agreed upon by both parties, likely through use of a public-key algorithm:

- a starting value, x_0
- a decimal precision to be used, *d*
- a number of iterations, *n*

Given this information, computing a key is the simple process of selecting the k least significant digits of $f^{[n]}(x_0)$, expressed in binary, calculated using a decimal precision of d.

This gives us a single key to use in encrypting data via an XOR operation.

Also, it is reassuring to notice that this key generation process itself is fairly secure for *n* sufficiently large (and in reality, not very large at all). A very small variation in the starting value x_0 will lead to a drastically different key (extreme sensitivity to initial conditions). A decimal precision that differs by any amount will yield a drastically different key, as mentioned above. And of course, due to chaotic behaviour, a single extra (or fewer) iteration will lead to a completely different key.

Algorithm 2: Chaos Encryption with Multiple Keys

Algorithm 2 is the focus of [3].

This algorithm will extend the first algorithm in a natural way which takes greater advantage of some of the amazing properties of chaotic functions, and show why chaos is well-suited for cryptographic purposes. Algorithm 1 depended on the extreme sensitivity to initial conditions in order to satisfy the requirement that the starting value, x_0, must be known exactly in order to produce the correct key. Algorithm 2 will use this same basic fact to provide the security of the key generation process - that is, the fact that without precise knowledge of the parameters of key calculation, incorrect keys will be produced. Unlike algorithm 1, algorithm 2 will provide defense against statistical attacks, without requiring increasing amounts of data to be communicated between the two parties via expensive

public-key cryptography. Non-chaotically, this statistical security would be achieved by transmitting numerous keys between the two parties via expensive public-key cryptography, which may be an undesirable burden (though it does have up-sides - namely, the receiving party need not compute keys on its own, but at the cost of decoding a lot of public-key encrypted data, and a marginally increased network load which may present a problem in some applications).

The following information must be agreed upon by both parties, likely through use of a public-key algorithm:

- a starting value, x_0
- a decimal precision to be used, *d*
- a number of iterations before the first key is calculated, *n*
- a number of iterations between each subsequent key, *m*
- the number of times each key should be re-used, *l*

Given this information, the set of keys to be used are given by $f^{[n+im]}(x_0), i \geq 0, i \in \mathbb{Z}$, where a decimal precision of *d* is maintained throughout all calculations, and each key is the *k* least significant digits of a specific iterated value expressed in binary form. It will, of course, be least intensive to compute them each in turn, meaning that to compute *r* distinct keys, a total of *n+(r-1)m* iterations of x_0 must be calculated by each party. This *i* will be determined by the message length. Specifically, if a message is divisible into *m* parts of length *k*, then $i = \lceil \frac{m}{l} \rceil$ is the number of different keys which must be generated to encode or decode the message.

Individually, each of these keys is generated by a slight modification of algorithm 1, meaning that each individual key retains the benefits ascribed to the keys generated by

algorithm 1. Namely, small variations in any of these parameters (with the exception of *l*) will yield a radically different key. Obviously, if this is true for each individual key then it is true for all keys in algorithm 2: a small variation in any parameter will not yield any correct keys at all.

The real advantage gained by algorithm 2 over algorithm 1 and conventional algorithms comes from the fact that numerous keys are used, and that these keys are related by a chaotic function.

In algorithm 1, vast amounts of data would be transmitted using a single key. This is generally a bad idea, because while the chaotic nature of the key generation may provide security, the fact that the key is just a number leaves the encryption susceptible to frequency analysis. That is, for example, suppose that a large document was transmitted, and each letter in the document was encoded separately, but by the same key. Then constructing the key would be a matter of finding the most common cipher-text block, and determining what keys would encrypt that to the most common letters in the alphabet (e and a). By using multiple keys this frequency analysis is made remarkably more difficult.

In typical applications of symmetric key cryptography, it is very common for large number of keys to be used to encrypt data. There are two ways to accomplish the sharing of a set of keys. Either all keys can be fabricated (ideally, by a perfectly random process) by one party and transmitted securely to the other party (which can be an expensive process), or the method (parameters) for creating a set of keys can be transmitted from one party to the other, which is the approach taken by algorithm 2. This approach is certainly dangerous. If one session key is compromised and the attacker is able to deduce other keys from the compromised one, then the security of the whole transmission is nullified. This is where we see a chaotic key-generating function shine. In order for an attacker with

knowledge of one (compromised) key to determine any other key, the attacker still must know *d* and *m* explicitly. That is, without knowledge of the parameters of key generation, one key will not allow an attacker to easily guess other keys. Because the chaotic function is resilient to statistical analysis techniques, it would not be easy to determine the key generation parameters from even a sizable number of compromised keys. It should also be noted that statistical analysis can, in some cases, compromise even sets of keys which are generated "randomly" by one party and transmitted to the other. Further security against subsequent keys being deduced from a compromised key is given if *d* is chosen larger than necessary to supply *k* binary digits, because the most significant digits are dropped from the key, which means that an attacker would not be able to tell where in the interval this key came from.

Certainly it is possible for new statistical methods targeted at chaos-based crypto-systems like this one to be developed, but as chaos cryptography is rarely used, such methods have yet to be developed, if such methods are possible. This is actually one of the fundamental problems in cryptography - it is often not possible to prove the resiliency of a crypto-system to attack without there being sufficient motivation and time for attackers to thoroughly attempt to penetrate it.

In [3] the authors evaluate this algorithm from the perspective of an attacker who realizes that it is infeasible to guess all of the keys individually, so the algorithm is attacked by trying all possible values of x_0. If the interval [0,1] was divided into 10^{30} discrete values, the attacker must attempt 10^{30} values, with an added complexity (brought on by the other parameters in the crypto-system) of roughly 10^7, making the number of attempts required to crack the encryption on the order of 10^{37}. A typical DES implementation, by contrast, would take roughly 10^{19} attempts to crack, which is, needless to say, significantly fewer attempts.

This comes at the cost of more intense computation for both parties involved in the information exchange, relative to DES.

The biggest problem with this algorithm, to me, is that the discreteization of our interval of chaos may create many orbits of small period, and if any of these orbits are reached in key generation (especially if the size of this small orbit is not coprime to *m*), then there is a danger or repeated keys. As the number of discrete points in our interval (which is a function of *d*) increases, the frequency of these small orbits should decrease, and the likelihood of this danger drops.

Algorithm 3: Chaos Encryption with Multi-orbit Keys

The idea behind algorithm 3 came to me as a result of reading [4], which described how multi-dimensional maps could be used to greatly increase the periods of orbits in discrete domains, and thus reduce the frequency of small-period loops. Basically, the idea is that a few different orbits should be followed, and their elements should be combined to create the keys. If one orbit should come upon a small-period loop then the other orbits would continue to provide variety in subsequently generated keys. Algorithm 3 can be thought of as an extension to algorithm 2 which insulates against the danger of small period orbits causing key repetition.

There is a delicate balancing act which must be performed in choosing suitable parameters at this juncture. Because each key will be created from several discrete values, less of the information in each discrete value is needed, so the number of decimal points carried in calculations can be reduced. However, such a reduction in the number of discrete values would result in more frequent small-order orbits, meaning that in order to ensure that keys are not repeated more orbits must be used. And of course, using more orbits with

greater decimal precision increases the computation load which is required.

The following information must be agreed upon by both parties, likely through use of a public-key algorithm:

- multiple starting values, $x_0, x_1, x_2, ..., x_j, j|k$ (in practice, I would think that *j* would never exceed 8, but generality is not a burden)
- a decimal precision to be used, *d*
- a number of iterations before the first key is calculated, *n*
- a number of iterations between each subsequent key, *m*
- the number of times each key should be re-used, *l*

Given this information, a key of length k is generated by taking the $\frac{k}{j}$ least significant digits from keys generated by algorithm 2 for each of the x_j, and concatenating them onto each other in the order of *j*'s.

This is an effective way to deal with the small-orbit problem presented by discrete domains. However, there is much analysis which would need to be done to determine the feasibility of this algorithm. For example, it is possible that by ignoring so much of the key which is generated by the chaotic function, some vulnerability may be exposed. An added bonus that this algorithm provides is the extra security of a compromised key not reveling completely the associated orbit values. For example, they key may provide the last 8 binary digits of the keys, but be lacking the first 56 binary digits.

Conclusions:

It is my opinion that the field of chaos-based cryptography holds much promise in future applications of symmetric key crypto-systems. That said, it is really impossible to

know how well these systems would stand up to attack before they are used, and experienced attackers attempt to penetrate the algorithms. I am interested in exactly how well algorithm 3 prevents repeated keys, and specifically how many orbits are needed to reduce the likelihood of orbital loops to a statistically insignificant level. This is certainly a topic which, in my mind, warrants further investigation.

References

[1] *An Overview of the History of Cryptology,* Publication by the Canadian Communications Security Establishment (CSE)

[2] Vishaal Kapoor *Chaos and Cryptography* December 4, 2003

[3] Dr. Ranjan Bose and Amitabha Banerjee, *Implementing Symmetric Cryptography Using Chaos Functions*, Indian Institute of Technology, Hauz Khas, New Delhi

[4] A. N. Pisarchik, N. J. Flores-Carmona and M. Carpio-Valadez *Encryption and Decryption of Images With Chaotic Map Lattices,* CHAOS, And Interdisciplinary Journal of Nonlinear Science, September 2006