A. Cover sheet

Name of submitted algorithm: FROG

Principal submitter's name: TecApro (represented by Dianelos Georges Georgoudis)

Telephone: 011-506-2243434

Fax: 011-506-2531496

Organization: TecApro Internacional S.A.

Postal Address: Apdo 857-2050 Costa Rica

E-mail address: dianelos@tecapro.com

Names of auxiliary submitters: none

Names of algorithm inventors/developers: Dianelos Georges Georgoudis,
Damian Leroux, Billy Simón Chaves

Name of algorithm owner: TecApro

Submitter's signature:

Backup point of contact:
Telephone: 011-506-2344400, 011-506-2342416
Fax: 011-506-2344401
Postal Address: P.O.Box 25216-630, Miami FL 33102, USA
E-mail address: tecapro@ibm.net

Table of Contents

B. Algorithm Specifications and Supporting Documentation

B.1. Introduction

FROG is a new cipher with an unorthodox structure. Any symmetrical cipher's job is to conceal the plaintext's information through a computational process of confusion and diffusion. The basic idea behind the design of FROG is to conceal the definition of most of this process in a secret internal key. The actual encryption algorithm operates as an interpreter that regards the secret internal key as a program and executes it as if it were a series of primitive instructions. The goal is to deny the attacker as much knowledge as possible about the actual process being performed and therefore defeat any attack, whether publicly known or not. The encryption and decryption operations used in FROG are extremely simple. All the complexity lies within the internal key the details of which are unknown to an attacker.

FROG is very easy to implement (the reference C version has only about 150 lines of code). Much of the code needed to implement FROG is used to generate the secret internal key, the internal cipher itself is a very short piece of code. The implementation will run well on 8 bit processors because it uses only byte level instructions - the only arithmetic operation used is exclusive-OR and, optionally, 1 byte modulus. Also no bit specific operations are used. The algorithm is fairly fast, a version implemented using 8086 assembler achieves processing speeds of over 2.2 Mbytes per second when run on a 200 MHz Pentium PC. A version implemented using Pentium specific assembler would run even faster.

The FROG reference program included in this submission allows for user keys of any length between 5 bytes and 125 bytes (i.e. between 40 bits and 1000 bits in multiples of 8 bits). The block size is defined as a constant and is set equal to 16 bytes which is the minimum requirement for AES. Nevertheless, this constant can be given any value (even odd values) between 8 and 128 bytes in which case the re-compiled code will encrypt blocks whose sizes vary from a minimum of 64 bits up to a maximum of 1024 bits.

B.1.0. Conventions and Organization of this Document

For simplicity, the description that follows is based on a block size of 16 bytes. No bit naming convention is specified because no bit operations are required. Byte values that appear in the examples are given as two digit Hex numbers starting with the most significant digit, e.g. A1 corresponds to the decimal value 161. FROG only uses byte arrays which are represented as V[I] where “V” is the name of the array and “I” is the index. An index of zero denotes the least significant byte of the array (i.e. the byte at the lowest position in memory).

Each part of the algorithm is formally specified using pseudo code. The pseudo code itself is not defined because its meaning should be clear to any programmer familiar with current day programming languages.

The diagrams used in the document are built from two elements: rectangular boxes that represent data and rounded boxes that represent processes. Figure 1, which shows the high level structure of FROG, illustrates the use of these elements.

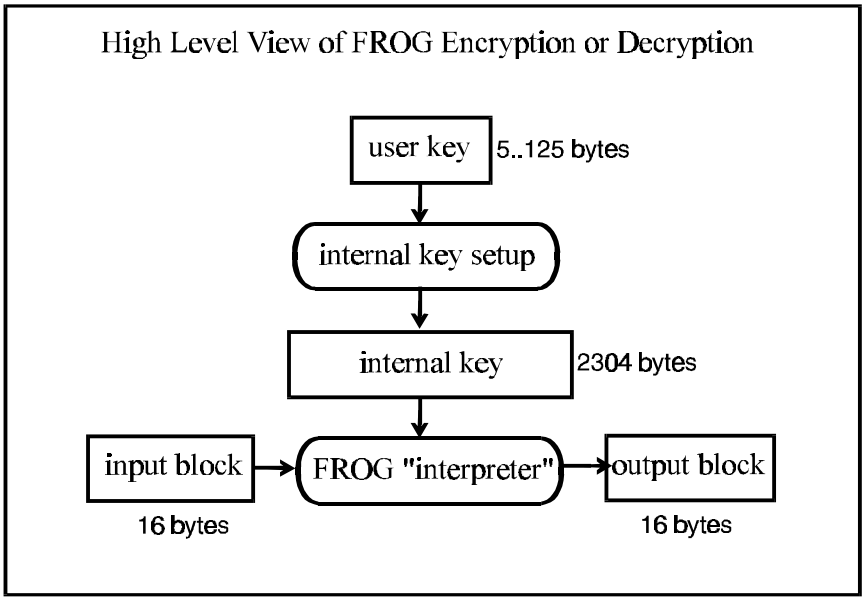

Fig. 1

Section B.1.1. explains how the algorithm's internal cycle, the FROG "interpreter" works. Sections B.1.2. - B.1.4. describe the internal key setup process of FROG. The organization of the rest of the document closely follows the structure specified in the NIST's requirements for the documentation of candidate algorithms.

B.1.1. Internal cycle

FROG uses 8 iterations. Each iteration uses one record of the internal key (called *internKey*), which is a data structure with eight records. Each of these records has three fields: an array of 16 bytes (called *xorBu*) which are used in an initial exclusive-OR operation with the block bytes, an array of 256 bytes (called *substPermu*) which represents a substitution table for byte values, and an array of 16 bytes (called *bombPermu*) each of which points to a different byte positions within the block (and therefore has a value between 0 and 15).

Each iteration traverses sequentially the 16 byte block (from the least significant byte up to the most significant byte) and performs four basic operations on each byte. The first two operations implement confusion and the last two implement diffusion:

step 1: Exclusive-OR the next byte of the block with the next byte of the *xorBu* field.

step 2: Replace the byte computed in step 1 by the byte in the substitution table (*substPermu*) indexed by it.

step 3: Modify the next byte in the block by exclusive-ORing it with the byte computed in step 2. When the end of the block is reached then the least significant byte of the block is considered to be the "next" byte.

step 4: Use the next byte of the *bombPermu* array to define a position in the block. Modify the byte in this position of the block by Exclusive-ORing it with the byte computed in step 2.

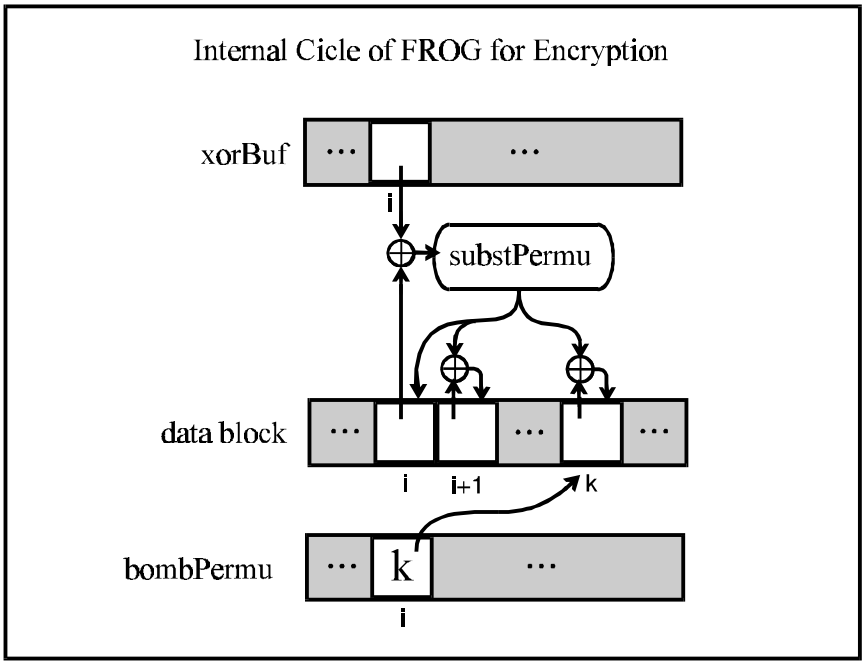

Fig. 2

The encryption process can be represented by the following pseudo code, in which "<=" denotes the assignment operation, and *blockSize* is the size in bytes of the block to be encrypted.

Procedure *FROGencrypt* (*plainText, cipherText, internKey*)
// convert plaintext into ciphertext
copy *plainText* **into** *cipherText*
for each of the eight records in *internKey* **do**
  begin
  //*xorBuf, substPermu, bombPermu* denote the fields of the current record
  for each byte in *cipherText* **do** [I <= 0 **to** *blockSize*-1]
    begin
    cipherText[I] <= *cipherText*[I] **XOR** *xorBu*[I]
    cipherText[I] <= *substPermu*[*cipherText*[I]]
    if I<*blockSize*-1
      then *cipherText*[I+1] <= *cipherText*[I+1] **XOR** *cipherText*[I]
      else *cipherText*[0] <= *cipherText*[0] **XOR** *cipherText*[I]
    K <= *bombPermu*[I]
    cipherText[K] <= *cipherText*[K] **XOR** *cipherText*[I]
    end
  end
End Procedure

The following example will clarify this very simple process. All values are in HEX starting with the least significant byte. First we define an internal key record with the following values (starting with the less significant byte indexed by zero):

```
xorBu: 05 f0 a3 ...
substPermu: a2 16 08 bb 03 f1 ...
bombPermu: 03 0f 00 ...
```

Let us now start encrypting the following plaintext block:
`00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0f`

and show the intermediate states of the block after each step:

Processing byte in position 0:

```
1. step: 05 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0f
2. step: f1 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0f
3. step: f1 f0 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0f
4. step: f1 f0 02 f2 04 05 06 07 08 09 0a 0b 0c 0d 0f
```

Processing byte in position 1:

```
1. step: f1 00 02 f2 04 05 06 07 08 09 0a 0b 0c 0d 0f
2. step: f1 a2 02 f2 04 05 06 07 08 09 0a 0b 0c 0d 0f
3. step: f1 a2 a0 f2 04 05 06 07 08 09 0a 0b 0c 0d 0f
4. step: f1 a2 a0 f2 04 05 06 07 08 09 0a 0b 0c 0d ad
```

Processing byte in position 2:

```
1. step: f1 a2 03 f2 04 05 06 07 08 09 0a 0b 0c 0d ad
2. step: f1 a2 bb f2 04 05 06 07 08 09 0a 0b 0c 0d ad
3. step: f1 a2 bb 49 04 05 06 07 08 09 0a 0b 0c 0d ad
4. step: 4a a2 bb 49 04 05 06 07 08 09 0a 0b 0c 0d ad
```

And so on. The confusion and diffusion process is fast and after only 3 iterations any statistical redundancies in the plaintext are obscured. Eight iterations were chosen as the basis for the FROG algorithm so as to ensure that the ciphertext generated has very good random properties.

Observe that the first two steps in the process are equivalent to using 16 different substitution tables for each of the 16 bytes in the block, which increases confusion achieved by these steps. The last two steps diffuse the result of this substitution process throughout the block. Two such operations were chosen so as to achieve an exponential speed of diffusion.

Notice that all the steps in the process are simple invertible operations. The decryption process traverses the internal key and the ciphertext block in the opposite direction, "undoing" all primitive operations performed during the encryption process, and thus recovering the plaintext. The internal key used in the decryption process is identical, except that all substitution tables (fields *substPermu*) are replaced by their inverse:

Procedure *FROGdecrypt* (*cipherText*, *plainText*, *internKey*)
// convert ciphertext into plaintext
copy *cipherText* **into** *plainText*
for each of the eight records in *internKey* **traversed in opposite direction do**
  begin
  //*xorBuf, substPermu, bombPermu* denote the fields of the current record
  for each byte in *plainText* **do** [I <= *blockSize*-1 **down to** 0]
    begin
    K <= *bombPermu*[I]
    plainText[K] <= *plainText*[K] **XOR** *plainText*[I]
    if I<*blockSize*-1
      then *plainText*[I+1] <= *plainText*[I+1] **XOR** *plainText*[I]
      else *plainText*[0] <= *plainText*[0] **XOR** *plainText*[I]
     plainText[I] <= *substPermu*[*plainText*[I]]
     plainText[I] <= *plainText*[I] **XOR** *xorBu*[I]
    end
  end
End Procedure

The *bompPermu* field is used to rapidly diffuse any change in the input throughout the block. In order to speed up as much as possible this diffusion process the *bombPermu* field should satisfy three conditions:

a). It must be a permutation, i.e. each of the block positions should be pointed at least once. In this way all positions in the block are exclusive OR-ed (bombed) with values that come from the substitution table indexed by another byte.

b). Second, the permutation should have a cycle length equal to the size of the block (16 in this case), that is to say if one regards the permutation of N numbers as a pointer chain, all N positions should be traversed before cycling back into the initial position. This is normally not the case with random permutations. For example, if we take a permutation of length 5, "4,0,3,2,1" and start at the first element (a 4 in position 0) we cycle back after only 3 transitions: 4 -> 1 -> 0 ->4. An example of a permutation with cycle length 5 is: "3,4,0,1,2". The rationale here is that if one uses a permutation with a cycle length smaller than the length of the block then the "bombing" effect is limited within sub-groups of elements in the block.

c). An element in the permutation must never point to the next element, i.e. the i-th element of the array must never contain the value i+1. In this way one guarantees the fourth step in the process (as described above) will not invert the diffusing effect of the third step.

It is interesting to note that the most important condition is the third one. A version of FROG that completely omits the implementation of the first and second conditions works almost as well as the standard version, and is simpler to implement.

Section B.1.3. describes the algorithm that generates permutations required by the first condition , and section B.1.4. describes the algorithms used to implement the second and third conditions. Section B.7 about possible future developments of FROG also refers to these conditions.

Observe that the block size is not a limiting factor for the operation of the FROG algorithm. For example, if we want to use a block size of 17 bytes, then the algorithm would still work as long as we reserve 17 arbitrary bytes for the *xorBu* array, and 17 bytes with values between 0 and 16 for the *bombPermu* array.

B.1.2 Key Setup

FROG computes the internal key as a function of the user key. As explained above each of the eight iterations uses a record that has 16 bytes for the *xorBuf* field, 256 bytes for the *substPermu* field and 16 bytes for the *bombPermu* field, so therefore the internal key contains 2,304 bytes. The Key Setup process of FROG is recursive: first it builds a simple internal key, then it uses FROG encryption in CBC (Cipher Block Chaining) mode to produce the definitive internal key. Figure 3 illustrates this process:

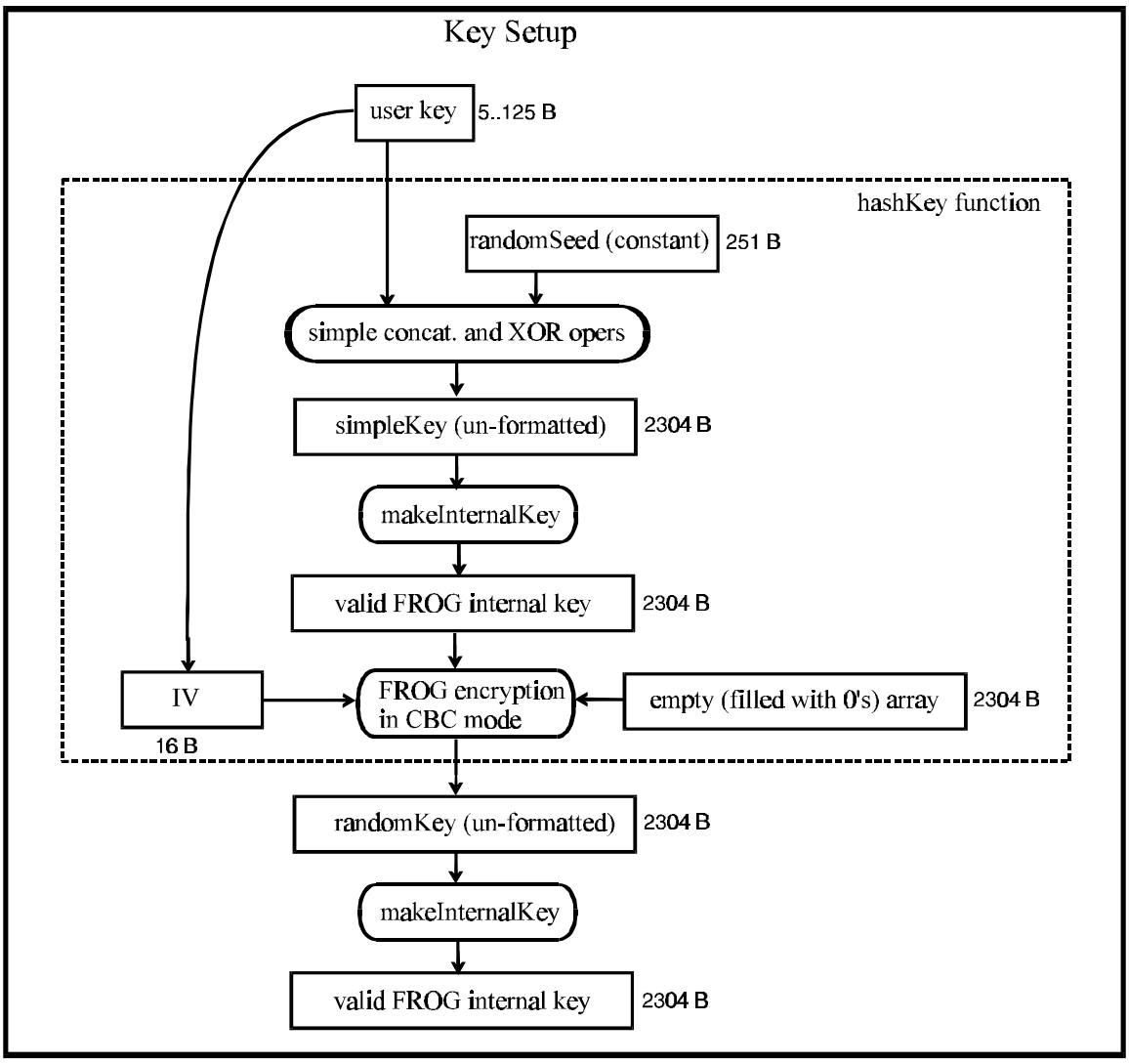

Fig. 3

As you can see in Fig 3, the large internal key is produced recursively, using the following two algorithms:

The first algorithm (function *makeInternalKey*) takes an unstructured, arbitrary array of 2,304 bytes and transforms it into a valid internal key. Specifically, it computes the permutation arrays *substPermu* and *bombPermu* for each of the eight records. To do this an algorithm is used (function *makePermutation* described in section B.1.3) that takes an array of N arbitrary bytes and returns a permutation with values 0 to N-1. If the internal key is to be used for decryption, then *substPermu* is inverted. The field *bombPermu* is further validated (see section B.1.4) in order to fulfill the special conditions described in the previous section.

The *makeInternalKey* algorithm in the key creation process can be represented by the following pseudo code:

Procedure *makeInternalKey* **(***internKey***)**
// convert *internKey* into a valid FROG internal key
  for each of the eight records in *internKey* **do**
  begin // *xorBuf, substPermu, bombPermu* denote the current records' fields
  makePermutation of 256 bytes (*substPermu*)
  if internal key is for decryption **then** *invert(substPermu)*
  makePermutation of blockSize bytes (*bombPermu*)
  Validate(bombPermu)
  end
End Procedure

The functions *makePermutation* and *Validate* are defined in sections B.1.3 and B.1.4 respectively.

The second algorithm (function *hashKey*) takes a key (with size between 5 and 125 bytes) and produces a 2,304 bytes long array (called *randomKey*) without loosing any entropy and filling it with data which statistically appears to be random. This is done in three steps as follows:

Step 1. A 2,304 bytes long array (called *simpleKey*) is filled with data that depends on both the user key and an internal constant (called *randomSeed*) which is filled with 251 true random bytes. s*impleKey* is first filled sequentially with copies of the user key, and then copies of the *randomSeed* are sequentially exclusive-ORed on top of this. Any trailing bytes are ignored. The resulting array is then processed by the function *makeInternalKey* and transformed into a valid internal key for FROG.

Step 2. An IV (Initialization Vector) is created using the less significant 16 bytes of the user key (if the key has less then 16 bytes, then the rest is filled with zeros). Then a value corresponding to the length of the user key in bytes (e.g. 16, 24 or 32 correspond to the 128-, 192- and 256-bit key sizes) is exclusive-ORed with the least significant byte of the IV (this is to ensure that different sizes of user key always produce different ciphertexts).

Step 3. FROG encryption function is then called to encrypt a zero filled array of 2,304 bytes in CBC (Cipher Block Chaining) mode using the internal key computed in step (a) and the IV computed in step (b). The result of this encryption process is an array (called *randomKey*) with very good statistical randomness.

Finally, this array is processed by the function *makeInternalKey* and transformed into the valid internal key. This completes the creation of the internal key which is then used to drive the encryption or decryption process.

The *hashKey* algorithm can be represented by the following pseudo code:

Procedure *hashKey* (*userKey, keyLen, randomKey*)
// hash *userKey* into a *randomKey* array of *internalKeySize* bytes (normally 2,304)
// *keyLen* is the user key's length in bytes
//
// Step a: create *simpleKey*
//
declare *simpleKey* array of *internalKeySize* bytes
S <= 0
K <= 0
for each byte in *simpleKey* **do** [I <= 0 to *internalKeySize*-1]
  begin
  simpleKey[I] <= *randomSeed*[S] XOR user*Key*[K]
  if S < 250 **then** S <= S+1 **else** S <= 0
  if K < *keyLen*-1 **then** K <= K+1 **else** K <= 0
  end
// convert *simpleKey* into valid FROG internal key
makeInternalKey for encryption (simpleKey)
//
// Step b: create *buffer* to be used subsequently as IV
//
declare *buffer* array of *blockSize* bytes and initialize with zeros
last <= *keyLen*-1
if *last* greater or equal *blockSize* **then** *last* <= *blockSize*-1
for I <= 0 **to** *last* **do** *buffer*[I] <= *buffer*[I] XOR *userKey*[I]
buffer[0] <= *buffer*[0] XOR *keyLen*
//
// Step c: call FROG encryption in CBC mode and produce *randomKey*
//
I <= 0
repeat
  begin
  FROGencrypt (*buffer*, *simpleKey*, *buffer*)
  size <= internalKeySize - I
  if *size* > *blockSize* **then** *size* <= *blockSize*
  copy *size* bytes of *buffer* into *randomKey*[I]
  I <= I+*size*
  end
until I equal *internalKeySize*
End Procedure

The 251 bytes of *randomSeed* is the only constant table used in FROG. Its definition is based on the initial sequence of numbers included in "A Million Random Digits" published by the RAND corporation in 1955. These are listed in groups of five decimal digits. *randomSeed* is initialized with the modulo 256 value of the numbers represented by the first 251 groups in this table. For example, the first group is 10097, which modulo 256 equals 113, the second group is 32533 which modulo 256 equals 21, etc. Here are the values (in decimal notation) of the *randomSeed* table, starting with the least significant byte:

```
113, 21,232, 18,113, 92, 63,157,124,193,166,197,126, 56,229,229,
156,162, 54, 17,230, 89,189, 87,169,  0, 81,204,  8, 70,203,225,
160, 59,167,189,100,157, 84, 11,  7,130, 29, 51, 32, 45,135,237,
139, 33, 17,221, 24, 50, 89, 74, 21,205,191,242, 84, 53,  3,230,
231,118, 15, 15,107,  4, 21, 34,  3,156, 57, 66, 93,255,191,  3,
 85,135,205,200,185,204, 52, 37, 35, 24, 68,185,201, 10,224,234,
  7,120,201,115,216,103, 57,255, 93,110, 42,249, 68, 14, 29, 55,
128, 84, 37,152,221,137, 39, 11,252, 50,144, 35,178,190, 43,162,
103,249,109,  8,235, 33,158,111,252,205,169, 54, 10, 20,221,201,
178,224, 89,184,182, 65,201, 10, 60,  6,191,174, 79, 98, 26,160,
252, 51, 63, 79,  6,102,123,173, 49,  3,110,233, 90,158,228,210,
209,237, 30, 95, 28,179,204,220, 72,163, 77,166,192, 98,165, 25,
145,162, 91,212, 41,230,110,  6,107,187,127, 38, 82, 98, 30, 67,
225, 80,208,134, 60,250,153, 87,148, 60, 66,165, 72, 29,165, 82,
211,207,  0,177,206, 13,  6, 14, 92,248, 60,201,132, 95, 35,215,
118,177,121,180, 27, 83,131, 26, 39, 46, 12
```

B.1.3 The Permutation Generation Algorithm

makePermutation is a general purpose algorithm that takes an input array of bytes and returns a permutation of the same length. In this context permutation means an array of length N that holds all values between 0 and N-1 without any repetition.

The algorithm works as follows:

An array called "*use*" is initialized sequentially with all values between 0 and N-1. The permutation is created sequentially one byte at a time taking bytes from the "*use*" array. When a byte is used it is removed from the "*use*" array and the length of the array is decreased by 1. In this way it is guaranteed that the same value will never used twice in the permutation. The values of the input array are used to compute an index into the "*use*" array. When computing the i-th element of the permutation this index is computed by adding the previous index used with the i-th byte of the input array:

```
index_i = ( index_(i-1) + input[i] ) mod (length of "use" array)
```

The modulo operation is necessary to guarantee that the index computed will point to a valid position. For the first iteration, the "previous index" is initialized to zero.

The function for creating permutations can be represented by the following pseudo code:

Procedure *makePermutation* (*input, lastElem)*
// inputs an array of *latsElem*+1 bytes and converts it into a permutation
declare array *use* of *lastElem*+1 bytes
for I <= 0 **to** *lastElem* **do** use[I] <= I
last <= *lastElem*
index <= 0
for all bytes of *input* [I <= 0 **to** lastElem-1] **do**
  begin
  index <= (*index*+*input*[I]) mod (*last*+1)
  input[I] <= *use*[*index*]
  if *index* < *last* **then** remove element pointed by *index* from the *use* array
  last <= *last* - 1
  if *index* > *last* **then** *index* <= 0
 end
input[*lastElem*] <= *use*[0]
End Procedure

The following example shows the conversion of the input array (101,34,61,208) into a permutation (numbers are given in decimal notation).

In the first iteration

```
index = (0 + 101) mod 4 = 1.
```

In the second iteration

```
index = (1 + 34) mod 3 = 2.
```

And so on:

```
input array          index       use array   value

101  34  61 208        1         0 1 2 3        1
  1  34  61 208        2         0 2 3          3
  1   3  61 208        1         0 2            2
  1   3   2 208        0         0              0
  1   3   2   0
```

The resulting permutation is (1,3,2,0).

B.1.4 The Implementation of the *bombPermu* Restrictions

The first restriction on the permutation *bombPermu* is that it must have a cycle length equal to the block size (16 in this case). If the permutation is regarded as a pointer list then it can interpreted as a single closed linked list or as a group of closed linked lists. The task at hand is to create a permutation that correspond to single closed linked list. To achieve this the algorithm proceeds as follows: starting at the first element it traverses the list. If it arrives back to the first element before having used up all values of the permutation then it looks for the fist position in the permutation that it has not yet traversed. This position holds a value that can serve as the start of another linked list. The algorithm then merges these two lists into one. This process is repeated until all the elements have been traversed, that is to say until only one list remains.

For example, the permutation 3,0,5,1,2,4 can be interpreted as to two closed linked lists, as follows: the element in position 0 is 3 which indexes the position in the permutation which holds a 1, which indexes the position in the permutation that holds a 0, which indexes the starting position. In this way we get the first closed linked list: 3->1->0->3. The last position traversed is position 1 which holds the value 0. The fist position not yet traversed is the position 2 that holds a 5. This value forms the start of the second linked list which is 5->4->2->5. These two lists are now merged together in a two step process. First change the value in position 1 from to 0 to 2. Now we have an incomplete list: 3->1->2->5->4->2->5. To close this list the value in position 4 is changed from 2 to 0: 3->1->2->5->4->2->0->3. The resulting permutation 3,2,5,1,0,4 has a cycle length of 6 so we have finished. Fig 4 clarifies this operation:

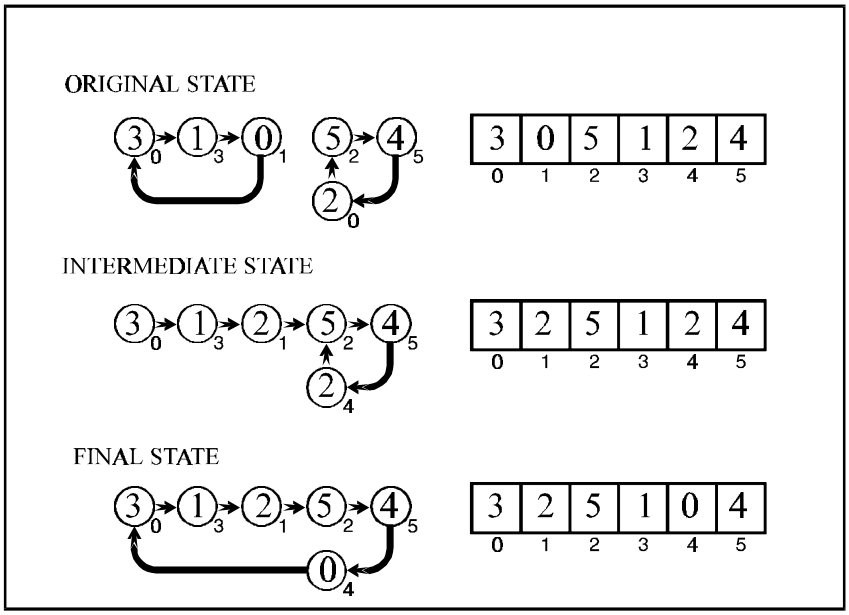

Fig. 4

The second restriction on the permutation *bombPermu* is that no element should point to the next position in the list. This requirement is fulfilled by traversing the permutation array from the least significant byte to the most significant byte and testing whether P(i)=(i+1) mod (block size). If this is so P(i) is simply redefined to point one position further to the right P(i)= (i+2) mod (block size).

This last process is very fast but, unfortunately, it will normally destroy the carefully crafted permutation. Even so FROG retains its excellent diffusion speed which is the goal of both these processes.

The function for applying the *bombPermu* conditions can be represented by the following pseudo code:

```
Procedure Validate (bombPermu)
// make certain that bombPermu has a cycle length of  blockSize
declare used array of blockSize bytes
fill array used with false
index<=0
for all but last element in bombPerm [ I <= 0 to blockSize-2 ] do
   begin
   if bombPermu[index]=0 then // short cycle found
      begin
      K <=  index
      repeat K<=(K+1) mod blockSize until NOT used[k] //first position not yet traversed
      bombPermu[index] <= K
       L <= K
       while bombPermu[L] not equal K do L <= BombPermu[L] //find who points to K
       bombPermu[L] <= 0 // now cycles are merged
       end
    used[index] <= true
    index <= BombPermu[index] // continue searching
    end
// now make certain that no element of bombpermu  points to next element
for all elements in bombPermu [I<=0 to blockSize-1] do
   if bombPermu[I] equal (I+1) mod  blockSize
     then bombPermu[I] <= (I+2) mod  blockSize
End Procedure
```

B.2 Estimate of the Computational Efficiency

The following data is based on the optimized ANSI C version of FROG as submitted in the present package. When running on the NIST AES analysis platform (a 200 MHz Pentium PC) it encrypts 1.3 Mbytes per second and decrypts 1.7 Mbytes per second. The key setup takes a relatively long 10 msec. An implementation that includes 16 bit 8086 assembler code for the inner cycle encrypts over 2.2 Mbytes per second on the same platform. An implementation that uses Pentium specific instruction set should encrypt and decrypt over 3.0 Mbytes per second (approximately 50 cycles per byte).

FROG's inner cycle processes each 16 byte block in a sequential manner. Even so a hardware implementation of FROG should be able to implement one iteration in approximately 20 clock cycles. A pipelined IC would then be able to process 0.8 bytes per cycle, e.g. a 200 MHz chip would process 160 Mbytes per second (1.3 Gbps).

The size of the user key (128, 196, or 256 bits) does not affect the speed of the algorithm because the user key is used only to initialize the internal key during the setup process. Also observe that the same computational effort must invested in key setup and in key change. There are no tradeoffs between speed and memory in the implementation of FROG.

B.2.1. On a Pentium platform

Program: ANCI C optimized version included in this submission.

Platform: 200 MHz Pentium PC, 64 MB of RAM, Windows 95

Speed estimate in clock cycles

Key/Block size	128/128	192/128	256/128
Encrypt one data block	2,600	2,600	2,600
Decrypt one data block	1,980	1,980	1,980
Key setup	1,960,000	1,960,000	1,960,000
Algorithm setup	125	125	125
Key change	1,960,000	1,960,000	1,960,000

In FROG encryption and decryption represent identical workload. The fact that a Pentium decrypts faster than it encrypts is related to internal CPU optimizations.

B.2.2. On a 8-bit computer

```
Program: Z80 assembly for internal cycle.

Platform: 2.5 MHz Z80, 64 kB of memory running under CP/M.

Speed estimate in clock cycles

Key/Block size               128/128    192/128    256/128

Encrypt one data block        17,900     17,900     17,900
Decrypt one data block        17,900     17,900     17,900
Key setup                 15,000,000 15,000,000 15,000,000
Algorithm setup                  200        200        200
Key change                15,000,000 15,000,000 15,000,000
```

B.3. Known Answer Tests and Monte Carlo Tests

These are submitted as text files on a separate diskette. The hexadecimal numbers that appear in the tests use the convention that the leftmost character represents the most significant hexadecimal digit, i.e. the 4 most significant bits. For example the decimal number 1000 is represented by 3e8.

The NIST specifications for the test were interpreted by taking bit 1 as the most significant bit and bit 128 as the least significant bit of a 128 bit block.

B.4. Expected Strength of the Algorithm

No attacks more efficient than exhaustive key search are known. The expected workfactor of the algorithm is therefore O(2^(N-1)) where N is number of bits in the user key, i.e. O(2^127) for a 128 bit key, O(2^191) for a 192 bit key and O(2^255) for a 256 bit key.

B.5. Algorithm Analysis

The design of FROG was guided by the goal to implement a cipher which is as simple as possible algorithmically but as complex as possible to model mathematically. The first characteristic produces efficiency and design transparency; the second produces strength, because in the absence of a mathematical model, the attacker can only search for statistical weaknesses in the algorithm which in the case of FROG are believed to be absent.

Most internal transformations, including the substitution tables and the tables that guide the diffusion process depend on the internal key and are therefore unknown. The internal

key is itself built by calling the FROG algorithm recursively and depends therefore on the user key in a very complex manner. No known or chosen plaintext attacks are expected to work.

No weak keys or equivalent keys or keys with complementation properties are known and none are expected to exist. There are no restrictions at all on key selection.

The simplicity of the structure of FROG and the absence of any constant tables preclude the possibility of a trap-door. The code in FROG has clear goals: the inner cycle implements two primitive operations for confusion and two more for diffusion. The internal key is constructed recursively and the structure chosen for the internal key obeys either design requirements or were chosen to accelerate the diffusion process. The only rather arbitrary code in FROG is the creation of the simple internal key (*simpleKey*) during the key setup process, in which as many copies as needed of the user key are just linked together. This happens to be the simplest way to create this key. However, a more significant argument is that the way *simpleKey* is initialized is not important to the strength of the cipher; in fact it can be changed deliberately so as to produce non-standard versions of FROG (the advantage of doing this is explained in B.6.)

One constant table (*randomSeed*) is included in FROG but its definition is based on random data that have been publicly known for the last 40 years (the RAND tables) and therefore no trapdoor can be hidden in this table. Moreover, the values in this table do not seriously affect the strength of the algorithm, you can fill this table with any random values and the resulting variant of FROG will work well. The purpose of this table is to increase the flexibility of the algorithm because it can be used to implement a master key (also see B.6).

There are no published materials relating to FROG.

B.6. Advantages and Limitations

FROG displays several positive characteristics:

- Simplicity of design
- Speed of operation
- Flexibility of key and block sizes
- Variability (it can easily be customized)

These advantages are described below.

Its only limitation that may be significant for some applications is its relatively long setup time (see section B. 7 for ideas on how the setup time might be reduced).

B.6.A. Different Uses

FROG can be used to implement a stream cipher with no loss of security.

FROG can be used to implement a MAC (Message Authentication Code), for example by encrypting a message in CFB (Cipher Feedback Mode). FROG's flexibility to allow for any plaintext sizes between 64 and 1024 bits can be helpful in this case, for example if you need an 80 bit MAC for an 800 bit long message. There are no known weaknesses when FROG is used to implement a hash algorithm or a MAC but, no tests have so far been conducted to check this experimentally.

During the design of FROG, powerful statistical analysis tools were used to determine whether FROG produces good pseudo-random output even when implemented with low iteration counts. FROG can certainly be used as a pseudo-random number generator, for example by initializing a plaintext with the generator's seed, exclusive OR-ing this plaintext with a sequential counter, and encrypting the resulting block to produce the next block of pseudorandom bits.

No other limitations of FROG relating to different potential uses of the cipher are known.

B.6.B. Implementation of FROG in Various Environments

FROG is very well suited for implementation on 8 bit processors (such as the ones included in smart cards or other embedded applications). It uses only byte instructions and can be implemented by a short program.

FROG is not very well suited for a pure IC (Integrated Circuit) hardware implementation, as its design flows from software engineering concepts. It certainly can be implemented in an ASIC (Application Specific Integrated Circuit), however a gate count cannot be estimated at this time and might be high.

A better solution, would be to implement a simple ASIC for the encryption and decryption processes only and to implement the key setup by software for an embedded processor. The algorithm's inner cycle operations on each byte of the block are so simple that they can be implemented in about 7 assembly language instructions using the 8086 assembly set. If FROG becomes the AES, future microprocessors designers might want to consider implementing these as a single machine instruction. This would allow pure software implementations of FROG to reach speeds of about 10 CPU cycles per byte.

ATM, HDTV and B-ISDN are fast communication technologies and in this context FROG's speed is an advantage. However, FROG is relatively slow during the key setup process which could limit its usefulness in some situations. The time needed for the key setup process is proportional to the encryption speed, because typically over one hundred encryptions must be executed to create the internal key. If the encryption speed is further optimized (for example, by implementing the internal cycle in hardware), then the key setup speed will increase very significantly. In some cases, such as smart-card applications or pre-paid cellular phones, the large internal key could be pre-computed and pre-loaded. FROG needs very little memory to operate efficiently, typically less than 2,500 bytes if a pre-loaded internal key is used and less than 5,000 bytes when key setup is included in the algorithm.

No other advantages or limitations concerning the implementation of FROG for use with ATM, HDTV or B-ISDN are known at present.

B.6.C. Other Key and Block Lengths

In this context FROG is extremely flexible. The reference code included here allows for key lengths between 40 and 1000 bits in 8 bit (1 byte) increments. Even though the code submitted sets the size block to a constant 16 bytes (or 128 bits which is the minimum requirement), this constant (*CODE_SIZE*) can be modified and set to any value between 8 and 128 (bytes) and the code can be recompiled to produce instances of FROG that encrypt blocks from 64 bits up to 1024 bits.

It is a simple programming task to produce a source code that allows for variable key and block sizes with up to 16,254 combinations of key and block sizes. It is important to note that a larger block size results in a larger internal key and thus longer setup times. The speed per byte of the encryption or decryption process does not depend significantly on the block size.

A 256 bit (32 bytes) key length is sufficient for defense against exhaustive key searching even under the most extreme assumptions. The fact that FROG allows up to 125 byte long keys means that the application can directly input the user's pass-word or passphrase without any previous pre-processing.

Variable block sizes can be extremely useful sometimes. For example, in the implementation of a confidential, random-access data base. The 1 byte increment of the block size permitted by FROG, means that such a data base can be encrypted without increasing the size of its records, resulting in higher speed encryption and greater ease of integration with existing applications. Another related example would be to encrypt only specific fields (columns) of a data base, regardless of their size.

Finally, it should be noted that FROG can implement directly the 64 bit block size and 112 bit key size of 3DES.

B.6.D. Other Advantages

There is no speed penalty when using FROG with larger user key sizes (neither the key setup time nor the encryption/decryption speed depend on the user key size).

The relative complexity of the key setup process can be useful as a defense mechanism against dictionary attacks. If this is deemed useful then a version of FROG can be produced that executes more than one CBC encryption pass in the key setup process, which would produce arbitrarily long (variable) setup times.

A better defense against dictionary attacks is to modify the standard values of the *randomSeed* table. This can be done by initializing this table with a master key. Normally the master key would be administered centrally and end users would not need not know its value. In this case an attacker must find a way to obtain the master key before being able to mount a dictionary attack. Even in the case where the master key is simply stored on the end user's hard disk, the attacker's workload is significantly increased.

A defense that will work with all imaginable attack methods (except ciphertext only attacks which are exceedingly weak) is to use non standard versions of FROG where the actual code is different from the reference version. Then the attacker will have to find a way to steal a working copy of the cipher, disassemble it and then cryptanalyze it. This is clearly an enormously more difficult proposition than cryptanalyzing a publicly available standard version of the cipher. This kind of defense may be appropriate for applications that work within the confines of an organization, or applications for very sensitive point to point communications, etc.

Customization of the algorithm can be achieved in the following manner: the FROG cipher contains a small part that initializes the initial version of the internal key (*simpleKey*) as a function of both the user key and the *randomSeed* (or master key). This code fragment can be freely and easily be changed, even by a programmer who is not specialized in encryption technology, without affecting the strength of the algorithm. The only requirements are: a) the entropy of the user key is not lost in the process (which means, simply, that a different user key should always produce a different *simpleKey*); and b) that the user key contents are well spread out through the resulting initial internal key. In this way, an unlimited number of non standard versions of FROG can be produced, which are stronger than the standard version as long as the attacker does not gain access to them, and are as strong as the standard version in the worst case.

B.7. Possible Future Developments

The simpler a cipher is the easier it is to detect a design flaw or a trap door. In this context it is considered important to investigate whether a version of FROG that omits the implementation of the first two conditions for the *bombPermu* array is not preferable even though it diffuses information a little slower. Changing the algorithm in this way would also have the effect of reducing the key setup time.

The key setup process can also be speeded up by using four iterations in the internal encryption process instead of eight (which are used to encrypt user data). This would practically double the speed of the setup process, and would probably not reduce the strength of the cipher.

Yet another possibility for speeding up the key setup process (at least on a Pentium processor) would be to use decrypts instead of encrypts in the process used to generate the internal key.