2006 IEEE Congress on Evolutionary Computation
Sheraton Vancouver Wall Centre Hotel, Vancouver, BC, Canada
July 16-21, 2006

Wheedham: An Automatically Designed Block Cipher by means of Genetic Programming

Julio C. Hernandez-Castro, *Member, IEEE*, Juan M. Estevez-Tapiador, *Member, IEEE*, Arturo Ribagorda-Garnacho, and Benjamin Ramos-Alvarez

***Abstract*— In this work, we present a general scheme for the design of block ciphers by means of Genetic Programming. In this vein, we try to evolve highly nonlinear and efficient functions to be used for the key expansion and the F-function of a Feistel network. Following this scheme, we propose a new block cipher design called Wheedham, that operates on 512 bit blocks and keys of 256 bits, of which we offer its C code (directly translated from the GP Trees) and some preliminary security results.**

I. Introduction

A block cipher consists of two paired algorithms, one for encryption, E and another for decryption, E^{-1}, that operate over two inputs: a data block of size n bits and a key of size k bits, yielding an n-bit output block. For every fixed key k and message M, decryption is the inverse function of encryption:

$$E_k^{-1}(E_k(M)) = M$$

For each key k, E_k is a permutation (a bijective mapping) over the set of input blocks. Each key selects one permutation among the $2n!$ possibilities. Typical key sizes in the past have been 40, 56, and 64 bits. Today, 128 bits is normally taken as the minimum key length needed to prevent brute force attacks. Typical block sizes have also been increased from 64 to 128 bits to ensure an adequate security level.

Most block ciphers are constructed by repeatedly applying a simpler function. Ciphers using this approach are known as iterated block ciphers. Each iteration is generally termed a round, and the repeated function is called the round function; Most modern block ciphers use between 8 to 32 rounds.

A. State of the art

Very few works have considered the use of Evolutionary Computation paradigms for developing cryptographic primitives. Among these, we should mention the pioneer article of [19], where the authors propose a new encryption algorithm relying on a reversible cellular automata (CA). CA's are very interesting candidates for being used as building blocks of new cryptographic primitives due to their output complexity and parallel nature ([20], [23]). As a result, they have been extensively used in the design of hash functions ([21],[22]) and stream ciphers [24]. However, block ciphers have been relatively forgotten by the researchers in the area.

II. Theoretical Background

For completeness and readability, we first provide a brief introduction to the main concepts used in this work.

A. Feistel Networks and Block-Cipher Design

A Feistel network is a general structure invented by IBM cryptographer Horst Feistel, who introduced it in 1973 in the design of the block cipher Lucifer [1]. A large number of modern block ciphers are based on Feistel networks due to several reasons. First, the Feistel structure presents the advantage that encryption and decryption are very similar (requiring only a reversal of the key schedule), thus minimizing the size of the code and circuitry required to implement the cryptosystem. Examples of well-known block ciphers based on this structure are: DES [2], FEAL [3], GOST [4], LOKI [5], CAST [6], Blowfish [7], and RC5 [8], among others.

Feistel networks gained much popularity after the adoption of DES as an international standard, which is basically a modification (approved by the NSA) of the original Lucifer cipher. As a consequence, Feistel networks have been extensively studied and some important theoretical results regarding precise bounds for their security have been obtained. In particular, Michael Luby and Charles Rackoff proved that if the round function F is a cryptographically secure pseudorandom number generator, with K_i (a parameter derived from the key) used as the seed, then 3 rounds is sufficient to make the block cipher secure, while 4 rounds is sufficient to make the block cipher strongly secure (i.e. secure against chosen-ciphertext attacks) [10].

In a classical Feistel network half of the bits operate on the other half. As pointed out by Bruce Schneier and John Kelsey (see [11]), there is no inherent reason that this should be so. Despite further works have generalized this basic structure, in this work we will refer exclusively to the classical Feistel scheme.

1) Construction Details: One of the fundamental building blocks of a Feistel network is the F-function, usually known as the *round function*. This is a key-dependent mapping of an input block onto a output block:

Julio C. Hernandez-Castro is with the Computer Science Department, Carlos III University of Madrid, Leganes, Madrid 28911, Spain (phone: +34-91-624-8847; fax: +34-91-624-9129; email: jcesar@inf.uc3m.es).

Juan M. Estevez-Tapiador is with the Computer Science Department, Carlos III University of Madrid, Leganes, Madrid 28911, Spain (email: jestevez@inf.uc3m.es).

Arturo Ribagorda is with the Computer Science Department, Carlos III University of Madrid, Leganes, Madrid 28911, Spain (email: arturo@inf.uc3m.es).

Benjamin Ramos is with the Computer Science Department, Carlos III University of Madrid, Leganes, Madrid 28911, Spain (email: benja1@inf.uc3m.es).

0-7803-9487-9/06/$20.00/©2006 IEEE

$$F : \{0,1\}^{n/2} \times \{0,1\}^k \rightarrow \{0,1\}^{n/2}$$

Here, n is the size of the block. Thus, F takes $n/2$ bits of the block and k bits derived from the key (usually known as the *round subkey*) and produces an output block of length $n/2$ bits. F should be a highly nonlinear function, and the Feistel construction allows it even to be non-invertible.

The general scheme of a Feistel network consists of j rounds in which the same scheme is repeated. X_{i-1} is the input to the i-th round, and the output, X_i, serves as input for the next one. The basic operation of each round is as follows. The input block at round i is split into two equal pieces:

$$L_i = msb_{n/2}(X_i)$$
$$R_i = lsb_{n/2}(X_i)$$

where $lsb_u(x)$ and $msb_u(x)$ select the least significant and most significant u bits of x, respectively. In this way, $X_i = (L_i \| R_i)$. For encryption, the basic computation is the following:

$$L_i = R_{i-1}$$
$$R_i = L_{i-1} \oplus F(R_{i-1}, K_i)$$

where $\oplus$ indicates modulo-2 addition and K_i is the subkey for round i. For decryption, the same scheme can be applied to the ciphertext without being necessary to invert the round function F. The only difference is that the subkeys have to be used in reversal order:

$$R_{i-1} = L_i$$
$$L_{i-1} = R_i \oplus F(L_i, K_i)$$

The basic operation of a single round in a Feistel network is graphically illustrated in Figure 1.

The subkeys K_i, $i = 1, \ldots, j$ are obtained from applying a key schedule (also known as key expansion) algorithm to the input key K.

B. Genetic Programming

Genetic Programming is a stochastic population-based search method devised in 1992 by John R. Koza [12]. It is inspired in Genetic Algorithms, the main difference with them being the fact that in the later, chromosomes are used for encoding possible solutions to a problem, while GP evolves whole computer programs. Within the scope of Evolutionary Algorithms, it exists a main reason for using GP in this problem: A block cipher is, in essence, a computer program, so its size and structure are not defined in advance. Thus, finding a flexible codification that can fit a GA is a difficult problem. Genetic Programming, nevertheless, does not impose restrictions to the size or shape of evolved structures. An additional advantage of GP is that some domain knowledge can be injected by selecting the most relevant primitives, whereas other Machine Learning methods use a predefined, unmodifiable set (neurons in NN, attribute comparisons in ID3, etc.).

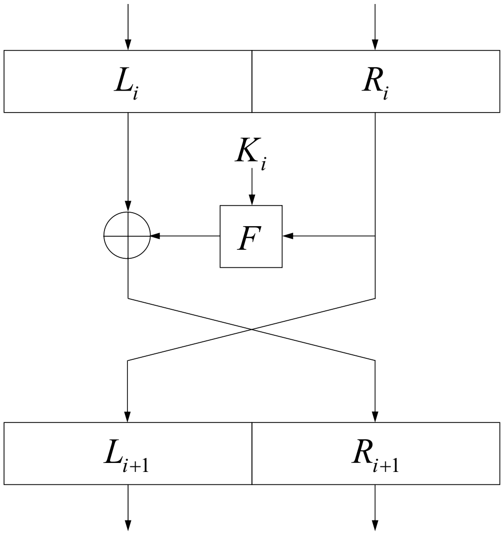

Fig. 1. Illustration of a round in a Feistel network

GP has three main elements:

- A population of individuals. In this case, the individuals codify computer programs or, alternatively, mathematical functions. They are usually represented as parse trees, made of functions (with arguments), and terminals. The initial population is made of randomly generated individuals.
- A fitness function, which is used to measure the goodness of the given computer program represented by the individual. Usually, this is done by executing the codified function over many different inputs, and analyzing its outputs.
- A set of genetic operators. In GP, the basic operators are reproduction, mutation, and crossover. Reproduction does not change the individual. Mutation changes a function, a terminal, or a complete subtree. The crossover operator exchanges two subtrees from two parent individuals, thereby combining characteristics from both of them into the offspring.

The GP algorithm starts a cycle consisting on fitness evaluation and application of the genetic operators, thus producing consecutive generations of populations of computer programs, until an ending condition is reached (generally, a given number of iterations or a maximum in the fitness function). In terms of classical search, GP is a kind of beam search, the heuristic being the fitness function. A typical GP implementation has many parameters to adjust, as the size of the population and the maximum number of generations. Additionally, every genetic operator has a given probability of being applied that should be adjusted.

C. The Avalanche Effect

Nonlinearity can be measured in a number of ways or, what is equivalent, has not a complete unique and satisfactory definition. Fortunately, this is of no concern to us, as we do not pretend to measure non-linearity, but a very specific mathematical property named avalanche effect. This property tries, to some extend, to reflect the intuitive idea of high-nonlinearity: a very small difference in the input producing a high change in the output, hence an avalanche of changes.

Mathematically, $F : 2^m \rightarrow 2^n$ has the avalanche effect if it holds that:

$$\forall x, y | H(x, y) = 1, \quad Average\bigg(H(F(x), F(y))\bigg) = \frac{n}{2}$$

So if F is to have the avalanche effect, the Hamming distance between the outputs of a random input vector and one generated by randomly flipping one of the bits should be, on average, $n/2$. That is, a minimum input change (one single bit) produces a maximum output change (half of the bits) on average.

This definition also tries to abstract the more general concept of output independence from the input (and thus our proposal and its applicability to the generation of block ciphers). Although it is clear that this independence is impossible to achieve (a given input vector always produces the same output), the ideal F will resemble a perfect random function where inputs and outputs are statistically unrelated. Any such F would have perfect avalanche effect, so it is natural to try to obtain such functions by optimizing the amount of avalanche. In fact, we will use an even more demanding property that has been called the Strict Avalanche Criterion [13] which, in particular, implies the Avalanche Effect, and that could be mathematically described as:

$$\forall x, y | H(x, y) = 1, \quad H(F(x), F(y)) \approx B\bigg(\frac{1}{2}, n\bigg)$$

It is interesting to note that this implies the avalanche effect, because the average of a Binomial distribution with parameters $1/2$ and n is $n/2$, and that the amount of proximity of a given distribution to another ($B(1/2, n)$ in this case) could be easily measured by means of a χ^2 goodness-of-fit test. That is exactly the procedure we will follow.

III. Implementation Issues

We have used the `lil-gp` genetic programming library [14] as the base for our system. `Lil-gp` provides the core of a GP toolkit, so the user only needs to adjust the parameters to fit his particular problem. In this section, we detail the changes needed in order to configure our system.

A. Function Set

Firstly, we need to define the set of functions: This is critical for our problem, as they are the building blocks of the functions we would obtain. Being efficiency one of the paramount objectives of our approach, it is natural to restrict the set of functions to include only very efficient operations, both easy to implement in hardware and software. Another, but minor, objective was to produce portable algorithms; so the inclusion of the basic binary operations such as **vrotd** (right rotation), **vroti** (left rotation), **xor** (addition mod 2), **or** (bitwise or), **not** (bitwise not), and **and** (bitwise and) are an obvious first step. Other operators as the **sum** (sum mod 2^{32}) are necessary in order to avoid linearity, being itself quite efficient.

The inclusion of the **mult** (multiplication mod 2^{32}) operator was not so easy to decide, because, depending on the particular implementations, the multiplication of two 32 bit values could cost up to fifty times more than an **xor** or an **and** operation (although this could happen in certain architectures, its nearly a worst case: 14 times [15] seems to be a more common value). In fact, we did not include it at first, but after extensively experimentation, we conclude that its inclusion was beneficial because, apart from improving non-linearity it at least doubled and sometimes tripled the amount of avalanche we were trying to maximize. That is the reason why we finally introduced it in the function set. However, there are many other cryptographic primitives that make an extensive use of multiplication, notably RC6 [9]

Similarly, after many experiments, we concluded that the functions **vroti** and **vrotd** were absolutely interchangeable and that using them at the same time was not necessary nor useful, so we arbitrarily decided to remove **vroti** and left **vrotd**. Anyway, with **vrotd** we have a similar problem than with **mult**: compared to other operators, in some architectures **vrotd** it is quite inefficient. So we tried to eliminate this operator and include the $\gg$ (regular right shift) instead. But we found that $\gg$ was not able of producing as much non-linearity as **vrotd**, and the efficiency gains of the obtained functions were not as good to ignore the loss of Avalanche Effect.

B. Terminal Set

The set of terminals in our case is relatively easy to establish. Firstly, it is mandatory for the key schedule algorithm to operate with the 256 bits of the key, in our case expressed as eight 32-bit integers. Secondly, the round function should accept as input the 256 bits of the different round subkeys and half the 512-bit input block. Thus, the terminals of the GP system will be represented by 8 32-bit unsigned integers (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7).

Finally, we included Ephemeral Random Constants (ERCs) for completing the terminal set. An ERC is a constant value that GP uses to generate better individuals (for a more detailed explanation on ERCs, see [12]). In our problem, ERCs are 32-bits random-values that can be included in the building blocks of the cipher as constants to operate with. The idea behind this operator was to provide a constant value that, independently from the input, could be used by the operators of the function, and idea suggested by [16].

C. Fitness Function

We have used two different fitness functions for the two main tasks to be accomplished for developing a block cipher following the Feistel scheme: finding a key schedule algorithm, and a round function. Our main idea was to make the key schedule more efficient than the round function, but complex and robust enough to avoid simple related-key attacks as those published against ciphers with simple key expansion mechanisms [17]. To achieve this, we used the following fitness function

$$Fitness = mean/\chi_c^2$$

where χ_c^2 is a corrected value of χ^2, which is calculated as follows:

$$\chi_c^2 = \chi^2 * 10^6$$

where:

$$\chi^2 = \sum_{h=0}^{h=32} \frac{(O_h - E_h)^2}{E_h^2}$$

and

$$E_k = 8192 * Pr\big(B(1/2, 32) = k\big)$$

So the fitness of every individual (key expansion algorithm) is calculated as follows: First, we use the Mersenne Twister generator [18] to generate eight 32-bit random values. Those values are assigned to (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7). The value over this input o_0 is stored. Then, we randomly flip one single bit of one of the eight input values, and we run again the key expansion algorithm, obtaining a new value o_1. Now, we compute and store the Hamming distance $H(o_0, o_1)$ between those two output values. This process is repeated a number of times (8192 was experimentally proved to be enough) and each time a Hamming distance among 0 and 32 is obtained and stored. For a perfect Avalanche Effect, the distribution of this Hamming distances should adjust to the theoretical Bernoulli probability distribution B(1/2, 32). Therefore, the fitness of each individual is calculated by using two different but related measures: first the mean of the calculated Hamming distances; and second, the chi-square (χ^2) statistic that measures the distance of the observed distribution of the Hamming distances from the theoretical Bernoulli probability distribution B(1/2, 32). Thus, our GP system tries to maximize the mean and minimize the χ^2 statistic in order to maximize the expression for the fitness function shown above.

On the other hand, for finding a good round function we used a similar but different fitness expression:

$$Fitness = 10^6/\chi^2$$

where the mean is not present, and thus we try to directly minimize the χ^2 statistic. The idea behind the use of a different fitness for the F function and key expansion is to maximize them according to related but not identical properties, thus minimizing the probability that a weakness in one of the two obtained functions extends to the other.

We should note that in both cases we are computing the value of the χ^2 statistic without the commonly used restriction of adding up only the values when $E_k > 5.0$, for amplifying the effect of a 'bad' output distribution, thus, the sensibility of our measure.

It was necessary to correct the χ^2 statistic because its values were much bigger (several orders of magnitude) than the values of the mean. Without this correction, the mean measure was negligible, and the fitness was guided only by the χ^2.

D. Tree Size Limitations

When using genetic programming approaches, it is necessary to put some limits to the depth and/or to the number of nodes the resulting trees could have. We tried various approaches here, both limiting the depth and not limiting the number of nodes, and vice versa. The best results where consistently obtained using this latter option. As we were interested in a fast key schedule function, we limited the number of nodes to 50. On the other hand, we allowed the round function to use up to 100 nodes for trying to assure a high degree of avalanche effect and robustness against differential and lineal cryptanalysis. We did not put a limit (other that the number of nodes itself) to the tree depth. This was a very important step for determining the overall efficiency of the resulting block cipher algorithm.

IV. Experimentation and Results

We ran 20 experiments with different seeds ($seed_i = (\pi * 100000)^i \pmod{1000000}$) for each of the two functions needed for completing the Feistel network. The results presented below were the best individuals generated over these 20 experiments, with a population size of 200 individuals, a crossover probability of 0.8, and an ending condition of reaching 1000 generations.

A. Key Schedule Algorithm

The tree corresponding to the best individual we found when searching for the key schedule algorithm is shown in Figure 2. This is an algorithm with an avalanche degree of 13.3083 (so when randomly flipping one input bit of the 256 bit input, the 32 bit output changes 13.3083 bits, on average).

B. Round Function

On the other hand, the tree corresponding to the best individual found when looking for the round function is also depicted in Figure 2. This individual codifies a round function that provokes an avalanche effect of 15.9774 (16 is the optimal value, on average) and a χ^2 statistic of 6.088, for a χ^2 probability distribution with 32 degrees of freedom.

Key Schedule Algorithm

```
=== BEST-OF-RUN ===
              generation: 731
                   nodes: 50
                   depth: 25
                    hits: 133083
TOP INDIVIDUAL:
-- #1 --
                    hits: 133083
             raw fitness: 380.1202
    standardized fitness: 380.1202
        adjusted fitness: 380.1202
TREE:
 (sum (sum a4 (mult a2 a7)) (mult (vrotd
 (sum (vrotd (vrotd (vrotd (vrotd (vrotd
 (vrotd (vrotd (vrotd (vrotd (sum (sum
 (sum (vrotd (vrotd (vrotd (vrotd (vrotd
 (sum (sum (sum a4 a7) (sum (sum a0 a6)
 a2)) a3)))))) a5) a1) a4)))))))))) (sum
 (sum (mult (mult a3 a5) a0) a6) a1)))
 b52ac753))
```

Round Function

```
=== BEST-OF-RUN ===
              generation: 507
                   nodes: 95
                   depth: 52
                    hits: 159774
TOP INDIVIDUAL:
-- #1 --
                    hits: 159774
             raw fitness: 164245.6997
    standardized fitness: 164245.6997
        adjusted fitness: 164245.6997
TREE:

 (sum a4 (xor (mult (sum a4 (vrotd (vrotd
 (mult (sum a4 (vrotd (vrotd (sum (vrotd a3)
 (vrotd (vrotd (vrotd (mult (sum a4 (vrotd
 (vrotd (vrotd (vrotd (vrotd (vrotd (vrotd
 (vrotd (sum a1 (sum 3f85c67d (xor (sum a4
 (sum a3 (vrotd (sum (xor a4 a2) (sum (vrotd
 (vrotd (vrotd (vrotd (vrotd (vrotd (vrotd
 (sum (vrotd (vrotd (vrotd (sum (xor a0 (xor
 (sum a4 (vrotd (sum (xor (mult (sum a5 a7)
 3f85c67d) a1) (xor a4 a2)))) (vrotd a3)))
 (mult a6 3f85c67d))))) a0)))))))) (mult (xor
 a0 a1) a2)))))) a1)))))))))))) 3f85c67d)))))
 ))) (mult 3f85c67d 3f85c67d))))) (mult
 3f85c67d 3f85c67d)) (vrotd a3)))
```

Fig. 2. Best individuals

V. Security Analysis

We have performed a preliminary security analysis of the Wheedham cipher, consisting of examining the statistical properties of its output over a low-entropy input. In our case, we have set the cipher key to zero, $k = (0, \ldots, 0)$, and the input block to a simple incremental counter:

$$X_i = (16i, 16i + 1, \ldots, 16i + 15)$$

Next we have tested the randomness of $E_{(0,\ldots,0)}(X_i)$. Following this scheme, we have generated a file of 323986 KB (316 MB) to be analyzed with three batteries of statistical tests, namely ENT [25], Diehard [26] and NIST [27]. The results obtained are presented in Tables I and II

Wheedham also passed the very demanding NIST statistical battery. Due to the huge amount of p-values generated, the report is not shown in this paper. It can be downloaded from `http://kolmogorov.seg.inf.uc3m.es/wtests`.

TABLE I

ENTROPY RESULTS OBTAINED WITH ENT

Entropy	7.999999 bits/byte
Compression Rate	0%
χ^2 Statistic	254.32 (50%)
Arithmetic Mean	127.4981
Monte Carlo π estimation	3.141257768 (0.01%)
Serial correlation coefficient	0.000096

TABLE II

TEST RESULTS OBTAINED WITH THE DIEHARD SUITE

Test	p-value(s)
Birthday Spacings	0.716586 0.137
GCD	0.468790 0.171352
Gorilla	0.221
Overlapping Permutations	0.5140 0.2616 0.4447 0.1479 0.1479 0.4794
Ranks of 31×31 and 32×32 Matrices	0.219 0.047
Ranks of 6×8 Matrices	0.829482
Monkey Tests on 20-bit Words	OK
Monkey Tests OPSO, OQSO, DNA	OK
Count the 1‘s in a Stream of Bytes	0.612116
Count the 1‘s in Specific Bytes	OK
Parking Lot Test	0.938111
Minimum Distance Test	0.619443
Random Spheres Test	0.843300
The Squeeze Test	0.129066
Overlapping Sums Test	0.086223
Runs Up and Down Test	0.263
The Craps Test	0.09819 0.41816 0.859600 0.583844
Overall p-value	**0.509349**

In the light of the results shown in the previous tables, we can conclude that the output successfully passed all the randomness tests, even when generated by only 8 Feistel rounds of the Wheedham cipher, and using an extremely low entropy input.

VI. Conclusions and Future Work

The results shown in the previous Section demonstrate that Genetic Programming can be successfully applied to design competitive block ciphers. In this line, Wheedham can be thought as an instance of a family of designs that can be explored with this paradigm.

Perhaps the most relevant aspect in this scheme is the appropriate selection of the fitness function. In the case of the core components of a block cipher, nonlinearity is clearly a paramount objective. In this work, we have opted to use the strict avalanche effect as the key property of the explored mathematical functions. Additionally, avalanche could be measured very efficiently, so it becomes an ideal component of fitness function that, by its own nature, should be evaluated thousands of times.

The generated block cipher has successfully passed several batteries of very demanding statistical tests. Although this does not ensures a certain security level for Wheedham, it guarantees that neither trivial weaknesses nor implementation bugs exist. Future work should include deeper analysis of the cipher, particularly against basic attacks, such as linear, differential or related-key cryptanalysis. We have tried to incorporate robustness against these attacks by construction in Wheedham, even though further testing and analysis is required. Additionally, speed tests should be performed to measure the exact efficiency of the proposed cipher. In particular, it could be interesting to compare Wheedham's efficiency against the current state-of-the-art, including AES-256.

APPENDIX: C SOURCE CODE

Next we provide a fully operative implementation of the Wheedham block cipher. We assume a 32-bits architecture, so variables typed as `unsigned long` represent unsigned integers of length 32 bits. The two main functions are `encryption(v, k, NRounds)` and `decryption(v, k, NRounds)`. The first one encrypts `v` (a data block of 512 bits) using as key `k` (256 bits) and applying `Nrounds` rounds. Despite in our preliminary analysis, Wheedham has prove to be strong enough with 8 rounds, *we strongly recommend to use at least 16 rounds* to ensure an adequate security margin. Function `decryption(v, k, NRounds)` takes ciphered block `v` and key `k`, and returns the original data block.

```
/***********************************************/
/* Right Shift */
/***********************************************/
unsigned long rshift (unsigned long a, int t)
{
  unsigned long tmp;
  int i;

  for (i=0; i<t; i++) {
   tmp = a>>1;
   if (a & 0x00000001)
      tmp = tmp | 0x80000000;
   else
      tmp = tmp & 0x7FFFFFFF;
  }

  return(tmp);
}

/***********************************************/
/* Key Schedule Algorithm */
/***********************************************/
unsigned long key_schedule (
               unsigned long a0, unsigned long a1,
               unsigned long a2, unsigned long a3,
               unsigned long a4, unsigned long a5,
               unsigned long a6, unsigned long a7)
{
  unsigned long T1, T2, T3, T4;

  T1 = a3*a5*a0 + a6 + a1;
  T2 = a4 + a7;
  T3 = a0 + a6 + a2;

  T4 = rshift(T2 + T3 + T4 + a3, 5) + a5 + a1 + a4;
  T4 = (rshift(rshift(T4, 9) + T1, 1)*0xB52AC753)
       + a4 + a2*a7;

  return(T4);
}

/***********************************************/
/* Round Function */
/***********************************************/
unsigned long round_function (
               unsigned long a0, unsigned long a1,
               unsigned long a2, unsigned long a3,
               unsigned long a4, unsigned long a5,
               unsigned long a6, unsigned long a7)
{
  unsigned long T1;

  T1 = rshift(a1 ^ ((a5 + a7) * 0x3F85C67D) +
       (a2 ^ a4), 1);
  T1 = rshift((a0 ^ ((T1 + a4) ^ rshift(a3, 1)))
       + (a6 * 0x3F85C67D), 3);
  T1 = rshift(T1 + a0, 7);
  T1 = rshift(T1 + a2 * (a0 ^ a1) + (a2 ^ a4), 1);
  T1 = rshift(((T1 + a3 + a4) ^ a1) + 0x3F85C67D
       + a1, 8);
  T1 = rshift((T1 + a4) * 0x3F85C67D, 3);
  T1 = rshift(T1 + rshift(a3, 1), 2);
  T1 = rshift((T1 + a4) * 0x5DC79909, 2);
  T1 = a4 + (((T1 + a4) * 0x5DC79909) ^
       rshift(a3, 1));

  return(T1);
}

/***********************************************/
/* Block Encryption */
/***********************************************/

void encrypt(unsigned long* v, unsigned long* k,
             int Nrounds)
{

   unsigned long subkey[8], sum[8];
   int i, t, j;

   subkey[7]=0;

   for(i=0; i<Nrounds; i++) {
    /* Subkey generation for this round */
    for (t=0; t<8; t++)
     subkey[t] = key_schedule(k[0]+subkey[(t+7)%8],
                              k[1]+subkey[(t+7)%8],
                              k[2]+subkey[(t+7)%8],
                              k[3]+subkey[(t+7)%8],
                              k[4]+subkey[(t+7)%8],
                              k[5]+subkey[(t+7)%8],
                              k[6]+subkey[(t+7)%8],
                              k[7]+subkey[(t+7)%8]);

    /* Application of the F-function */
    sum[7] = 0;
```

```
     for (t=0; t<8; t++)
       sum[t] = round_function(
                         v[8]+subkey[0]+sum[(t+7)%8],
                         v[9]+subkey[1]+sum[(t+7)%8],
                         v[10]+subkey[2]+sum[(t+7)%8],
                         v[11]+subkey[3]+sum[(t+7)%8],
                         v[12]+subkey[4]+sum[(t+7)%8],
                         v[13]+subkey[5]+sum[(t+7)%8],
                         v[14]+subkey[6]+sum[(t+7)%8],
                         v[15]+subkey[7]+sum[(t+7)%8]);

     /* XOR with the left side */
     for (t=0; t<8; t++)
       sum[t] ^= v[t];

     /* Swap left and right sides */
     for (t=0; t<8; t++) {
       v[t] = v[t+8];
       v[t+8] = sum[t];
     }
    }
}

 /**********************************************/
 /* Block Decryption */
 /**********************************************/

 void decrypt(unsigned long* v, unsigned long* k,
              int Nrounds)
{

    unsigned long **subkeys, sum[8];
    int i, t, j;

    /* Memory allocation for the subkeys table */
    subkeys = (unsigned long **) malloc(
              (Nrounds+1)*sizeof(unsigned long *));
    for (i=0; i<Nrounds; i++)
       subkeys[i] = (unsigned long *) malloc(8*
                    sizeof(unsigned long));
    subkeys[i] = NULL;

    /* Generation of all the subkeys */
    subkeys[0][7]=0;
    for(i=0; i<Nrounds; i++) {
     /* Subkey generation for this round */
     if (i!=0) subkeys[i][7] = subkeys[i-1][7];
     for (t=0; t<8; t++)
       subkeys[i][t] = key_schedule(
                         k[0]+subkeys[i][(t+7)%8],
                         k[1]+subkeys[i][(t+7)%8],
                         k[2]+subkeys[i][(t+7)%8],
                         k[3]+subkeys[i][(t+7)%8],
                         k[4]+subkeys[i][(t+7)%8],
                         k[5]+subkeys[i][(t+7)%8],
                         k[6]+subkeys[i][(t+7)%8],
                         k[7]+subkeys[i][(t+7)%8]);
     }
     for(i=0; i<Nrounds; i++) {
      /* Application of the F-function */
      sum[7] = 0;
      for (t=0; t<8; t++)
       sum[t] = round_function(
         v[0]+subkeys[Nrounds-i-1][0]+sum[(t+7)%8],
         v[1]+subkeys[Nrounds-i-1][1]+sum[(t+7)%8],
         v[2]+subkeys[Nrounds-i-1][2]+sum[(t+7)%8],
         v[3]+subkeys[Nrounds-i-1][3]+sum[(t+7)%8],
         v[4]+subkeys[Nrounds-i-1][4]+sum[(t+7)%8],
         v[5]+subkeys[Nrounds-i-1][5]+sum[(t+7)%8],
         v[6]+subkeys[Nrounds-i-1][6]+sum[(t+7)%8],
         v[7]+subkeys[Nrounds-i-1][7]+sum[(t+7)%8]);

      /* XOR with the left side */
      for (t=0; t<8; t++)
        sum[t] ^= v[t+8];

      /* Swap left and right sides */
      for (t=0; t<8; t++) {
        v[t+8] = v[t];
        v[t] = sum[t];
      }
    }

    for (i=0; i<Nrounds; i++)
    free(subkeys[i]);
    free(subkeys);
}
```

References

[1] H. Feistel, “Cryptography and Computer Privacy,”, in *Scientific American*, Vol. 228, No. 5, May 1973, pp. 15–23.

[2] National Bureau of Standards, NBS FIPS PUB 46, “Data Encryption Standard,” National Bureau of Standards, U.S. Department of Commerce, Jan 1977.

[3] A. Shimizu and S. Miyaguchi, “Fast Data Encipherment Algorithm FEAL,” in *Advances in Cryptology – EUROCRYPT ’87*, Lecture Notes in Computer Science, Vol. 304, Springer-Verlag, 1988, pp. 267–278.

[4] GOST, Gosudarstvennyi Standard 28147-89, “Cryptographic Protection for Data Processing Systems,” Government Committee of the USSR for Standards, 1989.

[5] L. Brown, M. Kwan, J. Pieprzyk, and J. Seberry, “Improving Resistance to Differential Cryptanalysis and the Redesign of LOKI,” in *Advances in Cryptology – ASIACRYPT ’91*, Lecture Notes in Computer Science, Vol. 739, Springer-Verlag, 1993, pp. 36–50.

[6] C.M. Adams and S.E. Tavares, “Designing S-boxes for Ciphers Resistant to Differential Cryptanalysis,” in *Proceedings of the 3rd Symposium on State and Progress of Research in Cryptography*, Rome, Italy, 15–16 Feb. 1993, pp. 181–190.

[7] B. Schneier, “Applied Cryptography, Second Edition.” John Wiley & Sons, 1996.

[8] R.L. Rivest, “The RC5 Encryption Algorithm,” in *Fast Software Encryption, Second International Workshop Proceedings*, Lecture Notes in Computer Science, Vol. 1008, Springer-Verlag, 1995, pp. 86–96.

[9] R. L. Rivest, M. J. B. Robshaw, R. Sidney, and Y. L. Yin, “The RC6 Block Cipher,” v1.1, August 20, 1998.

[10] M. Luby and C. Rackoff, “How to Construct Pseudorandom Permutations and Pseudorandom Functions,” in *SIAM Journal on Computing*, Vol. 17, 1988, pp. 373–386.

[11] B. Schneier and J. Kelsey, “Unbalanced Feistel Networks and Block-Cipher Design.” in *Fast Software Encryption, Third International Workshop Proceedings*, Lecture Notes in Computer Science, pp. 121–144, Springer-Verlag, 1996.

[12] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

[13] R. Forré, “The strict avalanche criterion: spectral properties of boolean functions and an extended definition,” in *CRYPTO 88*, Lecture Notes in Computer Science, pp. 450-468. Springer-Verlag New York, Inc., 1990.

[14] The `lil-gp` genetic programming system is available at: *http://garage.cps.msu.edu/software/lil-gp/lilgpindex.html*

[15] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, “The microarchitecture of the pentium 4 processor.” in *Intel Technology Journal*, Q1 2001.

[16] D. J. Wheeler and R. M. Needham, “TEA, a tiny encryption algorithm,” in *Lecture Notes in Computer Science*, Vol. 1008, pp. 363-369. Springer-Verlag, 1995.

[17] J.Kelsey, B. Schneier, D. Wagner, “Related-Key Cryptanalysis of 3-WAY, Biham-DES,CAST, DES-X, NewDES, RC2, and TEA” in *Proceedings of the ICICS 97 Conference. Lecture Notes in Computer Science*, Vol. 1334, pp. 233-246. Springer-Verlag, 1997.

[18] M. Matsumoto, T. Nishimura, ”Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator”, ACM Trans. on Modeling and Computer Simulation Vol. 8, No. 1, January pp.3-30, 1998

[19] M. Seredynski, P. Bouvry, “Block cipher based on reversible cellular automata” Next Generation Computing Journal, 23(3), pp. 245-258, May 2005.

[20] S. Wolfram, “Cryptography with cellular automata”, Advances in Cryptology: Crypto ’85 Proceedings, LNCS, vol. 218, pp 429-432, 1986.

[21] M. Mihaljevic, Y. Zheng and H. Imai, ”A cellular automaton based fast one-way hash function suitable for hardware implementation”, Lecture Notes in Computer Science, vol. 1431, pp. 217-233, 1998

[22] S. Hirose and S. Yoshida, ”A one-way hash function based on a two-dimensional cellular automaton”, The 20th Symposium on Information Theory and Its Applications (SITA97), Matsuyama, Japan, Dec. 1997, Proc. vol. 1, pp. 213-216.

[23] S. Nandi et al. ”’Theory and applications of cellular automata in cryptography”’, IEEE Transactions on Computers, Vol. 43, pp. 1346-1357, 1994

[24] M. J. Mihaljevic ”‘An improved key stream generator based on the programmable cellular automata”’, Lecture Notes in Computer Science, Vol. 1334, pp. 181-191, 1997

[25] J. Walker, ENT: http://www.fourmilab.ch/random/

[26] G. Marsaglia, W. W. Tsang ”‘Some Difficult-to-pass Tests of Randomness”’, Journal of Statistical Software, Volume 7, 2002, Issue 3

[27] A. L. Rukhin, J. Soto, J. R. Nechvatal, M. Smid, E. B. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, N. A. Heckert, J. F. Dray, S. Vo ”‘A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Application”’ NIST SP 800-22, U.S. Government Printing Office, Washington:2000, CODEN: NSPUE2