On the use of Genetic Programming to develop cryptographic hashes

A. Torres-Vázquez, A. Ribagorda, B. Ramos

***Abstract*—Nowadays, hash functions are suffering a crisis due to the development of new attacks. Therefore, it is necessary to think about new construction schemes for hash functions, as well as new compression algorithms. In this paper, Genetic Programming is used to generate a compression function which will be applied to a scheme that also takes advantage of the recently proposed T-functions.**

***Index Terms*—Hash, Genetic Programming.**

I. Introduction

Cryptographic hashes have a crucial role in modern cryptology. Just like non cryptographic ones, these functions transform a message of arbitrary length in a small fixed-length bit sequence to *summarize* or to probabilistically identify a big amount of information. The message length doesn't affect the hash length; the message to be hashed can even be smaller than the hash itself.

This work is focused only on the study of cryptographic hashes [4]. They differ from the traditional ones because they are required to fulfil extra security properties. This fact allows them to be used in several security applications such as message authentication, data integrity or digital signatures. Within this context, the digest of a hash univocally identifies its input. In this way, if the hash value is sent together with the message, the receiver can check the integrity of the message by simply applying again the hash function. A main property of cryptographic hashes is that small changes in the input should cause completely different hashes.

This article is organized as follows: In section 2 we present a background in hashes including the basic definition and the main properties of cryptographic hashes, and an introduction to the Merkle Damgård scheme with a brief state of the art. Section 3 presents the new hash scheme we propose. Section 4 deeply studies the aspects concerning Genetic Programming (GP) and the way it is applied to develop our proposed compression function. In section 5, some implementation issues concerned with GP are carefully explained. Section 6 deals with some experimental results. In section 7 a security analysis of the whole scheme is carried out and finally, in section 8 some conclusions and future work are described.

Anna Torres-Vázquez is Associate Professor at Escuela Universitaria Cardenal Cisneros, Avda. Jesuitas, 34 28806 Alcala de Henares, Madrid. e-mail: anna.torres@cardenalcisneros.es

Professor Ribagorda is with the Computer Science Department, Carlos III University, Avda. Universidad, 30 28911 Leganes, Madrid. e-mail: arturo@inf.uc3m.es

Benjamín Ramos is Associate Professor at the Computer Science Department, Carlos III University, Avda. Universidad, 30 28911 Leganes, Madrid. e-mail: benja1@inf.uc3m.es

II. Background

A. Cryptographic Hashes

Formally, a hash F is defined as $F : D \rightarrow Im$, where $D = \{0,1\}^*$ is the domain and $Im = \{0,1\}^n$ is the image set [4]. Since F is a mapping between an arbitrary finite-length string and a fixed-length string of n bits and $|D| > |Im|$ (usually $|D| >> |Im|$), then F could not be one-to-one. This implies the existence of collisions can not be avoided, where a collision happens when a pair of distinct inputs result in the same output, i.e. $m_1 \neq m_2 \mid F(m_1) = F(m_2)$.

Every hash function should have the following two properties:

- Compression: F maps an input m of arbitrary finite length, to an output $F(m)$ of fixed bit-length n. That is, for any input length the hash value will always have the same length. Generally $|m| >> n$.
- Ease of computation: Given F and an input message m, $F(m)$ is easy to compute.

Furthermore, cryptographic hash functions must follow three more properties to grant the adequate security level:

- Preimage resistance: Given an output y, it is computationally infeasible to find any input

which hashes to that output, i.e. to find any preimage m' such that $F(m') = y$. That is, F should be a one-way function.
- Second preimage resistance: Given a message m_1, it is computationally infeasible to find another input message $m_2 \neq m_1$ such that $F(m_1) = F(m_2)$. A hash function satisfying this property is said to be weak collision resistant; two different inputs should not hash to the same value.
- Collision resistance: It is computationally infeasible to find any two distinct inputs m_1 and m_2 such that they hash to the same value $F(m_1) = F(m_2)$. In this case, the function is said to be strong collision resistant.

B. Merkle-Damgård processing

As previously mentioned, a hash function maps a message of arbitrary finite length to a digest of fixed length. Nowadays, the most popular way to construct hash functions is by following an iterated scheme in which a compression function is applied iteratively to generate the hash value.
Among the iterated schemes, the most widely used is the Merkle-Damgård processing ([3], [2]), illustrated in Figure 1. In this scheme, the message is divided in fixed-length blocks to which the compression function is applied sequentially. The last processed block contains information about the length of the input added during the padding stage (MD-Strengthening ([3], [2])), what increases the construction security. Very popular hashes as SHA-1 or MD5 follow this scheme.

C. State of the art and motivation

The scheme we propose arises after the crisis suffered by some of the most popular hash functions based on the Merkle - Damgård scheme. The vulnerabilities of actual hashes have increased due to attacks such as the Joux multi-collision attack [6], based on finding collisions in the intermediate states, or the Wang-Yin-Yu attack [5], which has broken MD5 and seriously compromised the security of SHA-1. As an answer to these attacks, new schemes are needed. Among other proposals, Lucks [7] has built a new scheme that makes the intermediate states bigger than the final hash, following a suggestion firstly made by Biham [14].

Another way of improving hash constructions is to develop better and stronger compression functions. A quite unexplored way to design compression functions is to use Evolutionary Computation paradigms. These approximations have been applied to develop hashes in the past. In particular, cellular automata have been used to build new hash functions ([24], [17], [18]). On the other hand, Genetic Programming has been recently applied to develop other cryptographic primitives such as block ciphers in [16]. Our proposal studies in depth the use of Genetic Programming to develop a new compression function.

III. A NEW HASH SCHEME

Our scheme processes a variable length message into a fixed-length output of 192 bits. Following Biham's suggestion [14], the internal state (consisting of 256-bits) is bigger than the hash output. The input message is broken up into chunks of 256-bit blocks; the message is padded so its length is divisible by 256. The padding is done following the MD-strengthening [3], [2]. The main algorithm divides the 256-bit internal state into six 32-bit words which are initialized with the same 8 first constants of SHA-1 [23]. Then, it operates on each 256-bit message block in turn, with the corresponding internal state modification. The processing of each message block consists of two main stages: the compression function and the use of Klimov T-functions [8], in order to modify and strengthen each internal state. The compression function, named V, has been created by means of Genetic Programming and it is explained in the next section. Klimov T-functions are permutations with very good statistical properties. For a map of n bits, let G be a T-function defined as

$$G(x) = x + (x^2 \vee C)(\bmod 2^n)$$

where C can be any constant as far as it ends with 101, which guarantees that the T-function will have a single cycle of length n . Our scheme starts with an initial state E_0 and follows the next steps during each round (see Figure 2):

for $i \in 1..L$ $\{$
$E_i^{'} = V(M_i) \oplus E_{i-1}$
$E_i =$ **T-function**$(E_i^{'})$
$\}$

being V the compression function and E_i the state matrix during the i-th round.
When the whole message is processed, and after a

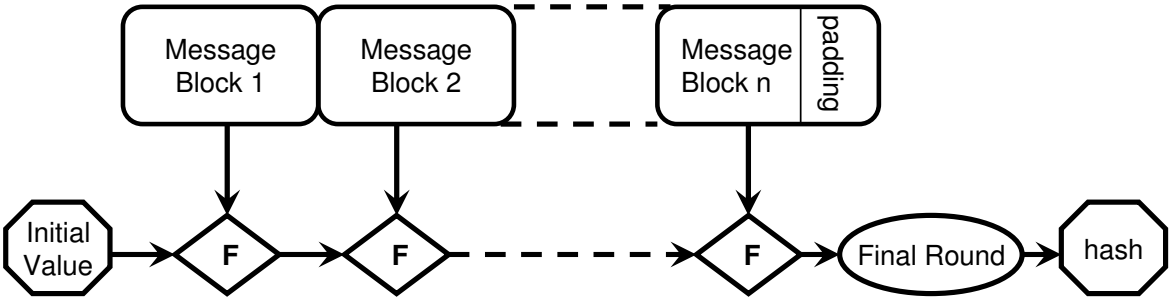

Fig. 1: Merkle-Damgård processing

padding stage, a final round is performed. It takes the most and the less significant words of the state and uses them iteratively as arguments of the T-function, then it makes a bitwise xor between both and with the state itself. Finally it calls the T-function again:

```
aux0[0]= T(state[0]);
aux7[0]= T(state[7]);
for (i=1; i<8; i++){
     aux0[i]=T(aux0[i-1]);
     aux7[i]=T(aux7[i-1]);
}
for (i=0; i<8; i++){
    aux[i]=aux0[i]^aux7[i]^state[i];
    state[i]=T(aux[i]);
}
```

Once the final round is executed, the size of the state is reduced from 256 to 192 bits by discarding the most and the less significant words. The obtained result is the final digest value.

IV. Genetic Programming

Genetic Programming (GP) consists of automatically evolving programs by means of Darwinian natural selection-based strategies. Its goal is the evolution of program populations, transmitting its heredity so that the new generations can adapt better to the environment.

Formally, Genetic Programming is a stochastic population-based search method guided by a fitness function. GP was devised in 1992 by John R. Koza [10]. It is inspired in Genetic Algorithms [9], being the main difference between them 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, it exists a main reason for using GP in this problem: A compression function is, in essence, a computer program, so its size and structure are not defined in advance. Thus, finding a flexible codification that can fit a GA is a difficult problem. Genetic Programming, nevertheless, does not impose restrictions to the size or shape of evolved structures. 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 (leaves). 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 nature, evolution takes place because there exists reproduction between the individuals of a population; it is a reproduction provided with heredity what permits the successors held better properties than their predecessors in terms of adaptation and survival. This survival is the real driving force of evolution, because the existence of finite resources causes a competition where those best adapted will be the ones with the higher probability of survival. 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.

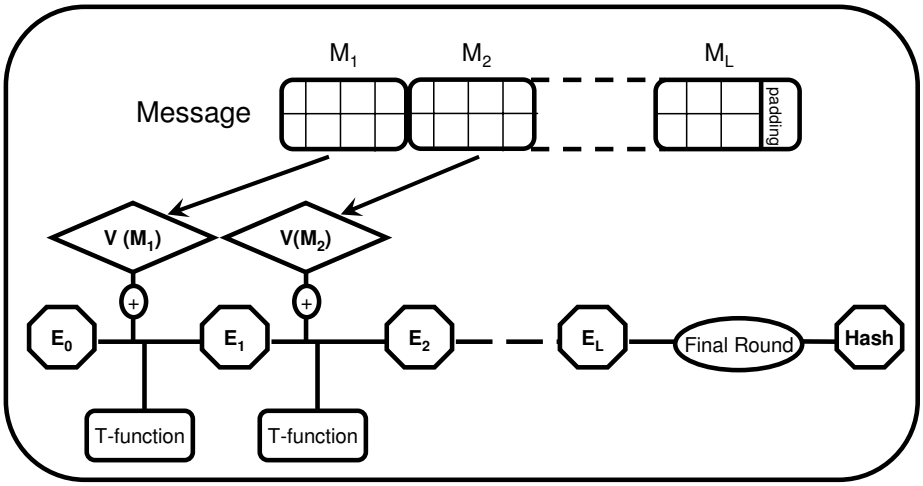

Fig. 2: Our proposal scheme

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

A. A fitness function for cryptographic hashes

According to section 2, a cryptographic hash is a way to probabilistically identify a message. Therefore, the quality of a cryptographic hash can be assessed to some extend by measuring the randomness of its output. With this purpose, there are tests batteries that measures entropy, the correlation coefficient, the mean, etc. A combination of these tests could be a good candidate for a fitness function. The problem of this choice is that hash functions don't need to pass more tests than those that form part of the fitness function. This fact can cause the function to find optimal (or almost optimal) values for the tests included in the fitness function, but to completely fail other tests which have not been included, even very simple ones.

Our proposal for the fitness function is completely different; instead of measuring the randomness of the output, we measure the non-linearity between the input and the output. Some aspects of nonlinearity can be measured by means of the avalanche effect. In this work, we use this property in the fitness function for the compression function we implement.

B. The Avalanche Effect

Nonlinearity can be measured in a number of ways or, what is equivalent, has not a complete unique and satisfactory definition. Fortunately, this is out of the scope of this paper, as we do not pretend to measure non-linearity, but a very specific mathematical property named avalanche effect. This property tries, to some extend, to reflect the intuitive idea of high-nonlinearity: a very small difference in the input producing a high change in the output, hence an avalanche of changes.

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

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

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

This definition also tries to abstract the more general concept of output independence from the input (and thus our proposal and its applicability to the generation of compression functions). Although it is clear that this independence is impossible to achieve (a given input vector always produces the same output), the ideal F will resemble a perfect random function where inputs and outputs are statistically unrelated. Any such F would have perfect avalanche effect, so it is natural to try to obtain such functions by optimizing the amount of avalanche. In fact, we will use an even more demanding property that has been called the Strict Avalanche Criterion [19] which implies the Avalanche Effect, and that could be mathematically described as:

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

This happens because the average of a Binomial distribution with parameters $1/2$ and n is $n/2$. The distance of a given probability distribution to another ($B(1/2, n)$ in this case) could be easily measured by means of a χ^2 goodness-of-fit test. That is exactly the procedure we will follow.

V. Implementation Issues

We have used the `lil-gp` genetic programming library [11] 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 the system for our purposes.

A. Function Set

As we have previously commented, problems in GP are represented using parse tree structures where functions are the intermediate nodes and terminals are the leaves. Firstly, we need to define the set of functions: This is critical for our problem, as they are the building blocks of the compression functions we would obtain. Being efficiency one of the paramount objectives of our approach, it is natural to restrict the set of functions to include only very efficient operations, both easy to implement in hardware and software. On the other hand we need to grant non-linearity in our compression function, so we will include some algebraic operations.

- **Binary operations**: Another, but minor, objective was to produce portable algorithms; so the inclusion of the basic binary operations are an obvious first step. Operations such as left and right rotations, and, xor, or and not are very efficient and easy to implement.
  - *xor* (Addition mod 2)
  - *and* (Binary Conjunction)
  - *or* (Binary Disjunction)
  - *not* (Binary Negation)

  The first two operators have a very important role when logical formulas are standardized and normalized. That is, when logical formulas are converted in their Algebraic Normal Form (ANF). On the other hand, we have noticed that in the experiments producing good individuals, the nodes *or and* and *not* have virtually disappeared during the evolutionary process. Intuitively, it could be thought that the best option would be to eliminate those nodes from the set of functions, but after some observations we conclude that results are much better when we allow the algorithm to rule out these functions than when we eliminate those nodes.

  - *rotd* (1 bit right rotation)
  - *roti* (1 bit left rotation)

  Previous works in Genetic Programming applied to cryptographic primitives, considered left and right rotation interchangeable, so they only used one of them (usually the right one) for their experimentation. This fact isn't exactly true, thus there is a type of cryptanalytic attack named cryptanalysis mod n [25], in which left and right rotations behave somehow in different ways. So including both rotations might increment the security of the resulting primitive.
- **Algebraic Operations** They are less efficient than the binary ones but they increase the non-linearity and thus the avalanche effect.
  - *sum* Sum mod 2^{32}
  - *mult* Multiplication mod 2^{32}

  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 [20] 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 the amount of avalanche we were trying to maximize. That is the reason why we finally introduced it in the function set. Anyway, there are many other cryptographic primitives that make an extensive use of multiplication, notably RC6 [21]

B. Terminal Set

The set of terminals in our case will be represented by eight 32-bit unsigned integers. (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) which provides an input of 256 bits to the compression function in each round. This choice has been made trying to obtain a trade-off between security and efficiency. We believe that nowadays 256 bits is a reasonable choice.

Finally, we included Ephemeral Random Constants (ERC's) for completing the terminal set. An ERC is a constant value that GP uses to generate better individuals (for a more detailed explanation on ERC's, see [10]). In our problem, ERC's are 32-bits random-values that behave as constants for the compression function 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.

C. Fitness Function

The objective of the fitness function is to evaluate the evolved algorithms to find within the whole set of compression functions obtained after the experimentation, those with best non-linearity properties. After some experiments with different fitness functions we found the one that best adapted to our requirements was

$$Fitness = \frac{1000000}{\chi^2}$$

So the fitness of every individual is calculated as follows: First, we use the Mersenne Twister generator [13] 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 R is stored. Then, we randomly flip one single bit of one of the eight input values, and we run again the algorithm, obtaining a new value nR. Now, we compute and store the Hamming distance $H(R, nR)$ between those two output values. This process is repeated a number of times (that can be fixed with the parameter $numexp$) and each time a Hamming distance among 0 and 32 is obtained and stored.

For a perfect Avalanche Effect, the distribution of this Hamming distances should adjust to the theoretical Bernoulli probability distribution $B(\frac{1}{2}, 32)$. Therefore, the fitness of each individual is calculated 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 $B(\frac{1}{2}, 32)$. Thus, our GP system tries to minimize the χ^2 statistic in order to maximize the expression for the fitness function shown above.

D. Experimentation Parameters

- **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 our fundamental objective was security, we decided to adopt a permissive attitude about the number of nodes, which we fixed in 100 to ensure a high degree of avalanche effect and a strong resilience against lineal and differential cryptanalysis. The depth of the tree was bounded by the number of nodes, because we thought it would be very useful to give some freedom to the system in order to achieve a better algorithm evolution.
- **Population size** This parameter has an important impact over the time the experiment is going to take. Even though, it is very important to increase, from the very first step, the genetic diversity, which is achieved raising the number of individuals, that is the possible programs from the population. After some experiments in which we increased the population size from 50 to 500, we could see that maintaining the rest of parameters with the same values, the results were considerably better when the population was bigger. Thus, even if each experiment would take much more time, we decided to fix the population size equal 500. It is to be said that the decrease of the running speed of each experiment does not affect the speed of the obtained algorithm.
- **Number of Generations** This parameter demands an agreement when fixed. A high number of generations will obviously delay each experiment, but on the other hand it will allow

a bigger evolution process because this is performed from generation to generation. Anyway, this evolution has an upper bound, from which the algorithm hardly evolves. After some experimentation changing the number of generations from 250 to 1500, we concluded that the balance between evolution and running time was semi-optimal around 1000 generations.

- **Individual evaluation parameter** When we talked about the fitness function, we mentioned a parameter that describes the number of 1 bit variations in one message carried out to evaluate the avalanche effect in each individual. This parameter called *numexp* fixes the reliable interval length for the fitness measure. Numexp has remarkable repercussions in terms of each individual evaluation's time, but if we want to grant some reliability in the obtained measures we should increase its value. As we did with the previous parameters, we should adopt an agreement, in this case, between evaluation costs and measure reliability. A small value would leave lots of bits without being changed, what would cause that some bad properties of the function with high repercussion in its security do not show up. On the other hand, fixing this parameter to a too big value, could affect the overall number of experiments to be run. A good agreement is achieved when numexp is set between 2048 and 8192, so we have fixed it to 4096.

VI. Experimentation and Results

We ran 30 experiments with different seeds ($seed_i = (\pi * 100000)^i \pmod{1000000}$) The result presented below was the best individual generated over these 30 experiments, with a population size of 500 individuals, a crossover probability of 0.8, and an ending condition of reaching 1000 generations.

A. Compression Function: V

The tree corresponding with the best individual found during the search of a compression function is shown in Figure 3. This algorithm has an avalanche effect of 16.0039 (nearly perfect), that is to say that when we change one input bit among the 256 bits, they will change, as average, half of the 32 bits of output. The fitness function resulting is 173966, and so the χ^2 has a value of approximately 5.77 which is an extremely low value for an statistic with 32 degrees of freedom.

Compression Function

```
=== BEST-OF-RUN ===
               generation: 786
                    nodes: 93
                    depth: 32
                     hits: 160039

TOP INDIVIDUAL:

-- #1 --
                     hits: 160039
              raw fitness: 173966.9419

TREE:

 (sum (sum (vroti (sum a1 (vroti a7)))
 (vrotd (sum a1 a0))) (sum (xor a6 a3)
 (sum (sum (vroti (vroti (vroti (vroti
 (mult 18106903 (vroti (vroti (vroti
 (mult 18106903 (sum (vroti (vroti (vroti
 (vroti (vroti (xor (mult (xor (vroti
 (vroti (xor (vroti (vroti (vrotd (xor
 (vrotd (xor a6 a3)) (sum (xor (sum a4 a0)
 (vroti a7)) (xor a5 a2)))))) a1)))
 (sum a4 a0)) 18106903) (xor a5 a2)))))))
 (xor (sum (xor (vroti a7) a1) (vroti a7))
 (xor a6 a3))))))))))) (vrotd (xor (vrotd
 (sum a1 (vroti a7))) (sum (sum 9f10b70
 (vroti a7)) (xor a5 a2))))) a2)))
```

Fig. 3: Best Individual

The individual obtained has a behaviour that exceeds the expectations we had about the avalanche effect and the value of the statistic χ^2. Furthermore, it is quite efficient, thus although it contains multiplications, there are only 3 of them, and one of its arguments is always a constant. Efficiency is achieved decreasing the number of nodes (set to 100 at the beginning). Not only the system has returned a function of 93 nodes (< 100) but also this function can be easily optimized because we can suppress one left rotation followed by one right rotation (their effects are neutralized) improving then the number of nodes to 91.

VII. Security Analysis

We have performed a preliminary security analysis of the scheme, consisting of examining the statistical properties of its output over a low-entropy input. In our case, we have set the input to a fixed-length string of 2048 bytes obtaining each byte by making a bitwise or between to random numbers and casting the result to a char. Following this scheme, we have generated a file of 312500 KB to be analyzed with the most demanding battery of statistical tests for cryptographic purposes: the NIST battery [26]. The results obtained are presented in Table I. In those cases where the same test is executed more

TABLE I: Nist Battery of statistical tests

Statistical Test	Proportion of Success
frequency	1.0000
block-frequency	1.0000
cumulative-sums	1.0000
runs	1.0000
longest-run	0.9900
rank	1.0000
fft	0.9900
nonperiodic-templates	0.99027027
overlapping-templates	0.98
universal	0.99
apen	1
random-excursions	0.981875
random-excursions-variant	0.987111111
serial	0.98
linear-complexity	0.99

than once, we present the average of the success proportion obtained.

In the light of the results shown in Table I, we can conclude that the output successfully passed The Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Application, even over a low entropy input.

Regarding speed, our scheme has been preliminary tested under an AMD Athlon XP processor at 2100 Mhz. In Table II it is shown the rate of MB/s processed, as well as a speed test for MD5. As we can see, MD5 processes a 20.6% more MB/s than our scheme, what is a very good result knowing that the internal state of our scheme doubles that of MD5.

TABLE II: Speed test

	MD5	Our Proposal
MB/s	45.8	36.4

VIII. Conclusions and Future Work

The results shown in the previous section demonstrate that Genetic Programming can be successfully applied to design competitive hash functions. In this line, V can be thought as an instance of a family of designs that can be explored within this paradigm.
In future works, Genetic Programming can be further explored to find good compression functions that fit the needs of distinct hash schemes. Another open field could be the search of alternatives to non-linearity in order to measure the fitness of those functions created with Genetic Programming.
In this paper, we have presented the design an implementation of a quite fast and secure scheme developed with the ease that Genetic Programming offers. So, we have proved Genetic Programming is a good alternative to classical approaches to develop cryptographic hashes.

References

[1] G. Yuval, *How to swindle Rabin*. Cryptologia, 1979 v.3 pp. 187-191.

[2] I. B. Damgård, *A Design Principle for Hash Functions*. Advances in Cryptology-CRYPTO'89, LNCS 435, 1990, pp. 416-427.

[3] R. C. Merkle, *One Way Hash Functions and DES*. Advances in Cryptology-CRYPTO'89, LNCS 435, 1990, pp. 428-446.

[4] A. Menezes and P. C. van Oorschot and S. A. Vanstone, *Handbook of applied cryptography*. CRC Press, 1996.

[5] X. Wang and L. Yin and H. Yu, *Finding Collisions in the Full SHA-1*. Advances in Cryptology-CRYPTO'05, LNCS 3621, 2005, pp. 17-36.

[6] A. Joux, *Multicollisions in iterated hash functions, application to cascaded constructions*. Advances in Cryptology-CRYPTO'04, LNCS 3152, 2004, pp. 306-316.

[7] S. Lucks, *Design Principles for Iterated Hash Functions*. Cryptology ePrint Archive, Report 2004/253, 2004. http://eprint.iacr.org/

[8] A. Klimov and A. Shamir, *A New Class of Invertible Mappings*. CHES'02, 4th Workshop on Cryptographic Hardware and Embedded Systems, LNCS 2523, 2003, pp. 470-483.

[9] J. H. Holland, *Adaptation in Natural and Artificial Systems*. MIT Press, 1992.

[10] J. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[11] *Lil-gp Genetic Programming system*. http://garage.cse.msu.edu/software/lil-gp/index.html

[12] J. Walker, *ENT: A Pseudorandom Number Sequence Test Program*. http://www.fourmilab.ch/random/

[13] M. Matsumoto and T. Nishimura, *Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator*. ACM Trans. on Modeling and Computer Simulation, v. 8, 1998, pp.3-30.

[14] E. Biham, *Recent advances in Hash Functions: The way to go*. Conference on Hash Functions (Ecrypt Network of Excellence in Cryptology), June 23-24, 2005, Poland. http://www.ecrypt.eu.org/stvl/hfw/Biham.ps

[15] C. Estbanez and J. C. Hernndez-Castro and P. Isasi and A. Ribagorda, *Evolving hash functions by means of Genetic Programming*. GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation, 2006.

[16] J. C. Hernández-Castro and J. M. Estévez-Tapiador and A. Ribagorda and B. Ramos, *Wheedham: An automatically designed block cipher by means of Genetic Programming*. Proceedings of CEC06., 2006, pp. 192199.

[17] M. Mihaljevic and Y. Zheng and H. Imai, *A cellular automaton based fast one-way hash function suitable for hardware implementation*. LNCS 1431, 1998, pp. 217-233.

[18] S. Hirose and S. Yoshida, *A one-way hash function based on a two-dimensional cellular automaton*. The 20th Symposium on Information Theory and Its Applications (SITA97), v.1, 1997, pp.213-216.

[19] R. Forré, *The strict avalanche criterion: spectral properties of boolean functions and an extended definition*. CRYPTO'88, LNCS 403, 1990, pp. 450-468.

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

[21] R. L. Rivest and M. J. B. Robshaw and R. Sidney and Y. L. Yin, *The RC6 Block Cipher. v1.1*. AES Proposal, 1998.

[22] R. L. Rivest, *The MD5 Message Digest Algorithm*. RFC 1321. April 1992.

[23] D. E. Eastlake and P. E. Jones, *US Secure Hash Algorithm 1 (SHA1)*. RFC 3174, September 2001.

[24] J. Daemen and R. Govaerts and J. Vandewalle, *A Framework for the Design of One-Way Hash Functions Including Cryptanalysis of Damgård's One-Way Function Based on Cellular Automata*. Advances in Cryptology - Asiacrypt'91, LNCS 739, 1993, pp.82-96.

[25] J. Kelsey and B. Schneier and D. Wagner, *Mod n Cryptanalysis, with applications against RC5P and M6*. Proceedings of the Sixth Fast Software Encryption Workshop, LNCS 1636, 1999, pp. 139-155.

[26] A. L. Rukhin and J. Soto and J. R. Nechvatal and M. Smid and E. B. Barker and S. Leigh and M. Levenson and M. Vangel and D. Banks and N. A. Heckert and J. F. Dray, *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.