JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 24, 1485-1504 (2008)

Automated Design of Cryptographic Hash Schemes by Evolving Highly-Nonlinear Functions

JUAN M. ESTEVEZ-TAPIADOR, JULIO C. HERNANDEZ-CASTRO,
PEDRO PERIS-LOPEZ AND ARTURO RIBAGORDA
Department of Computer Science
Carlos III University of Madrid
28911 Leganes, Madrid, Spain
E-mail: {jestevez; jcesar; pperis; arturo}@inf.uc3m.es

In the last years, a number of serious flaws and vulnerabilities have been found in classic cryptographic hash functions such as MD4 and MD5. More recently, similar attacks have been extended to the widely used SHA-1, to such an extent that nowadays is prudent to switch to schemes such as SHA-256 and Whirlpool. Nevertheless, many cryptographers believe that all the SHA-related schemes could be vulnerable to variants of the same attacks, for all these schemes have been largely influenced by the design of the MD4 hash function. In this paper, we present a general framework for the automated design of cryptographic block ciphers and hash functions by using Genetic Programming. After a characterization of the search space and the fitness function, we evolve highly nonlinear and extremely efficient functions that can be used as the core components of a cryptographic construction. As an example, a new block cipher named Wheedham is proposed. Following the Miyaguchi-Preneel construction, this block cipher is then used as the compression function of a new hash scheme producing digests of 512 bits. We present a security analysis of our proposal and a comparison in terms of performance with the most promising alternatives in the near future: SHA-512 and Whirlpool. The results show that automatically-obtained schemes such as those presented are competitive both in security and speed.

Keywords: hash function, block cipher, non-linear functions, cryptography and coding, evolutionary computation, information security

1. INTRODUCTION

Cryptographic hash functions are key components of nearly all cryptographic protocols, and of many security applications. Common usage scenarios range from the reduction of the amount of data to be signed (due to the slow signing algorithms known to date), to timestamping or checking a file's integrity. Apart from these, hash schemes have been extensively used to generate pseudorandom numbers or to prove the knowledge of a secret piece of information without revealing it, to name just a few.

Formally, a hash function F: $\{0, 1\}^* \rightarrow \{0, 1\}^n$ maps an arbitrary-length string into a fixed-length string of n bits, with two basic properties:

1. *Compression*: For any input of length M bits, the output (hash value) is always a string of length n, usually with $n \ll m$.
2. *Ease of Computation*: Given F and an input string x, $F(x)$ should be easy to compute.

Received October 31, 2006; revised March 28, 2007; accepted May 2, 2007.
Communicated by Tzong-Chen Wu.

The hash of a string is mainly used to univocally identify the string. However, as $|\{0, 1\}^n| \ll |\{0, 1\}^*|$, there always will be *collisions*, *i.e.* pairs of different strings $m_1, m_2 \in \{0, 1\}^*$, with $m_1 \neq m_2$, such that $F(m_1) = F(m_2)$.

Hash functions used in the much more demanding field of cryptography must satisfy three additional properties:

1. *Preimage resistance*: Given an output y, it must be computationally infeasible to find a preimage, *i.e.* a string x such that $F(x) = y$.
2. *Second preimage resistance*: Given an input string x_1, it must be computationally infeasible to find a different string $x_2 \neq x_1$ such that $F(x_1) = F(x_2)$. Functions satisfying this requirement are said to be weak collision resistant.
3. *Collision resistance*: It must be computationally infeasible to find any two different strings $x_1 \neq x_2$ such that $F(x_1) = F(x_2)$. Functions satisfying this requirement are said to be strong collision resistant.

If a hash function fails to satisfy one or more of the previous properties, it should not be used in cryptographic applications.

Unfortunately, every major cryptographic hash function in use is currently under successful attack, including the two most widely known, which are MD5 and SHA-1. In response to the recent SHA-1 attacks, the National Institute of Standards and Technology (NIST) has recommended to move from SHA-1 to the larger members of the SHA-2 family of hash functions (SHA-224, SHA-256, SHA-384, and SHA-512). These attacks have put a shadow on the global security of all the SHA-2 hash schemes, as many in the Crypto community now believe that all of the SHA-2 hash schemes could be vulnerable to variants of the same attacks that have been proven successful against their ancestors. This idea comes from the fact that all these schemes have been largely influenced by the design of the MD4 hash function (which is, by the way, completely broken nowadays).

Many consider it will be necessary, or at least prudent, to develop an additional hash function scheme in a competition similar to the one that leaded to the development of AES. For this reason, the NIST is organizing a series of workshops (two have been carried out to date), mainly for agreeing a replacement strategy for SHA-1. So, even if the current attacks over SHA-1 are considered not to be practical and SHA-1 is judged to be operationally secure, their developers and main advocates from NIST have started looking for replacement as everyone expects these attacks to get worse sooner or later.

In this current state of affairs, only the Whirlpool hash function seems to be a long-term serious replacement for the SHA-1/2 proposals. It uses the Miyaguchi-Preneel scheme with an underlying block cipher reminiscent of AES. However, NIST's idea is to sponsor a contest similar to AES to stimulate research in the area and to attract researchers to send their comments, suggestions and new proposals in the hope of finding new ideas and schemes that increase the general understanding on hashes, and helps in finding new and more secure algorithms.

1.1 Overview of the Rest of the Paper

The hash function presented in this work has at its core a number of automatically-obtained nonlinear functions. The methodology proposed to obtain such functions is

based on the use of Genetic Programming, so it was necessary to define a way for measuring nonlinearity to guide the search procedure.

The paper is organized as follows. Section 2 introduces some theoretical background about block ciphers, hash functions based on them, and the avalanche effect. In section 3 we present the approach used to search for functions with a nearly optimal degree of avalanche effect. Our results, including the final design of a block cipher and a hash function, are presented in section 4. In section 5 we provide an analysis of our proposals, both in terms of speed and security. Finally, section 6 concludes the paper presenting our main conclusions and future research directions. In addition, the source code (in C) of both proposals is provided in Appendices A and B.

2. THEORETICAL BACKGROUND

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

2.1 Block-cipher Design

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 $2^n!$ 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. 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 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 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. 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 [10] 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).

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. Despite further works have generalized this basic structure, in this work we will refer exclusively to 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 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 ith 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:

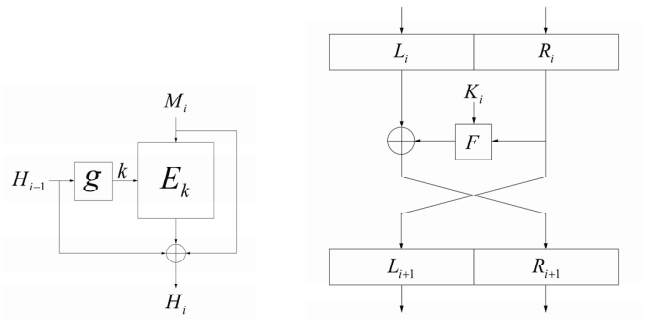
$$L_i = msb_{n/2}(X_i)$$
$$R_i = lsb_{n/2}(X_i)$$

where $lsb_u(x)$ and $msb_u(x)$ select the least significant and most significant u bits of x, respectively. In this way, $X_i = (L_i \parallel R_i)$. 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:

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



(a) (b)

Fig. 1. (a) Illustrations of Miyaguchi-Preenel scheme; (b) A round of a general Feistel network.

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

2.2 Hash Functions Based on Block Ciphers

There are several methods to use a block cipher to build a cryptographic hash function. Many schemes, such as that used in this paper, turn a block cipher into a compression function, which is then used within a Merkle-Damgård iterative structure.

The Miyaguchi-Preneel scheme is an extension of the Matyas-Meyer-Oseas method. The block cipher is fed with each message block m_i as the cleartext to be encrypted. The output block is then XORed with m_i and the previous hash value H_{i-1}, thus producing the current hash value or "state", H_i. In the first round, the state is initialized to constant, pre-specified values H_0.

Obviously, it is required that the block and key sizes be equal, as the previous state (H_{i-1}) is used as the key of the block cipher. Furthermore, the cipher might also have other requirements on the key. To solve any of these possible cases, it is assumed that there is a function $g()$, which transforms H_{i-1} into a valid key for the cipher. In this way, the Miyaguchi-Preneel scheme can be summarized in the following expression:

$$H_i = H_{i-1} \oplus m_i \oplus E_{g(H_{i-1})}(m_i).$$

Fig. 1 (a) graphically shows the construction.

2.3 Highly-nonlinear Functions and 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, we do not pretend to measure non-linearity, but a very specific mathematical property named avalanche effect. This property tries, to some extent, 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, a function F: $2^m \rightarrow 2^n$ has the avalanche effect if it holds that:

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

where $H(x, y)$ represents the Hamming distance between vectors x and y. So if F is to have the avalanche effect, the Hamming distance between the outputs of a random input vector x, and one generated by randomly flipping one of the bits of x (*i.e.* y) 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 [13] which, in particular, implies the Avalanche Effect, and that could be mathematically described as:

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

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 $n/2$.

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 amount of avalanche effect makes a poor diffusion of its input, and as a consequence some forms of cryptanalysis 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 effect is a primary design goal.

3. METHODOLOGY AND EXPERIMENTATION ISSUES

At the core of our work it 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 [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 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, unmodifiable 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 [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.

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. 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. 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. Additionally, 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 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 in the resulting functions did not justify the loss of Avalanche Effect.

3.1.2 Terminal set

The set of terminals in our case is relatively easy to establish provided that we are finding a block cipher that operates on blocks of 512 bits with keys of 256 bits. 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. Second, 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].

3.1.3 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

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

Here, O_h is the observed value of the distance between output vectors, and

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

is the expected value.

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 be consistent with 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 slightly 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, and thus the sensibility of our measure.

Furthermore, 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 with respect to the χ^2 statistic, and the fitness was guided only by the χ^2.

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 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.

4. RESULTS

The best function f_k found when searching for a key schedule algorithm is shown in Fig. 2 in the classic Lisp-like notation provided by lil-gp. This is an algorithm with an avalanche effect 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).

On the other hand, the tree corresponding to the best individual found when looking for the round function (f_r) is also depicted in Fig. 2. The round function provokes an avalanche effect of 15.9774 (16.0 is the optimal value) and a χ^2 statistic of 6.088 for a χ^2 probability distribution with 32 degrees of freedom.

Function f_k

```
=== 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))
```

Function f_r

```
=== 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 found.

Key Schedule Algorithm

```
 INPUT: k[0..7]
OUTPUT: subkey[0..7]

ALGORITHM:
 subkey[7] = 0
 For t=0 to 7 do
   a = subkey[(t+7) mod 8]
   subkey[t] = fk(k[0]+a, ... , k[7]+a)
 End-for
```

Round Function

```
 INPUT: k[0..7], subkey[0..7], v[0..7]
OUTPUT: f[0..7]

ALGORITHM:
 f[7] = 0
 For t=0 to 7 do
   a = subkey[t] + f[(t+7) mod 8]
   f[t] = fr(v[0]+a, ... , v[7]+a)
 End-for
```

Fig. 3. Usage of f_k and f_r as key schedule algorithm and round function, respectively.

4.1 The Wheedham Block Cipher

The two functions obtained have been used as the key components of a new block cipher named Wheedham. The result is a classic Feistel network wherein each round operates according to the structure shown in Fig. 1 (b).

The resulting algorithms are shown in Fig. 3. Both functions have been used in feedback mode, *i.e.* the input is always mixed with the last output. For notation purposes, $f_k(x_0, \ldots, x_7)$ and $f_r(x_0, \ldots, x_7)$ represent previous functions, where the 256-bit input has been split into 8 32-bit variables. The same has been considered for $k[0 \ldots 7]$, *subkey*$[0 \ldots 7]$, and $v[0 \ldots 7]$ (the 256-bit user key, the round subkey, and half the data block, respectively), which have been represented as arrays of 8 32-bit variables.

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

4.2 The MPW-512 Hash Function

The cryptographic hash proposal presented in this paper, named MPW-512, is a Miyaguchi-Preneel construction based on a modified version of the Wheedham block cipher (see Fig. 1 (a)) as the component E_k. Wheedham requires a key of 256 bits but processes blocks of 512 bits. Therefore, the *g*() function is then simply an XOR between the 256 most and least significant bits of the input. This is, $H_{i\text{-}1}$ is split into two equal blocks of 256 bits, which are then XORed and passed as the key.

The notation MPW-512-R*n* will be used to denote the hash scheme in which the compression function is Wheedham-R*n*. Due to the very nature of the Miyaguchi-Preneel construction, the block cipher is not required to behave (from a security point of view) as if it would be used isolated. For this reason, a low number of rounds (*e.g.* 2 or 4) seems enough to ensure an appropriate security level.

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 $\mathbb{Z}_{32}$), together with a highly nonlinear operation such as the multiplication mod 2^{32} is intended to provide for a good resistance against both differential and linear cryptanalysis [28].

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 rounds, we conjecture that Wheedham is secure against both linear and differential cryptanalysis [29], 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** and multiplication, as proposed in [30] (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.

The main security drawback of the use of 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 [33]. 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 in Wheedham), and s-box lookups (Wheedham doesn't obtain its non-linearity by the use of s-boxes).

Timing attacks, however, focus not on the algorithm but on its implementation, so it is possible to render an implementation of Wheedham immune at the cost of some speed. If avoiding timing attacks is a must, we propose an alternative for both the round function and the key schedule (see B) that does not rely on the use of multiplications nor any input-dependant functions and is, therefore, not vulnerable to this kind of side attack. For more insights on the subject, please refer to the excellent discussion in [33].

Furthermore, the key schedule algorithm has been deliberately chosen to be complex enough to avoid related-key attacks [31] to which algorithms with simpler key schedules are prone. The fact that the proposed key schedule 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 [31], and in the *Designing Strong Key Schedules* section at [32] (where authors suggest "[…] we recommend that designers maximize avalanche in the subkeys […]") we have evaded linearity in the design, and we have additionally avoided the generation of independent round subkeys.

No fixed points for the compression function are known to the authors, and no other vulnerabilities are currently known.

By using the Miyaguchi-Preneel scheme, we basically reduce the problem of attacking the hash function to that of attacking the underlying block cipher Wheedham. We know of no currents attacks for any of the two. The use of only 2 rounds of Wheedham in the compression function could be explained by the fact that, by construction, they are enough to achieve a nearly perfect amount of avalanche effect, and additional rounds,

while surely increasing the security of the overall scheme, will significantly decrease its speed.

We know of no weaknesses in the proposed scheme, and we conjecture MPW-512 is secure and collision-free, in that collisions for MPW-512 cannot be found with substantially less effort than the theoretical one. In particular, we conjecture that the difficulty of finding a message with a given message digest is in the order of $O(2^{512})$ operations, and the difficulty of finding two messages with the same message digest is in the order of $O(2^{256})$ operations.

Table 1. Entropy results obtained with ENT.

	Wheedham-R8	MPW-512-R2
Entropy	7.999999 bits/byte	7.999999 bits/byte
Compression Rate	0%	0%
χ^2 Statistic	254.32 (50%)	284.46 (10%)
Arithmetic Mean	127.4981	127.4986
Monte Carlo π estimation	3.141257768 (0.01%)	3.141290894 (0.01%)
Serial correlation coefficient	0.000096	0.000034

Table 2. Test results obtained with the DIEHARD suite.

	p-value(s)	
Test	Wheedham-R8	MPW-512-R2
Birthday Spacings	0.637468 0.739	0.716586 0.137
GCD	0.625940 0.425561	0.468790 0.171352
Gorilla	0.322	0.221
Overlapping Permutations	0.8776 0.3785 0.3903 0.6957 0.6853	0.5140 0.2616 0.4447 0.1479 0.4794
Ranks of 31×31 and 32×32 Matrices	0.499 0.509	0.219 0.047
Ranks of 6×8 Matrices	0.674742	0.829482
Monkey Tests on 20-bit Words	OK	OK
Monkey Tests OPSO, OQSO, DNA	OK	OK
Count the 1‘s in a Stream of Bytes	0.761543	0.612116
Count the 1‘s in Specific Bytes	OK	OK
Parking Lot Test	0.758757	0.938111
Minimum Distance Test	0.145881	0.619443
Random Spheres Test	0.345585	0.843300
The Squeeze Test	0.819257	0.129066
Overlapping Sums Test	0.003846	0.086223
Runs Up and Down Test	0.963	0.263
The Craps Test	0.025891 0.560224 0.611184 0.563171	0.09819 0.41816 0.859600 0.583844
Overall p-value	**0.272729**	**0.509349**

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 input messages X_i, such as messages consisting in series of 0's, series of 1's, series of 2's, incremental counters, *etc.* Following this scheme, we have produced a number of files (316 MB in total) to be used as input for both schemes.

First, we have tested the randomness of $E_{(0,\ldots,0)}(X_i)$, *i.e.*, encrypting the low-entropy input described above with key set to 0. The resulting ciphertext has been analyzed with three batteries of statistical tests, namely ENT [25], Diehard [26], and NIST [27]. The results obtained are presented in Tables 1 and 2. A similar experiment has been performed by analyzing the hashes obtained after applying MPW-512-R2 to the same inputs. Both proposals also passed the very demanding NIST statistical battery: All proportions are over 0.95 and all p-values compatible with a uniform $U(0, 1)$. Due to the huge amount of p-values generated, the report is not shown here.

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.

5.3 Performance

The speed test have been carried out over an Intel Pentium 4 CPU at 1.8 GHz according to the following procedure: We select a message size S and generate 100 random messages of size S. The hash function is applied to each of these 100 messages, measuring the time required to compute each of them. Finally, we take the average over the 100 samples.

This process has been iterated for messages sizes ranging from less than 1 MB to 10 GB. In order to compare our proposal with current, similar standards, the process has been repeated for SHA-512 and Whirlpool hash functions.

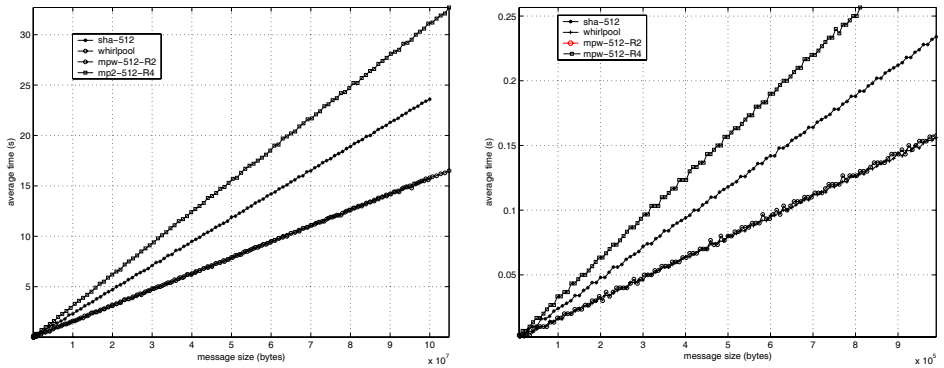


Fig. 4. Performance of MPW-512 with respect to SHA-512 and Whirlpool. The right graph shows the speed with short messages (< 1 Mbyte).

The results are shown in Fig. 4. MPW-512-R2 outperforms SHA-512 and runs practically at the same speed that Whirlpool. On the other hand, increasing the number of rounds to 4 makes the hash scheme to be slightly slower than SHA-512. In any case, it is clear that MPW-512 is competitive in terms of speed.

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, Wheedham and MPW-512 can be thought as instances of an entire family of designs. In particular, different proposals could have been obtained by repeating the experimentation.

We believe that both schemes incorporate robustness against most of the currently-known attacks by construction. However, it is obvious that more work has to be done before being able to consider these new cryptographic primitives 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 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 has additional implications (to fields like Gubernamental policies on Cryptography, Controls on Cryptographic Research and Export, *etc.*) which should be tackled by future works.

APPENDIX A: C SOURCE CODE

Next we provide a fully operative implementation of the Wheedham block cipher and the MPW-512 hash function. We assume a 32-bits architecture, so variables typed as unsigned long represent unsigned integers of 32 bits length. The interface functions are:

- encryption (v, k, NRounds): Encrypts v (a data block of 512 bits) using as key k (256 bits) and applying Nrounds rounds. The result is stored in v.
- decryption (v, k, NRounds): Decrypts the ciphered block v with key k and returns the result in v.
- MPW (buffer, length, H): Computes the hash of buffer, which has length length bytes. The result is returned in H.

```
/*************************************************/
/* 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);
}
```

```
/* 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];
```

```
/***********************************************/
/* 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],
    /* 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];
     }
    }

/***********************************************/
/* MPW  */
/***********************************************/
void MPW (unsigned long *buffer, int length,
          unsigned long *H)
{
  int i;
  unsigned long *p;

   /* Initialization */
   H[0] = 0xb7f9b656; H[1] = 0x2e9fdc17; H[2] = 0xed248575;
   H[3] = 0x9874a502; H[4] = 0x02b503a0; H[5] = 0x0a7b4eac;
   H[6] = 0x7998efdd; H[7] = 0xa4ef23e9; H[8] = 0x692f9f44;
   H[9] = 0xcaa896d6; H[10]= 0x5621076e; H[11]= 0xe01bdcdf;
   H[12]= 0xe7341c72; H[13]= 0xf627460c; H[14]= 0xfa82dd58;
   H[15]= 0x5922aabd;

   for (i=0, p=buffer; i<length; i++, p+=16)
      MPW_Round(p, H);

}

/***********************************************/
/* MPW_Round */
/***********************************************/
void MPW_Round (unsigned long *m, unsigned long *H)
{
   int i, Nrounds, b1;
   unsigned long r, key[8], plainBlock[16];

   /* g-function */
   for (i=0; i<8; i++)
      key[i] = H[i] ^ H[i+8];

   r = H[0];
   for (i=1; i<16; i++)
      r ^= H[i];
   b1 = r & 0x00000001;
   Nrounds = b1 ? 2 : 4;

   memcpy(plainBlock, m, 32);

   /* E-function */
   encrypt(plainBlock, key, Nrounds);

   /* Update */
   for (i=0; i<16; i++)
      H[i] = H[i] ^ m[i] ^ plainBlock[i];

}
```

APPENDIX B: ALTERNATIVE ROUND AND KEY SCHEDULE FUNCTIONS

In addition to the algorithms discussed before, we provide two alternatives for the key schedule and round functions. One of the major advantages of these new proposals is that the do not employ the mult operation. This feature makes them resilient to timing attacks.

The technique used to obtain these functions is slightly different from that described in this paper. Apart from removing the mult operation, the trees evolved are seriously limited in size, so functions evolved have a very low number of nodes. In this case, however, the tree is supposed to be within a finite-length loop (8 rounds in our case). As a result, a good degree of avalanche effect can be obtained by iterating very simple structures, avoiding operations such as multiplication.

```
/*************************************************/
/*Alternative Key Schedule: No timing attack     */
/*************************************************/
   int i;
   unsigned long T1;

   for(i=0;i<8;i++) {
     T1=(rshift(rshift(a3+a0,2)+a1,1)+a2+a7+a4+a0)^a5;
     T1=rshift(T1,1)+a6;
   }
   return(T1);
```

```
/*************************************************/
*Alternative Round Function: No timing attack   */
*************************************************/
  int i;
  unsigned long T1;

  for(i=0;i<8;i++) {
    T1 = ((a0+a2)^a4)^(a6+a5)^(a7^a1);
    T1 = (rshift(T1,4)+a0+a2)^a3;
  }
  return(T1);
```

REFERENCES

1. H. Feistel, "Cryptography and computer privacy," *Scientific American*, Vol. 228, 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, 1977.
3. A. Shimizu and S. Miyaguchi, "Fast data encipherment algorithm FEAL," in *Proceedings of Advances in Cryptology – EUROCRYPT*, LNCS 304, 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 *Proceedings of Advances in Cryptology – ASIACRYPT*, LNCS 739, 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*, 1993, pp. 181-190.
7. B. Schneier, *Applied Cryptography*, 2nd ed., John Wiley & Sons, 1996.
8. R. L. Rivest, "The RC5 encryption algorithm," in *Proceedings of the 2nd International Workshop on Fast Software Encryption*, LNCS 1008, 1995, pp. 86-96.
9. R. L. Rivest, M. J. B. Robshaw, R. Sidney, and Y. L. Yin, "The RC6 block cipher," v1.1, 1998.
10. M. Luby and C. Rackoff, "How to construct pseudorandom permutations and pseudorandom functions," *SIAM Journal on Computing*, Vol. 17, 1988, pp. 373-386.
11. B. Schneier and J. Kelsey, "Unbalanced Feistel networks and block-cipher design," in *Proceedings of the 3rd International Workshop on Fast Software Encryption*,

LNCS 1039, 1996, pp. 121-144.
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 *Proceedings of Advances in Cryptology – CRYPTO*, LNCS 403, 1990, pp. 450-468.
14. "The lil-gp genetic programming system," 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," *Intel Technology Journal*, Vol. Q1, 2001.
16. D. J. Wheeler and R. M. Needham, "TEA, a tiny encryption algorithm," in *Proceedings of the 2nd International Workshop on Fast Software Encryption*, LNCS 1008, 1995, pp. 363-369.
17. 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 1st International Conference on Information and Communication Security*, LNCS 1334, 1997, pp. 233-246.
18. M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudorandom number generator," *ACM Transactions on Modeling and Computer Simulation*, Vol. 8, 1998, pp. 3-30.
19. M. Seredynski and P. Bouvry, "Block cipher based on reversible cellular automata," *Next Generation Computing Journal*, Vol. 23, 2005, pp. 245-258.
20. S. Wolfram, "Cryptography with cellular automata," in *Proceedings of Advances in Cryptology – CRYPTO*, LNCS 218, 1986, pp. 429-432.
21. M. Mihaljevic, Y. Zheng, and H. Imai, "A cellular automaton based fast one-way hash function suitable for hardware implementation," in *Proceedings of the 1st International Workshop on Practice and Theory in Public Key Cryptography: Public Key Cryptography* , LNCS 1431, 1998, pp. 217-233.
22. S. Hirose and S. Yoshida, "A one-way hash function based on a two-dimensional cellular automaton," in *Proceedings of the 20th Symposium on Information Theory and its Applications*, 1997, Vol. 1, pp. 213-216.
23. S. Nandi, *et al*., "Theory and applications of cellular automata in cryptography," *IEEE Transactions on Computers*, Vol. 43, 1994, pp. 1346-1357.
24. M. J. Mihaljevic, "An improved key stream generator based on the programmable cellular automata," in *Proceedings of the 1st International Conference on Information and Communication Security*, LNCS 1334, 1997, pp. 181-191.
25. J. Walker, *ENT*, http://www.fourmilab.ch/random/.
26. G. Marsaglia and W. W. Tsang, "Some difficult-to-pass tests of randomness," *Journal of Statistical Software*, Vol. 7, 2002.
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, and S. Vo, "A statistical test suite for random and psuedorandom number generators for cryptographic application," NIST SP 800-22, U.S. Government Printing Office, Washington, 2000, CODEN: NSPUE2.
28. M. Matsui, "Linear cryptanalysis method for DES cipher," in *Proceedings of Ad-*

vances in Cryptology – EUROCRYPT, LNCS 765, 1994, pp. 386-397.
29. E. Biham and A. Shamir, "Differential cryptanalysis of the data encryption standard," in *Proceedings of Advances in Cryptology – EUROCRYPT*, LNCS 1403, 1998, pp. 85-99.
30. J. Kelsey, B. Schneier, and D. Wagner, "Mod *n* cryptanalysis, with applications against RC5P and M6," in *Proceedings of the 6th Fast Software Encryption Workshop*, LNCS 1636, 1999, pp. 139155.
31. 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 1st Information and Communication Security Conference*, LNCS 1334, 1997, pp. 233-246.
32. J. Kelsey, B. Schneier, and D. Wagner, "Key-schedule cryptanalysis of IDEA, G-DES, GOST, SAFER, and triple-DES," in *Proceedings of Advances in Cryptology – CRYPTO*, LNCS 1109, 1996, pp. 237-251.
33. D. J. Bernstein, "Salsa20 design," http://cr.yp.to/snu²e/design.pdf, 2005.

Juan M. Estevez-Tapiador is Associate Professor at the Computer Science Department of Carlos III University of Madrid. He holds a M.Sc. in Computer Science from the University of Granada (2000), where he obtained the Best Student Academic Award, and a Ph.D. in Computer Science (2004) from the same university. His research is focused on cryptography and information security, especially in formal methods applied to computer security, design and analysis of cryptographic protocols, and steganography. In these fields, he has published around 40 papers in specialized journals and conference proceedings. He is member of the program committee of several conferences related to information security and serves as regular referee for various journals. Currently, he is Visiting Professor at the Department of Computer Science, University of York, UK.

Julio C. Hernandez-Castro is Associate Professor at the Computer Science Department of Carlos III University of Madrid. He has a M.Sc. in Mathematics from Complutense University, and a M.Sc. in Coding Theory and Network Security from Valladolid University. He has a Ph.D. in Computer Science from Carlos III University. His interests are mainly focused in cryptography and cryptanalysis, network security, steganography and evolutionary computation. He has devoted most of his research to the applications of evolutionary computation tehcniques to cryptology, which also was the subject field of his Ph.D. Thesis. He has spent long research visits in the following security research centers: Cryptography and Computer Security Laboratory of Bradford University (UK), Basic Research In Computer Science (BRICS, Denmark), IRISA/INRIA Rennes, and at the Laboratoire d'Informatique Fondamentale de Lille (LIFL/CNRS, France). He is a keen chess player.

Pedro Peris-Lopez is Assistant Professor at the Computer Security Group of the Carlos III University, Madrid, Spain. He is M.Sc. Telecommunications Engineer. His current research interests are in the field of protocols design, cryptographic primitives, authentication, privacy, lightweight cryptography, etc. Nowadays, he is specially focusing on cryptographic protocols devoted to Radio Frequency Identification Systems (RFID). In these fields, he has published a great number of papers in specialized journals and conference proceedings.

Arturo Ribagorda is Full Professor at Carlos III University of Madrid, where he is also the Head of the Cryptography and Information Security Group and currently acts as the Director of the Computer Science Department. He has a M.Sc. in Telecommunications Engineering and a Ph.D. in Computer Science. He is one of the pioneers of computer security in Spain, having more than 25 years of research and development experience in this field. He has authored 4 books and more than 100 articles in several areas of information security. Additionally, he is member of the program committee of several conferences related to cryptography and information security.