The Enigmatique Toolkit

by Christopher Billings

Keywords: poly-alphabetic multi-algorithmic symmetric substitution cipher

Abstract: This paper describes a new method of creating systems of poly-alphabetic, symmetric ciphers with a large set of algorithms. The method uses two translation tables containing the code-text character set, one for encryption and the other for decryption. After each character is encrypted or decrypted, these tables are permuted through a series of pseudo random, pairwise swaps. The implementation of alternative swap formulas leads to systems with large sets of encryption/decryption algorithms. Systems that contain more than a googol (1E100) algorithms have been easily implemented. An algorithm set can be easily sub-setted, producing hierarchical systems where a program may contain a single algorithm or a portion of the full set. The strength of these systems has not been determined. However strength can be inferred through the use of a substantial number of numeric keys , pseudo random number generators, algorithm switching, and other features.

Introduction:

The term 'Enigmatique' designates a general method of generating computer programs that contain a vast number of encryption/decryption algorithms. In technical language, Enigmatique is a general method for creating multi-algorithmic, poly-alphabetic subsitution ciphers. Here multi- means millions, billions, trillions, ... and poly means 256 factorial.

Enigmatique can be configured to use alphabets of any size. An alphabet with 256 characters is the most useful since it can easily encrypt any computer file.

Modern cryptography has left the classic substitution ciphers far behind. One reason is that these ciphers use only a small number of alphabets (permutations of the characters). The German Enigma machine suffered from the defect that it could only employ only a tiny fraction of the 26! permutations of 26 letters.

Enigmatique was inspired by classical substitution ciphers of the pre-computer era. But Enigmatique features a strong avalanche effect. It employs an internal state that performs a random walk through the N! permutations of an N-valued codetext character set. And with 2^R algorithms (for an R-bit algorithm number), Enigmatique uses techniques that require the power and accuracy of modern digital computers.

The ability to create systems with vast numbers of encryption algorithms -- by the millions, billions, trillions, ... -- offers special capabilities previously unknown in cryptography. As a simple example, a program for a server could contain four billions algorithms. Then each user can be provided with an optimized computer program with only one single algorithm. Each of these user-level programs is unique and incompatible with the others. Communications between users is established through the server-level program that handles any and all of the user-level algorithms.

Enigmatique provides a new twist on security. Conventionally, it is assumed that all users have the same encryption algorithm. If an attacker steals one program, he can identify the exact algorithm used by everyone. Thus security depends upon keeping the keys secret. In Enigmatique this type of attack is impossible, unless the master program is stolen. None of the user programs provides information sufficient to attack the communications of any other user. Organizations may concentrate their security on the guarding of the master program. The security of user-level programs, while important, are less acute

with Enigmatique.

This does not overthrow the Kerkhoff Principle. But it raises a new obstacle for an attacker.

The heart of an Enigmatique system requires a series of special formulas that are diverse, convoluted, without apparent pattern, and easily written. These special formulas are used for both encryption and decryption, guaranteeing that whatever is encrypted can be decrypted in the same amount of time.

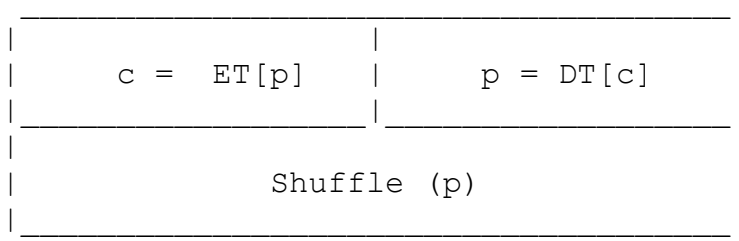
Someday, somewhere, a graduate student will write a program that generates a series of these special formulas random. This will be a milestone in the history of cryptography -- a system with billions of randomly generated encryption algorithms.

This will bring to mind the story that Donald Knuth tells of his mis-guided attempt to devise a “Super Random” number generator only to find that the output degenerated, sometimes producing a constant series. Donald Knuth emphatically abjures, “random numbers should not be generated with a method chosen at random.” [1] It goes without saying that nothing in cryptography can be trusted without careful testing.

Enigmatique makes all this possible!

Basic Concept:

Enigmatique is founded on a simple concept as illustrated in this diagram:



The box on the upper left represents the first step in the encryption process. ET

1 Knuth, Donald; “The art of Computer Programming vol. 2; Seminumerical Algorithms” p6.

represents the encryption table. In the general case, the array ET contains a permutation of the numbers 0 ... n-1 (for an n-valued code text character set). Array DT contains the inverse permutation. The encryption routine converts each plain text character (p) to an index of this array. Then it takes the value in the indexed position and converts it to the code text character (c).

Likewise, the upper right box represents the first step in the decryption process. The code text character (c) converts to an index of the decryption table (DT). Then the value in the that position is converted to the plain text character (p).

The above rather technical discussion applies to the general case where the code text character set is a subset of the full character set. However, very often the code text character set is the full 256-character set. Then the numerical value of a character coincides with its value as an index.

It is the Shuffle function that gives Enigmatique its special character. Without the Shuffle, the encryption degenerates to a simple substitution cipher, or to a very inefficient copy program. The Shuffle function initiates a pseudo-random permutation of ET by means of a series of pair-wise swaps. At the same time, corresponding swaps are performed on DT to keep it synchronized with ET. At all times the two tables must contain inverse permutations of each other.

Note: The discussion below assumes that the implementor uses a 32-version of the C/C++ language.

The Shuffle function contains several sequences of the form:

```
i +=  pseudo-random-calculation-1;
j +=  pseudo-random-calculation-2;
Swap(i,j);
```

Since both i and j may have any 32-bit value, the Swap function must apply a modulus to these values in order to reference ET and DT. The Swap function interchanges the selected entries of ET and corresponding entries of DT.

The first swap in Shuffle is slightly different. It reads:

```
i += p;
j +=  pseudo-random-calculation;
Swap(p,j);
```

This code creates a strong avalanche effect. Here p is the plain text character which is used as the argument to Shuffle(p). This ensures that doubled letters, as in 'bookkeeper' are encrypted to different code text characters. This also ensures that sequences of a single character (that may occur in office documents) are not easily identifiable.

It is recommended that i and j are modified only by addition, subtraction, or by XOR. On byproduct of this feature is that the initial values of I and j become special keys. This also strengthens the avalanche effect.

A wide variety of formulas can be used in the pseudo-random calculations. For example:

```
i += K(E(i+5))^R(D(j));
j ^=  Z(D(7-j)*397+E(i));
Swap (i,j);
```

These formulas uses several functions. The original Enigmatique system uses these:

K(m) - returns an entry from a table of keys. Since m may be any 32-bit value, the function must apply a modulus to obtain a valid index to the table of keys. Any number of keys may be used.

E(m) - returns an entry from the encryption table.

D(m) - returns an entry from the decryption table.

R(m) - returns a value from the pseudo random number generator. The argument is used to apply a rotation and an optional complement to the result. In addition, the implementation may also use m to select a seed from an array of seeds. The PRNG used by the author is a simple congruential generator with a 32-bit seed, but which returns a 16-bit result. As a design decision, a future implementation may replace this PRNG with a different one.

Z(m) - hashes the argument, using the same congruential formula as the PRNG.

RL(m) - combines the results of two calls to the PRNG into a 32-bit result. The argument is used to apply independent rotations and complements to each half of the result.

X(m) - is an unspecified function to be defined in a future implementation. The employment of such functions is a design decision.

In addition, these formulas use constants and the variables i and j. The operators plus, minus, complement, and XOR may be used freely. Multiplication, division, and other operators may be used provided precautions are taken against the loss of randomness.

The basic requirement is that the shuffle routine produce exactly the same results for decryption that it does on encryption. This requirement is easily met. Shuffle functions constructed as described above are guaranteed to work properly. Everything encrypted can be decrypted. Furthermore, encryption and decryption take the same amount of time.

Multi-algorithmic systems

Up to this point we have not discussed the most significant innovation of Enigmatique, the method of creating multi-algorithmic systems. This is done by exploiting one of the special features of the C/C++ language, conditional expressions. The syntax for a conditional expression is:

```
( conditional-exp ? expression-t : expression-f )
```

When a conditional expression is executed, the conditional expression is evaluated. If the result is true, i.e. non-zero, expression-t is evaluated and its value is returned Otherwise, expression-f is evaluated and its value is returned.

Enigmatique uses conditional expressions in the following way:

Example A:

```
i += (alg&b1 ? special-formula-t1 : special-formula-f1);
j += (alg&b2 ? special-formula-t2 : special-formula-f2);
Swap (i, j);
```

Here alg is some portion of the algorithm number. b1 and b2 are used to test a

particular bit in the algorithm number. They have the values 1, 2, 4, 8, ... Suppose that the result of the first test is true and the second is false. Then the above three statements reduces, in effect, to:

Example B:

```
i += special-formula-t1;
j += special-formula-f2;
Swap (i, j);
```

Suppose that the two tests of the algorithm number yielded the opposite results. Then the above reduces to:

Example C:

```
i += special-formula-f1;
j += special-formula-t2;
Swap (i, j);
```

When the program sets the algorithm number to a specific value, that selects the code for either B or C or two other examples that are not shown here. The value of the algorithm number determine which formulas are executed and which are bypassed, thus selecting different encryption algorithms.

A basic Enigmatique system uses two special formulas for each bit of the algorithm number. One formula is executed when the algorithm bit is zero, the other when the algorithm bit is one. A basic system that uses a 32-bit algorithm number thus has sixty four special formulas. When the algorithm number is set to a specific value, one half of the formulas are activated, the other half bypassed. Each setting of the algorithm number thus selects one of more than four billion encryption algorithms. Each additional bit in the algorithm number doubles the number of encryption algorithms. Thus with a 64-bit algorithm number, a system can have more than 17 billion billion encryption algorithms, and a 96-bit system has 70 billion billion billion algorithms, etc., etc., etc.

As stated above, a basic Enigmatique system uses two special formulas per algorithm bit. However, other alternatives may be employed. For example, an implementation may use four bits to select one of sixteen formulas (as shown in an example below.) Then additional formulas are required.

Coding Example:

Consider this example based on a working program:

```
#define ALT(a,b,f,g)  (a&b ? f : g)
enum bits {B0=1, B1=2, B2=4, B3=8, B4=16, B5=32, B6=64, B7=128};
static unsigned long SAVED_J_VALUE = 0;
void Shuffle (unsigned long plain)
{
      unsigned long i = SAVED_I_VALUE + plain;
      unsigned long j = SAVED_J_VALUE ;


      j += ALT(A0,B0,ALT(A0, B1, 17*R(i+12)^K(29*j),
                                 R(DT(i*11))-K(j+5) ),
                     ALT(A0, B1, R(E(i+19)*K(i*77+5)),
                                 R(i)^K(D(j)) ));

      Swap(plain,j);

      i += ALT(A0,B2,R(i)^K(1),23*R(i+ 3)^K(71*i));
      j ^= ALT(A0,B3,Z(E(i+97)),R(D(i*23)))  ;
      Swap(i,j);

      i -= ALT(A0,B4,R(i)^K(2),29*R(i+25)^K(34*i));
      j += ALT(A0,B5,R(D(i+71)),R(E(i*97)))  ;
      Swap(i,j);

      i += ALT(A0,B6,R(i)^K(3),47*R(i+31)^K(51*i));
      j ^= ALT(A0,B7,R(E(i+59)),R(D(i*101)))  ;
      Swap (i, j);

      i +=  ALT(A0>>8,B0,41*(j+3),(D(j*19) * K(4)) ^ R(i) );
      j -=  ALT(A0>>8,B1,E(j+171)*K(j),D(i+20));
      Swap (i, j);

      j +=  ALT(A0>>8,B2,(D(i + j)*K(5)) - 77, R(E(E(j-5)))) ;
      i ^= ~ALT(A0>>8,B3,D(j),13*E(i+5)) ;
      Swap(i,j);

      i +=  ALT(A0>>8,B4,K(i+8),R(j)) ;
      j +=  ALT(A0>>8,B5,E(j+17)*K(i),D(i+20));
      Swap (i, j);

      j +=  ALT(A0>>8,B6,E(i-7),D(D(j+99)));
      i *=  ALT(A0>>8,B7,K(E(i+29)),R(D(i+j)))  ;
      Swap (i,j);
```

```
      i +=  ALT(A0>>16,B0,K(i+8),R(j)) ^
            ALT(A0>>16,B1,E(j+17)*K(i),D(i+20));
      j +=  ALT(A0>>16,B2,E(i-7),D(D(j+99))) *
            ALT(A0>>16,B3, R(E(i+20)),Z(R(5+i)))  ;
      Swap (i,j);

      i +=  ALT(A0>>16,B4,K(i+8),R(j));
      j ^=  ALT(A0>>16,B5,E(j+17)*K(i),D(i+20));
      Swap (i, j);

      j +=  ALT(A0>>16,B6,E(i-7),D(D(j+99)));
      i ^=  ALT(A0>>16,B7,E(i)*17,R(5))  ;
      Swap (i, j);


      //  just for fun
      #define QUAD(alg,ba,bb,e,f,g,h) \
            ALT(alg, ba, ALT(alg,bb,e,f), ALT(alg,bb,g,h))



      i +=  QUAD(A0>>24,B0,B1, E(j+17)*K(i),D(i+20),R(E(i
+20)),Z(R(5+i)));
      j +=  QUAD(A0>>24,B2,B3, E(i-7),D(D(j+99)),R(E(i+20)),Z(R(5+i)));

      Swap (i,j);


      i -=  QUAD(A0>>24,B4,B5, K(i+8),R(j),E(j+17)*K(i),D(i+20));
      j ^=  QUAD(A0>>24,B6,B7, E(i-7),D(D(j+99)),E(i)*17,R(5));
      Swap (i,j);


      SAVED_J_VALUE = j;
      SAVED_I_VALUE = i;
}
```

Ladies and gentlemen, you have just seen four billion encryption algorithms. And there are endless variations on this theme.

Single-algorithm encryption programs

The master source code for an Enigmatique system may be used to create a series of low level programs, each containing a single, unique encryption algorithm. This exploits the fact that many compilers, for many different

languages, can produce optimized code. Consider the folowing expression:

```
i += ( alg & 4 ?  f : g);
```

Suppose that alg is a variable. Then the compiler generates code for the entire expression. Both formulas f and g will appear in the compiled code. This appears in the master encryption program.

On the other hand suppose that alg is a constant, say 6. Then the program will always execute f, but never g. The compiler will recognize that formula g is dead code and ignore it. The compiler will also recognize that the conditional expression is also dead code and ignore it. The compiler will generate code for the following:

```
i += f;
```

All trace of the algorithm number and the formula g is eliminated from the compiled program.

To take advantage of this capability, a special header file specifies the algorithm for each user of the system. Let's call this header file UserData.h. And suppose the algorithm number is stored in an array defined by:

```
    unsigned long AlgorithmNumber [3];
```

To generate the master program, the header file UserData.h contains:

```
#define A0  AlgorithmNumber [0]
#define A1  AlgorithmNumber [1]
#define A2  AlgorithmNumber [2]
```

(Refer to the large example above which uses A0 as the algorithm number.)

Here the algorithm number is completely variable. That means that the compiler will generate the master program with a set of 70 billion billion billion algorithms. To generate a program with a single algorithm, the symbols A0, A1, and A2 must be defined as constants with specific values. For example:

```
#define A0   531691946ul
#define A1   740517342ul
#define A2    32759624ul
```

Now the compiler will recognize that half of the special formulas are dead code, and eliminate them from the compiled program. The result is a program optimized for a single algorithm. If an attacker steals such a program, he will be able to analyze that one algorithm. But he will not be able to reconstruct any other algorithm.

In addition, intermediate level programs can be created. For example:

```
#define A0  AlgorithmNumber [0]
#define A1   740517342ul
#define A2   32759624ul
```

creates a program with four billion algorithms and

```
#define A0   531691946ul
#define A1  AlgorithmNumber [1]
#define A2  AlgorithmNumber [2]
```

creates one with 17 billion billion algorithms. And there are many variations on this theme.

Sixteen-fold Swap

This is a example of an initial swap where four bits in the algorithm number are used to select one of sixteen special formulas:

```
#define ALT(alg,Bn,exp1,exp2)  \
     ((alg) & (Bn) ? (exp1) : (exp2))
#define QUAD(alg,b1,b2,f1,f2,f3,f4)\
     ALT(alg,b1,ALT(alg,b2,f1,f2),ALT(alg,b2,f3,f4))


     j ^= QUAD(A0,B1,B3, QUAD(A0,B0,B2,FA,FB,FC,FD),
          QUAD(A0,B2,B0,FE,FF,FG,FH),
          QUAD(A0,B2,B0,FI,FJ,FK,FL),
          QUAD(A0,B0,B2,FM,FN,FO,FP));

     Swap(plain,j);
```

Design Decisions

Enigmatique offers the implementor a vast array of design decisions. How may algorithms are to be implemented? How many swaps will be used. How many special formulas are needed? What will they look like? Which pseudo random number generator should be used? Should multiple PRNG's be used? Or a single PRNG with multiple seeds?

How many numeric key words should be used? In the general method there is no limit on the number of key words that can be used. It is up to each implementation to impose a limit, say 100 words (3200 bits). The key space can be made so large that brute force attacks are impossible. Also the number of keys can be made variable. One encryption run may use, say, five 32-bit key words, and another thirty. It is up to the implementor to write a k(m) function that can handle the alternatives.

It was stated above that the k(m) functional must apply a modulus to its argument in order to index an array of keys. If, say, four keys are used, then

```
KeyTable [m%4]
```

uses only two bits of m. To bring more bits into play the program may use a double modulus, for example:

```
KeyTable [ (m%(4*KeysInUse-1)) % KeysInUse ]
```

Enigmatique allows the implementor to adjust the trade off between speed and strength. Any Enigmatique system can be strengthened by making the formulas more complex, or by adding swaps to the system. Conversely, a system can be made faster by simplifying the formulas or reducing the number of swaps.

Many simple tweaks to a system will create a new encryption system which is not compatible with the original.

Applications

There are several ways to utilize an Enigmatique system. If Alice and Bob have identical encryption programs with, say, four billion algorithms, they could utilize, not one time pads, but one time algorithms.

Enigmatique lends itself to hierarchical systems. For example, Alice and Bob may have programs where the two programs implement different algorithms. In this case, Alice and Bob can not send encrypted messages back and forth directly. They will send their messages to an email server which decrypts each message and re-encrypts it for the other person. Suppose the master program implements four billion algorithms. This system an theoretically support four billion users, each having a single, unique algorithm. Alternatively, it could support, say, four million users, each using a different set of 1024 algorithms, or four thousand users, each having one million algorithms.

Furthermore this system can be organized into a hierarchy tree. Each bottom user assigned one thousand algorithms, a department server with one million algorithms, and a corporate server with the full set.

There are endless variations on this theme. If additional levels are needed, then another word can be added to the algorithm number, providing up to 17 billion, billion algorithms.

Security is enhanced when users have different (non-overlapping) algorithm sets. Alice can not decrypt Bob's messages and vice versa. This is useful, even in a business environment, when Alice and Bob may be both coworkers and bitter rivals.

This system raises the bar for attackers. If Trudy should steal Bob's laptop, she can read Bob's messages. That will do damage to the company. But the damage will be limited. Even if she decodes Bob's encryption algorithm, she cannot read the messages for Alice or Dave or Edward or any other user. The reason is that Bob's encryption lacks code that is essential for the other algorithms. If Trudy cannot guess or determine the formulas required by other users, she cannot read their messages.

Hierarchical Systems

There are many ways to set up a hierarchical Enigamtique system. Imagine an inverted tree network. The root (top) of the tree containss the entire set of encryption algorithms. Nodes on the branches, twigs, and leaves contain smaller and smaller subsets. There are two basic rules in this scheme. First, if one node is under another, then it contains a subset of the algorithms of the higher node. Second, any other pair of nodes have no algorithms in common.

It is useful to assign multiple algorithms to every leaf (low-level user) in this tree. Then algorithm switching can occur within a message. (See below). It is convenient to assign the bits of the algorithm number in ten bit groups. Then each level hierarchy can be assigned a number between 0 and 1024. A 32 algorithm number can be written, for example, as 0.999.414.503. This number may stand for:

Group number:	999
User number:	414
User's algorithm:	503

The remaining two bits may used for a super-group, or may be reserved for special purposes, for example:

0	Normal traffic, e.g. email
1	Archive of ordinary company documents
2	Archive of company trade secret documents
3	Archive of military classified documents.

This so far, will not be adequate for many organizations. Additional levels of hierarchy can be easily provided for by adding one or two 32-words to the algorithm number. Each 32-bit word of the algorithm number multiplies the number of algorithms by 4.29 billion. Thus two words can provide 17 billion billion algorithms, and three words 70 billion billion billion algorithms.

A basic scheme may allocate two formulas for each bit of the algorithm number. An alternative, is to allocate the bits by twos and threes. This uses a new macro called OCTET.

```
i += QUAD(alg, bits,  Four formulas);
j += OCTET(alg, bits, Eight formulas);
Swap (i,j);
```

Each swap uses five bits of the algorithm number. Under this scheme, each word of the algorithm number generates seven swaps. (As opposed to sixteen swaps in the basic scheme.) A critical factor is the number of swaps being performed. This method encrypts twice as fast, but with a possible loss of strength.

Special Features

A number of special features can be incorporated in an Enigmatique system. For example:

1. Text keys. A text key is used by both the encryptor and the decryptor. Both ends encrypt the text key at the beginning of the message, then throws the code-text away.

2. Algorithm switching by users that have multiple algorithms. A small number is chosen, say n=35. After each character is encrypted or decrypted, the value i mod n is calculated. If zero the algorithm number is changed using the quantity j. Thus after each character is processed there is one chance in n that the algorithm is changed.

3. Some of the formulas use a quantity k (small k). This quantity is set originally to a small odd number. After each character is processed, k is incremented by two. When it reaches a certain limit, it is set back to its original value. This produces a sawtooth effect.

These techniques provide a 'toolkit' for those who implement Enigmatique systems.

Advances in the Art

Enigmatique advances the art and craft of cryptography in five important areas:

1. This is the first method that allow the creation of systems with vast numbers of encryption algorithms. The number of algorithms that can be used is unlimited.

2. Enigmatique exploits the n factorial permutations of an n-valued character set. This is more than 8.578 times 10 to the 506th power. An Enigmatique system can utilize only a tiny fraction of this number -- because the universe will not last long enough.

3. The difference between encryption and decryption is trivial. Decryption can be guaranteed to work properly and operate is the same time as encrption.

4. Enigmatique is based on simple principles that can be easily taught. A competent programmer can create a powerful system in a few days.

5. The strength of Enigmatique systems is not fixed. Any Enigmatique system can be strengthened with small changes to the program. I am confident that if my encryption programs prove to be weak, that someone, using this method, will create much stronger versions of Enigmatique.

The Challenge

The ultimate question for any system of cryptography is how strong is it? I am not willing to make claims that I cannot prove. Some of the claims I have made above may appear to be snake oil. Such a notion should be dispelled by a careful examination of this document. And I can prove everything with working code. The ultimate answer to any question is: Read my code!

Please note that the author has made not any claims about the strength of this encryption, other than the mild statement (5) above. Basically, definitive statements about it strength are premature. The history of cryptography is littered with the wreckage of many broken systems, including some of the best efforts of outstanding cryptographers. It is hoped that Enigmatique will not share that fate.

Therefore Enigmatique is not offered as a secure system. Instead, it is ofered as a challenge.

Please note that Enigmatique is not a specific algorithm, but a general method. There will be many different implementations of this type of system, some strong some weak. The strength of an Enigmatique system will vary with the skill and care exercised by the cryptographer.

The challenge for cryptographers is to identify features that contribute to the strength of the system. How many swaps are needed for a strong system? what types of special formulas strengthen or weaken the system? Are there weak keys or problematic plain texts?

The challenge for cryptanalysts is to find ways of breaking these systems. Exhaustive key search will not work since the keyspace can be made very large. It is unlikely that any form of linear or differential cryptanalysis will work. The arithmetic operations and permutations used in Enigmatique should disrupt any linear or differential relationships.

A successful cryptanalysis of a good Enigmatique system may require a new type of cryptanalysis, and that it will be a general type of attack that works against any Enigmatique algorithm. It seems unbelievable that a trillion algorithm system can be broken one algorithm at a time.

Perhaps the most interesting feature of Enigmatique is its conceptual simplicity. To truly understand a block cipher takes a strong mathematical background and a lot of perseverance. Enigmatique, on the other hand, can be used by ordinary programmers to create 'difficult' ciphers.

Conclusion

The ability to create efficient systems with billions or trillions of encryption algorithms is a new development in cryptography. Such systems present special obstacles to those attempting to break cryptographic systems -- whether for good or for evil. The simplicity of the Method Enigmatique means that ordinary computer programmers can create powerful systems in a short amount of time. Around the world there are thousands of people who have the background, the skills, and the talent necessary to invent a system like Enigmatique without any knowledge of the work that is described in this paper. All that is missing is a flash of inspiration.

The Enigmatique Toolkit provides a set of techniques that can be used selectively to build a special type of cryptographic systems. Source code is available at www.enigmatiquecryptographia.com