International Journal of Knowledge-based and Intelligent Engineering Systems 11 (2007) 1–12 1
IOS Press

Automated design of a lightweight block cipher with Genetic Programming

Javier Polimón*, Julio C. Hernández-Castro, Juan M. Estévez-Tapiador and Arturo Ribagorda
Computer Science Department, Carlos III University of Madrid, 28911 Leganes, Madrid, Spain

Abstract. In this paper, we present a general framework for the automated design of cryptographic block ciphers by using Genetic Programming. We evolve highly nonlinear and extremely efficient functions that can be used as core components of any cryptographic construction. As an example, a new block cipher named Raiden is proposed. We present a preliminary security analysis of our proposal and a comparison in terms of performance with similar block ciphers such as TEA. The results show that automatically-obtained schemes, such as the one presented here, could be competitive both in security and speed.

1. Introduction

Block ciphers are key components of nearly all cryptographic protocols, and of many security applications. Common usage scenarios abound, because block ciphers are used (together with stream ciphers) whenever a given degree of confidentiality is to be provided. They are a very basic cryptographic primitive, as they could also be used to build or derive hash functions, stream ciphers, pseudorandom number generators and message authentication codes (MACs).

1.1. Overview of the rest of the paper

The paper is organized as follows. Section 2 introduces some theoretical background about block ciphers, including concepts such as Feistel networks and the avalanche effect, and presents the TEA block cipher. In Section 3 we present the approach used to search for lightweight functions with a nearly optimal degree of avalanche effect. Our results, including the final design of a block cipher, are presented in Section 4. In Section 5 we provide a analysis of our proposal, both in terms of speed and security. Finally, Section 6 concludes the paper presenting our main conclusions and future research directions.

2. Theoretical background

For completeness and readability, we first introduce some basic concepts and cryptographic constructions that will be extensively used throughout this paper.

*Corresponding author. E-mail: {jpolimon, jcesar, jestevez, arturo}@inf.uc3m.es.

ISSN 1327-2314/07/$17.00

2.1. Block-cipher design

A block cipher consists of two related 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, so it should hold:

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

For each key k, E_k should behave as closely as possible to a random permutation (a bijective mapping) over the set of input blocks. Each key selects one permutation among the $2^n!$ possibilities. Typical key sizes used 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.

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 and 32 rounds.

2.1.1. Feistel networks

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 nearly identical (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. As a consequence, they 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 [9] that if the round function F is a cryptographically secure pseudorandom number generator, with K_i (a parameter derived from the key, usually termed the i-round subkey) used as the seed, then 3 rounds are sufficient to make the block cipher secure, while 4 rounds are sufficient to make the block cipher strongly secure (i.e. secure against chosen-ciphertext attacks). In a similar vein, in [10] the author improves the proven security bounds for random Feistel schemes with 5 rounds, showing that there is no adaptive chosen plaintext/chosen ciphertext attack on then where an attacker with unlimited computer power but limited to m queries can have any success, with m significantly less than 2^n queries.

In a classical Feistel network half of the bits operate on the other half. As pointed out by Bruce Schneier and John Kelsey [11], there is no inherent reason that this should be so. Later works have generalized and extended this basic structure, but in this preliminary study we shall focus our attention exclusively in the classical Feistel scheme.

2.1.2. 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:

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

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 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 reverse order:

$$R_{i-1} = L_i$$

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

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.

2.2. Highly-nonlinear functions and the avalanche effect

The avalanche effect is a mathematical property that tries, to some extent, to abstract the intuitive idea of high-nonlinearity: a very small difference in the input should produce 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:

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

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 input bits should be, on average, $n/2$. That is, a minimum input change (one single bit) produces, on average, the change of half of the output bits.

This definition also tries to abstract the more general concept of output independence from the input. Although it is clear that this independence is impossible to achieve (a given input vector always produces the same output), the ideal round function F will resemble a perfect random function where inputs and outputs are statistical 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 [12] which, in particular, implies the Avalanche Effect, and that could be mathematically described as:

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

where B denotes a binomial distribution. It is interesting to note that the previous expression implies the avalanche effect, as the average of a binomial distribution with parameters $1/2$ and n is exactly $n/2$.

Furthermore, the amount of proximity of a given distribution to another (a $B(1/2, n)$ in this case) can be easily measured by means of a χ^2 goodness-of-fit test.

Having a good degree of avalanche effect is more than a desirable property for many cryptographic primitives. A block cipher or a hash function which does not have a significant avalanche effect makes a poor diffusion of its input, and as a consequence some forms of cryptanalysis can be successfully applied.

In some cases, this fact can be enough to completely break the primitive. For this reason constructing a primitive with a substantial degree of avalanche is a primary design goal.

2.3. TEA and XTEA: Tiny encryption algorithms

TEA [13] is a block cipher designed by David Wheeler and Roger Needham, of the Cambridge Computer Security Laboratory, and presented at the 1994 Fast Software Encryption Workshop.

TEA follows a Feistel-like scheme that obtains its robustness by combining the use of addition and xor operations to achieve a high degree of nonlinearity. The use of shifts, additionally, makes that key bits are mixed with input bits twice in each round.

Every TEA cycle has two rounds, and its authors state that in the worst possible scenario six cycles are needed to ensure that a 1 bit change in the key or input bits generates around 32 changes in the 64 bit output. The authors recommend the use of 32 cycles (thus, 64 rounds) to ensure an adequate security level. TEA operates over 64 bit blocks, with 128 bit keys.

TEA code is extremely simple: It can be written in a few lines and it is even easy to memorize:

```
/*********************************************/
/* TEA Block Cipher */
/*********************************************/
void encrypt(unsigned long* v, unsigned long* k) {
unsigned long v0=v[0], v1=v[1], sum=0, i;
unsigned long delta=0x9e3779b9;
unsigned long k0=k[0], k1=k[1], k2=k[2], k3=k[3];
for (i=0; i < 32; i++) {
sum += delta;
v0 += ((v1<<4) + k0) ^ (v1 + sum) ^ ((v1>>5) + k1);
v1 += ((v0<<4) + k2) ^ (v0 + sum) ^ ((v0>>5) + k3);
}
v[0]=v0; v[1]=v1;
}
```

It does not use S-boxes nor a complex key-schedule and, in fact, this last feature has turned out to be one of its major weaknesses, becoming a key player in most of the cryptanalytic attacks proposed so far.

The two most important attacks against TEA are:

Related-key attacks: Pointed out by John Kelsey, Bruce Schneier, and David Wagner, in [14].

Existence of equivalent keys: In TEA, every key is equivalent to another three keys. This reduces its effective key size from 128 to 126 bits.

There are, however, more published attacks that, although only applicable to reduced-round versions of the algorithm, cast a doubt on its overall security [15].

At the present time, the cryptographic community has mostly shifted its interest to ciphers with 128 bits of block and key lengths, but for historical reasons, as well as for allowing fair comparisons with the TEA algorithm, we are focusing in this paper on 128 bit keylengths and 64 bit blocks. However, the described method is easily applicable to obtain block ciphers of any block and key length.

In order to solve some of the most important TEA weaknesses, its authors designed a new version called XTEA (eXtended TEA) where the simple TEA key schedule was changed by a more complex one:

```
/*********************************************/
/* XTEA Block Cipher */
/*********************************************/
void encipher(unsigned int num_rounds, unsigned long* v,
unsigned long* k)
{
 unsigned long v0=v[0], v1=v[1], i;
 unsigned long sum=0, delta=0x9E3779B9;
 for(i=0; i<num_rounds; i++)
  {
   v0 += ((v1 << 4 ^ v1 >> 5) + v1)^(sum + k[sum & 3]);
   sum += delta;
   v1 += ((v0 << 4 ^ v0 >> 5) + v0)^(sum + k[sum>>11 & 3]);
  }
 v[0]=v0; v[1]=v1;
}
```

XTEA, however, has not achieved a success comparable to that of TEA, in part because it is much slower than TEA and because soon after its proposal some additional weaknesses were published. Some results even suggest that it is, in fact, weaker than its predecessor with respect to many attacks [16–18].

TEA has been implemented in a huge number of different programming languages and has been used in many applications: It was the basis of the hash function used in the XBOX (which turned out to be a bad idea), it is widely

used for confidentiality in IRC channels, and also for encryption in PDAs and other mobile devices due to its very high speed and portability.

Its success, specially before the appearance of the two main cryptanalytic attacks, justifies the need for a secure alternative that should enjoy all its main properties (i.e. speed, short code, low gate count, easy to remember and to implement, high portability) but having increased security.

3. Methodology and experimentation issues

At the core of our work is the idea of designing functions with a (nearly) optimal amount of avalanche effect. These functions can be easily adapted to work as good components of cryptographic constructions, such as the round function or the key schedule algorithm of a block cipher. However, instead of using predesigned structures for such functions, we make use of a general approach to automatically find appropriate constructions: Genetic Programming.

Genetic Programming is a stochastic population-based search method devised in 1992 by John R. Koza [19]. It is inspired by Genetic Algorithms, the main difference with them being the fact that in the latter, chromosomes are used for encoding possible solutions to a problem, while GP evolves whole computer programs. Within the scope of Evolutionary Algorithms, this is the 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 non-trivial task. 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, static set (neurons in NN, attribute comparisons in ID3, etc.).

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/evaluations or a global 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, like 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.

3.1. Experimentation

We have used the `lil-gp` genetic programming library [20] 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.

3.1.1. 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 algorithms we would obtain. Efficiency being 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, such as those included in the TEA round function:

```
y += (z<<4)+ a ^ z + sum ^ (z>>5) + b
```

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.

We introduced at first the **mult** (multiplication mod 2^{32}) operator but, although the individuals thus obtained enjoyed a high degree of nonlinearity, they could not compete against the TEA round function in terms of speed. This, however, highly depends on the particular architecture used, so the multiplication of two 32 bit values could cost up to fifty times more than an **xor** or an **and** operation (although this happens in certain platforms, its nearly a worst case: 14 times [21] seems to be a more common value).

After many experiments, we decided to introduce the substraction operator **resta** and to substitute the rotation operators **vrotd**, **vroti** for shift operators **rotd**, **roti** which were much faster and generated individuals of similar nonlinearity.

3.1.2. Terminal set

The set of terminals in our case is relatively easy to establish provided that we are looking for a block cipher that operates on blocks of 64 bits with keys of 128 bits. Firstly, it is mandatory for the key schedule algorithm to operate with the 128 bits of the key, in our case expressed as four 32-bit integers. Second, the round function should accept as input the 32 bits of the different round subkeys and half the 64-bit input block. Thus, the terminals used by the GP system in this case will be represented, for the round function, by two 32-bit unsigned integers a_0, and sk; for the key schedule we need four 32-bit integers represented by four 32-bit terminals named a_0, a_1, a_2, and a_3.

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 [19]). 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, an idea suggested by [13].

3.1.3. Fitness function

We have used the same fitness function 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. An additional 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 [14]. To achieve this, we used the following fitness function

$$\text{Fitness } = 10^6/\chi^2$$

where χ^2 is calculated as follows:

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

and

$$E_k = 2048 * Pr\big(B(1/2, 32) = k\big)$$

So the fitness of every individual is calculated as follows: First, we use the Mersenne Twister generator [22] to generate an adequate number of 32-bit random values. Those values are assigned to (a_0, a_1, $\ldots a_i$). The value of our key schedule or round function candidate over this input o_0 is stored. Then, we randomly flip one single bit

Table 1
Average avalanche and execution time comparison

Function	Average Hamming Value	Execution Time (ms.)
Raiden Round Function	6.22	0.87
TEA Round Function	5.13	0.84

of one of the input values, and we run the algorithm again, 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 (2048 was experimentally found to be enough to give an adequate precision of the measurement, and quick enough to allow for rapid fitness computation) and each time a Hamming distance among 0 and 32 is obtained and stored. For a perfect Avalanche Effect, the distribution of the Hamming distances should be consistent with the theoretical Bernoulli probability distribution B(1/2,32). Therefore, the fitness of each individual is calculated by using 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 minimize the χ^2 statistic in order to maximize the expression for the fitness function shown above.

We should note that 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 and, thus, the sensitivity of our measure.

3.1.4. 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 the latter option.

As we were interested in a fast key schedule function, we limited the number of nodes to 10. On the other hand, we allowed the round function to use up to 15 nodes to try 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.

In both cases, each run of the GP system consisted of 5000 generations of a population of 800 individuals, with the crossover probability being 0.8, and 0.2 for reproduction.

4. Results

The best function f_r found when searching for a round function algorithm is shown in Fig. 1 in the classic Lisp-like notation provided by `lil-gp`. This is an algorithm with an avalanche effect of 6.22 (so when randomly flipping one input bit, the 32 bit output changes 6.22 bits, on average). This is a good result, specially when compared with the value associated with TEA's round function which is limited to 5.13, while both have a similar speed.

On the other hand, the tree corresponding to the best individual found when looking for the key schedule (f_k) is also depicted in Fig. 1. This key schedule algorithm provokes an avalanche effect of 2.94, which is more than one unit higher than the associated value of the TEA key schedule of 1.83.

4.1. The Raiden block cipher

These two new functions have been used as the key components of a new block cipher named Raiden.

The result is a classic Feistel network wherein each round operates according to the Feistel structure. Further possibilities will be explored in future works.

The C code including both functions and implementing the full code of the Raiden block cipher is shown below:

```
/*********************************************/
/* Raiden Block Cipher */
/*********************************************/
void raiden(unsigned long *in,unsigned long *res,unsigned long *key)
```

Function f_r	Function f_k
(sum (sum a0 a1) (xor (resta a2 a3) (roti a0 a2)))	(xor (roti (resta sk a0) 9) (xor (resta sk a0) (rotd (sum sk a0) 14))))

Fig. 1. Best individuals found.

```
{ unsigned long b0=in[0],b1=in[1],i,k[4]={key[0],key[1],key[2],key[3]},sk;
for(i=0; i< 16; i++)
{
sk=k[i%4]=((k[0]+k[1])+((k[2]-k[3])^(k[0]<<k[2])));
b0 += ((sk+b1)<<9)^((sk-b1)^((sk+b1)>>14));
b1 += ((sk+b0)<<9)^((sk-b0)^((sk+b0)>>14));
}
res[0]=b0;
res[1]=b1;
}
```

As the number of rounds is an adjustable parameter, we will use the notation Raiden-Rn to denote the cipher with n rounds. In our preliminary analysis (see below) Raiden has proven to be strong enough with 8 cycles. However, we strongly recommend using at least 16 cycles (32 rounds) to ensure an adequate security margin.

5. Analysis

We have performed a preliminary analysis of our proposal, both in terms of speed and security. The results are provided in what follows.

5.1. Security

The repeated mixing of 32-bit addition (which is nonlinear over $\mathbb{Z}_2$) and **xor** (which is nonlinear in $\mathbb{Z}_{32}$), is intended to provide for a good resistance against both differential [**?**] and linear cryptanalysis [**?**].

Additionally, other operations such as rotation, are included to give adequate diffusion by extending changes from high significant bits to low significant bits, and vice versa. All in all, after the proposed 16 cycles, we conjecture that Raiden is secure against both linear and differential cryptanalysis, so that no attacks significantly faster than exhaustive key search exist.

Moreover, the combined use of the proposed operations makes the existence of weaknesses against *Mod n* cryptanalysis highly unlikely, as addition and rotation (two of the most vulnerable operations) are not used alone but together with **xor**, as proposed in [**?**] (authors claim that both **xor** and multiplication mod 2^{32} are very difficult to approximate mod 3) to increase strength against this kind of attack.

A major advantage of not using the multiplication mod 2^{32} operation comes with respect to timing attacks where the input-dependent time used to perform this operation (due to optimizations) could leak some information [**?**]. Other operations that are usually prone to timing attacks and that we have avoided by construction are data-dependent rotations (only fixed-amount rotations are used), and s-box lookups (Raiden doesn't obtain its non-linearity by the use of s-boxes).

Timing attacks will, then, not work against Raiden. For more insights on the subject, the reader should refer to the excellent discussion in [**?**].

Furthermore, the key schedule algorithm has been deliberately chosen to be complex enough to avoid related-key attacks [**?**] to which algorithms with simpler key schedules are prone. The fact that the proposed key schedule

Table 2
Tests results obtained with ENT

	TEA	Raiden
Entropy	7.999994 bits/byte	7.999994 bits/byte
Compression Rate	0.00%	0.00%
χ^2 Statistic	(25%)	(50%)
Arithmetic Mean	127.4946	127.5125
Monte Carlo π estimation	(0.01%)	(0.01%)
Serial correlation coefficient	0.000076	0.000044

Table 3
Overall p-values obtained with the DIEHARD suite

	p-value(s)	
Test	Raiden	TEA
1	0	0
2	0	0
3	0.6295	0
4	0.6819	0.4251
5	0.1727	0.6344
6	0.8949	0.3825
7	0.6066	0.6524
8	0.5384	0.3358
16	0.4253	0.8230
32	0.4034	0.9637

achieves a high degree of avalanche ensures, to a certain extend, that no trivial input differences will produce output differences that could be used to mount a cryptanalytic attack, and that no equivalent nor weak keys exist.

In particular, and following the advice presented in the *Prudent Rules of Thumb for Key-Schedule Design* section in [**?**], and in the *Designing Strong Key Schedules* section at [**?**] (where authors suggest "[...] we recommend that designers maximize avalanche in the subkeys [...]") we have avoided linearity in the design, and we have additionally avoided the generation of independent round subkeys.

5.2. Statistical properties

An additional analysis consists in examining the statistical properties of the output over a very low-entropy input. In our case, we have generated a large battery of low-entropy inputs by ANDing and ORing random numbers multiple times. Following this scheme, we have produced a number of files to be used as input for test batteries.

The resulting ciphertext has been analyzed with four batteries of statistical tests, namely ENT [**?**], Diehard [**?**], NIST [**?**] and SEXTON [**?**]. The results obtained are presented in the following Tables.

From this we can conclude that Raiden successfully passes the ENT randomness battery (results shown in Table 1), as all of the obtained statistics are well within the confidence interval that should correspond to a random variable. Additionally, this is confirmed by the fact that TEA obtains similar results to those of Raiden.

In Table 3, we show the overall p-value that Diehards gives after the computation of all its 229 tests. From this we can conclude that Raiden successfully passes the Diehard test battery, as all of the obtained statistics are well within the confidence interval that should correspond to the measure of a random variable, which is approx. (0.05, 0.95). Additionally, this is confirmed by the fact that TEA obtains similar (but slightly worse) results than those of Raiden.

In Table 4, we can observe the results of both Raiden and TEA over the NIST statistical test, where the most meaningful columns are those that show the proportion of tests passed. It is generally considered that if for any test this proportion of successfully passed tests is lower than 0.96, the overall NIST test should be considered as not passed. From that we can conclude that TEA does not pass all the tests (it fails the Runs, Non-periodic Templates and Random. Excursion Variant tests) while Raiden successfully passes them all.

Finally, we will analyze the results obtained over the SEXTON battery of tests presented in Table 5: Here we can observe that the results of Raiden and TEA are even better than those obtained by two well-known ciphers RC-4 and Snow. This test battery comprises 70 different tests, adjusted so not all of them could be passed even by a random

Table 4
NIST's battery results

Test	Raiden p-value	Raiden proportion	TEA p-value	TEA proportion
Frequency	0.671	1.000	0.739	0.969
Block Frequency	0.350	1.000	0.804	1.000
Cumulative Sums	0.732	0.969	0.736	0.969
Runs	0.468	1.000	0.534	**0.937**
Longest Run	0.253	0.969	0.299	0.969
Rank	0.350	0.969	0.464	1.000
FFT	0.739	0.969	0.011	0.969
Nonperiodic Templates	0.213	1.000	0.299	**0.937**
Overlapping Templates	0.407	1.000	0.862	1.000
Universal	0.468	1.000	0.534	1.000
Apen	0.804	1.000	0.804	1.000
Random Excursions	0.331	1.000	0.306	1.000
Rand. Ex. Variant	0.437	1.000	0.275	**0.954**
Serial	0.581	1.000	0.602	1.000
Linear Complexity	0.862	1.000	0.534	0.969

Table 5
Sexton's battery results

Result	Raiden	TEA	RC4	Snow	Random Variable
Passed tests	53	53	55	58	56
Failed (1 order)	16	17	14	11	12
Failed (2 orders)	1	0	1	1	2
Failed (3 orders)	0	0	0	0	0
Failed (4 orders)	0	0	0	0	0

variable. When not passed, the SEXTON test battery observes the amount of the distance to the expected confidence interval, thus assigning different failure "orders". Typical non cryptographic generators usually fail one or more of these tests with 3 and 4 orders of error. Together with the NIST test, the SEXTON test is one of the most stringent test batteries.

Although authors acknowledge that statistical tests are not very meaningful, and do not pretend to prove the security of any cryptographic primitive by showing that its output (even over very low-entropy inputs) passes a number of batteries of tests, we believe that these results are useful to show that no trivial weaknesses exist in the proposed constructions nor in their implementations, so they deserve the scrutiny of the academic community.

6. Conclusions and future work

The results described in this paper show that paradigms such as Genetic Programming can be successfully applied to design competitive (in terms of security and speed) cryptographic primitives. In this respect Raiden, which has proven to be even better than TEA in some tests, can be thought as an instance of an entire family of designs. In particular, different proposals could have been obtained by repeating the experimentation.

We believe that the proposed scheme incorporates robustness against most of the currently-known attacks by construction. However, further work has to be done before considering these new cryptographic primitive as secure enough.

We do not pretend to prove any kind of security properties just by analyzing the results of a battery of statistical tests (even if they include very stringent tests over very low-entropy inputs). However, we believe that these results show that no trivial weaknesses exist in the proposed constructions nor in their implementations, so they deserve the scrutiny of the academic community.

From the authors' point of view, the possibility of automatically obtaining cryptographic primitives could have interesting additional implications (to fields like Governmental policies on Cryptography, Controls on Cryptographic Research and Export, etc.) which should be tackled by future works.

7. Raiden decryption routine C code

```
void raiden-decypher(unsigned long *data, unsigned long
*res, unsigned long *key)
{ unsigned long b0=data[0],b1=data[1],i, k[4]={key[0],key[1],key[2],key[3]}, keys[16];
for(i=0;i<16;i++) {
//Round subkeys computation
 k[i%4]=((k[0]+k[1])+((k[2]+k[3])^(k[0]<<k[2])));
 keys[i]=k[i%4];
}
for(i=16; i!=-1; i--) {
//Round subkeys are applied in reverse order
 b1 -= ((keys[i]+b0)<<9)^((keys[i]-b0)^((keys[i]+b0)>>14));
 b0 -= ((keys[i]+b1)<<9)^((keys[i]-b1)^((keys[i]+b1)>>14));
}
 res[0]=b0; res[1]=b1; }
```

References

[1] H. Feistel, Cryptography and Computer Privacy, *Scientific American* **228**(5) (May 1973), 15–23.
[2] National Bureau of Standards, NBS FIPS PUB 46, Data Encryption Standard National Bureau of Standards, US 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] M. Luby and C. Rackoff, "How to Construct Pseudorandom Permutations and Pseudorandom Functions", *SIAM Journal on Computing* **17** (1988), 373–386.
[10] Jacques Patarin. "On Linear Systems of Equations with Distinct Variables and Small Block Size". Proceedings of the ICISC 2005, 299–321.
[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, Springer-Verlag, 1996, pp. 121–144.
[12] R. Forré, "The strict avalanche criterion: spectral properties of boolean functions and an extended definition" in: *CRYPTO 88*, Lecture Notes in Computer Science, Springer-Verlag New York, Inc., 1990, pp. 450–468.
[13] David J. Wheeler and Roger M. Needham, TEA, a tiny encryption algorithm, in: *Fast Software Encryption: Second International Workshop*, (vol. 1008) of Lecture Notes in Computer Science, Bart Preneel, ed., 1994, pp. 363–366.
[14] J. Kelsey, B. Schneier and 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), Springer-Verlag, 1997, pp. 233–246.
[15] Vikram Reddy Andem, A Cryptanalysis of the Tiny Encryption Algorithm, Masters thesis, The University of Alabama, Tuscaloosa, 2003.
[16] Deukjo Hong, Youngdai Ko, Donghoon Chang, Wonil Lee and Jongin Lim, "Differential cryptanalysis of XTEA" Technical Report TR0313, Center for the Information Security and Technologies (CIST), Seoul, Korea, 2003.
[17] Dukjae Moon, Kyungdeok Hwang, Wonil Lee, Sangjin Lee and Jongin Lim, "Impossible differential cryptanalysis of reduced round XTEA and TEA", *Lecture Notes in Computer Science* **2365** (2002), 49–60. ISSN 0302-9743.
[18] Youngdai Ko, Seokhie Hong, Wonil Lee, Sangjin Lee and Jongin Lim, Related key differential attacks on 26 rounds of XTEA and full rounds of GOST, In Proceedings of FSE '04, Lecture Notes in Computer Science, Springer-Verlag, 2004.
[19] J.R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, Cambridge, MA, USA, 1992.
[20] The `lil-gp` genetic programming system is available at: http://garage.cps.msu.edu/software/lil-gp/lilgpindex.html.
[21] 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.
[22] M. Matsumoto and T. Nishimura, Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator, *ACM Trans. on Modeling and Computer Simulation* **8**(1) (January, 1998), 3–30.
[23] E. Biham and A. Shamir, Differential cryptanalysis of the data encryption standard, *Proc. Advances in Cryptology* (*EUROCRYPT '98*), LNCS (Vol. 1403), 1998, pp. 85–99.

[24] M. Matsui, "Linear cryptanalysis method for DES cipher", *Proc. Advances in Cryptology* (*CRYPTO '94*), LNCS vol. 1994, pp. 386–397.
[25] J. Kelsey, B. Schneier and D. Wagner, Mod n Cryptanalysis, with Applications Against RC5P and M6, *Proc. Fast Software Encryption* (*FSE '99*), LNCS vol. 1636, 1999, pp. 139155.
[26] D.J. Bernstein. "Salsa20 design", http://cr.yp.to/snuffle/design.pdf. 2005.
[27] J. Kelsey, B. Schneier and D. Wagner, Key-Schedule Cryptanalysis of IDEA, G-DES, GOST, SAFER, and Triple-DES, *Proceedings of Advances in Cryptology—CRYPTO '96*, LNCS (vol. 1109), 1996, pp. 237–251.
[28] J. Walker, ENT: http://www.fourmilab.ch/random/.
[29] G. Marsaglia and W.W. Tsang, Some Difficult-to-pass Tests of Randomness, *Journal of Statistical Software* **7** (2002), Issue 3.
[30] 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 and S. Vo, A Statistical Test Suite for Random and Psuedorandom Number Generators for Cryptographic Application NIST SP 800-22, US Government Printing Office, Washington:2000, CODEN: NSPUE2
[31] David Sexton's webpage with instructions and the code of his test battery is at http://www.geocities.com/da5id65536/.