Annales UMCS Informatica AI XII, 3 (2012) 37–53
DOI: 10.2478/v10065-012-0027-x

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI
http://www.annales.umcs.lublin.pl/

Hash function generation by means of Gene Expression Programming

Sébastien Varrette[1*], Jakub Muszyński[1†], Pascal Bouvry[1‡]

[1] *University of Luxembourg, Computer Science and Communications (CSC) Research Unit, 6, rue Richard Coudenhove-Kalergi, L-1359 Luxembourg, Luxembourg*

Abstract – Cryptographic hash functions are fundamental primitives in modern cryptography and have many security applications (data integrity checking, cryptographic protocols, digital signatures, pseudo random number generators etc.). At the same time novel hash functions are designed (for instance in the framework of the SHA-3 contest organized by the National Institute of Standards and Technology (NIST)), the cryptanalysts exhibit a set of statistical metrics (propagation criterion, frequency analysis etc.) able to assert the quality of new proposals. Also, rules to design "good" hash functions are now known and are followed in every reasonable proposal of a new hash scheme. This article investigates the ways to build on this experiment and those metrics to generate automatically compression functions by means of Evolutionary Algorithms (EAs). Such functions are at the heart of the construction of iterative hash schemes and it is therefore crucial for them to hold good properties. Actually, the idea to use nature-inspired heuristics for the design of such cryptographic primitives is not new: this approach has been successfully applied in several previous works, typically using the Genetic Programming (GP) heuristic [**1**]. Here, we exploit a hybrid meta-heuristic for the evolutionary process called Gene Expression Programming (GEP) [**2**] that appeared far more efficient computationally speaking compared to the GP paradigm used in the previous papers. In this context, the GEPHashSearch framework is presented. As it is still a work in progress, this article focuses on the design aspects of this framework (individuals definitions, fitness objectives etc.) rather than on complete implementation details and validation results. Note that we propose to tackle the generation of compression functions as a multi-objective optimization problem in order to identify the Pareto front *i.e.* the set of non-dominated functions over the four fitness criteria considered. If this goal is not yet reached, the first experimental results in a mono-objective context are promising and open the perspective of fruitful contributions to the cryptographic community.

*sebastien.varrette@uni.lu
†jakub.muszynski@uni.lu
‡pascal.bouvry@uni.lu

1 Introduction

Cryptographic hash functions are fundamental primitives in modern cryptography and are of crucial importance for our digital life. In particular, they are used in many security applications (data integrity checking, cryptographic protocols, digital signatures, pseudo random number generator etc.). Formally speaking, a hash function $H : \{0,1\}^* \longrightarrow \{0,1\}^n$ maps a binary string of arbitrary length into a binary string of some fixed length n (often called footprint) with at least the *compression* and *ease of computation* properties [**3**]. If hash functions satisfy the additional requirements such as *preimage resistance*, *second preimage resistance* and most importantly *collision resilience*, they are a very powerful tool in the design of techniques to protect the authenticity of information. Recent advances in hash functions cryptanalysis permitted successful attacks against the major cryptographic hash function in use, including the well-known MD5 [**4**] and SHA-1 [**5**] hash schemes which are therefore no longer considered as secured. In response, the National Institute of Standards and Technology (NIST) recommended to move to the SHA-2 family of hash functions (SHA-224, SHA-256, SHA-384 and SHA-512) and the SHA-3 contest has been launched[1] to find new schemes. At the time of writing, the third and last round of this contest has been released and permits to exhibit five candidates for the next SHA-3 standard. At the same time novel hash functions are designed (especially in the framework of the SHA-3 contest), the cryptanalysts exhibit a set of statistical metrics (propagation criterion, frequency analysis etc.) able to assert the quality of new proposals.

This article investigates the ways to build on this experiment and those metrics to generate automatically the compression functions by means of Evolutionary Algorithm (EA). Such functions are at the heart of the construction of iterative hash schemes and it is therefore crucial for them to hold good properties. Actually, the idea to use nature-inspired heuristics for the design of such cryptographic primitives is not new: this approach has been successfully applied in several previous works, typically using the GP heuristic [**1**]. Here, we exploit a hybrid meta-heuristic for the evolutionary process called GEP [**2**] that appeared far more efficient computationally speaking compared to the GP paradigm used in the previous papers. In this context, the GEPHashSearch framework is presented.

This paper is organized as follows: section 2 presents the background of this work (cryptographic hash functions, EAs) and reviews the related works. Section 3 provides a brief overview of the Gene Expression Programming (GEP) heuristic while section 4 holds the main contribution of this paper. It details GEPHashSearch, a GEP–based framework to build the compression functions with reasonably good properties. This is still a work in progress such that section 5 remains limited. Yet the first experimental results which are proposed there are still promising. Finally, section 6 concludes the paper and provides the future directions.

[1] See `http://csrc.nist.gov/groups/ST/hash/sha-3/`.

2 Context & Motivations

2.1 Cryptographic hash functions

Cryptographic hash functions are fundamental primitives in modern cryptography and have many security applications (data integrity checking, cryptographic protocols, digital signatures, pseudo random number generators etc.). Formally speaking, a hash function $H : \{0,1\}^* \longrightarrow \{0,1\}^n$ maps a binary string of arbitrary length into a binary string of some fixed length n (often called *footprint* or *fingerprint*) with at least the *compression* and *ease of computation* properties [**3**].

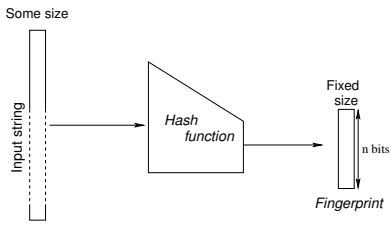

Fig. 1. Principle of a hash function

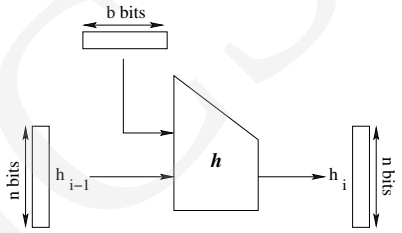

Fig. 2. Compression function of a hash function

To be of cryptographic use, hash functions must satisfy the additional properties, such as *preimage resistance*, *second preimage resistance* and most importantly *collision resilience*. One talks about a *collision* between x and x' when $x \neq x'$ and $H(x) = H(x')$. Considering that the input of a hash function can be of any size (in particular $> n$), collisions are unavoidable. Knowing that if y is such that $y = H(x)$, then x is called the *preimage* of y, the above mentioned properties can be defined as follows[2]:

- **Preimage resistance**: given y, one can not find - in *reasonable* time - some x such that $y = H(x)$. Given y, 2^n computations are at most required for finding x.
- **Second preimage resistance**: given x, one can not find - in *reasonable* time - $x' \neq x$ such that $H(x) = H(x')$. As above, given x, 2^n evaluations are at most required for finding y.
- **Collision resistance**: one can not find in *reasonable* time x and x' such that $H(x) = H(x')$. Note that there is a free choice of both inputs. There are $2^{\frac{n}{2}}$ evaluations required to find a valid couple (x, x') (This result comes from the Birthday paradox).

The additional properties are often desired:

- *non-correlation*: input and output bits should not be correlated. Related to this, an *avalanche effect* property similar to the one of good block ciphers is required: modification of a single bit in the input should change at least half

[2]In these definitions, *reasonable* means that there exists no attack that operate faster than the exhaustive or brute-force search among all possible inputs.

of the output bits. This rules out hash functions for which preimage resistance fails to imply the 2nd-preimage resistance simply due to the function effectively ignoring a subset of input bits.

- *near-collision resistance*: it should be hard to find any two inputs (x, x') such that $H(x)$ and $H(x')$ differ in only a small number of bits.
- *partial-preimage resistance* or *local one-wayness*. It should be as difficult to recover any substring as to recover the entire input. Moreover, even if part of the input is known, it should be difficult to find the remainder (e.g., if t input bits remain unknown, it should take on the average 2^{t-1} hash operations to find these bits.)

In practice, most arbitrary-length hash functions are built in the iterative process based on a fixed-length compression function or a block cipher. For instance, SHA-1 [**5**], MD5[**4**], as well as all the other hash functions we know, are constructed by applying some variant of the Merkle-Damgård construction to an underlying compression function $h : \{0,1\}^b \times \{0,1\}^n \longrightarrow \{0,1\}^n$ (see Figure 2). The general model for the iterated hash functions operates as follows: the hash input x of the arbitrary finite length is first split into fixed-sized chunks $x_1, x_2, \ldots x_L \in \{0,1\}^b$ which gives the expanded message $(x_1, \ldots, x_L)$. An iterated hash H iterates the underlying compression function h as follows:

$$h_0 = IV;\ \{h_i = h(h_{i-1}, x_i)\}_{1 \le i \le L};\ H(x) = g(h_L). \tag{1}$$

h_{i-1} serves as the n-bit chaining variable between the stage $i-1$ and the stage i. h_0 is a predefined starting value or initializing value (IV). An optional output transformation g is used in a final step to map the n-bit chaining variable to an n-bit result $g(h_L)$; g is often the identity mapping $g(h_L) = h_L$. The one or two last chunks of the expanded message are padded, and the last chunk x_L may contain the additional information, such as the length $\|x\|$ of the non-expanded message x.

Incremental hashing. Alternative constructions make use of *incremental hashing*: The idea behind the method presented in [**6**] is that if one changes a message and one has already computed the hash function of the (unchanged) message, one only needs to recompute the changed part in order to obtain the new hash value. This is done as follows: each chunk x_i of the message $x = x_1 x_2 \ldots x_L$ is prefixed by a block index. Then the hash value $y_i = h(\langle i \rangle . x_i)$ is computed and the output is combined via a special operation: $y = y_1 \odot y_2 \odot ... \odot y_L$.
The computation of h can be done by any standard hash function as soon as it is collision free. With this technique the computation of the hash value can be parallelized, therefore it is also possible to recompute only parts x_i of the message. In all cases, a good choice for the combining operation $\odot$ is crucial. A first natural thought is to use the bitwise XOR, but it was shown that it is insecure [**6**]. Using multiplication or addition in a group $\mathcal{Z}_p^*$ (multiplication and addition modulo p) is generally seen as a good choice for the combining operation (in this case, the incremental hashing scheme is called MuHASH and AddHASH where, for security reasons, $|p|$ should have at least

512 or 1024 bits, making the final hash value of the same length. If a smaller length is required, the output y can be hashed with a standard collision-free hash function (For instance, applying SHA-1 to MuHASH leads to a 160 bit fingerprint). The security depends on the discrete logarithm problem in the underlying group. As we will see in the sequel, the EA–based heuristic presented in this paper derives from the incremental hashing scheme in the way *linking functions* are used to connect the genes of the GEPHashSearch individuals.

2.2 Evolutionary Algorithms (EA)

EA is a class of solving techniques based on the Darwinian theory of evolution [**7**] which involves the search of a *population* X_t of solutions. Members of the population are feasible solutions and called *individuals*. Each iteration of an EA involves a competitive selection that weeds out poor solutions through the *evaluation* of a fitness value that indicates the quality of the individual as a solution to the problem. The evolutionary process involves at each generation a set of stochastic operators that are applied on the individuals, typically recombination (or cross-over) and mutation. Execution of simple EA requires high computational resources in the case of non-trivial problems, in particular the evaluation of the population is often the costliest operation in EAs. There exists many useful models of EAs, yet a pseudo-code of a general execution scheme is provided in Algorithm 1.

Algorithm 1. General scheme of an EA in the pseudo-code.
$t \leftarrow 0$;
Generation(X_t); // *generate the initial population*
Evaluation(X_t); // *evaluate population*
while Stopping criteria not satisfied
  $\hat{X}_t \leftarrow$ ParentsSelection(X_t); // *select parents*
  $X'_t \leftarrow$ Modification($\hat{X}_t$); // *cross-over + mutation*
  Evaluation(X'_t); // *evaluate offspring*
  $t \leftarrow t + 1$;
end while

2.3 Hash function generation by means of EAs

The idea to use nature-inspired heuristics for the design of cryptographic primitives and in particular hash functions is not new: this approach has been successfully applied in several previous works as reviewed in [**8**]. This probably started to attract attention of the researchers in the 90's with the Ph.D. Thesis of Clark [**9**] where different heuristic techniques (genetic algorithms, simulated annealing, and tabu search) were compared to break classical cryptosystems. In particular, the use of simulated annealing was proposed in the cryptanalysis of a certain class of stream ciphers. As mentioned in [**8**], Millan, Clark, and Dawson then additionally proposed a model for the generation of the Boolean functions with excellent cryptographic applications, thus starting a very

fruitful research line and showing these techniques could also help in cryptography, not only cryptanalysis. At the level of hash functions, the work of Daemen & al. in [**10**] opens the research area for the generation of hash functions by means of evolutionary technics (in this paper, a Cellular Automata (CA)) as initiated by the seminal work of Damgård in [**11**]. For instance in [**12**], an evolutionary technique was applied for the design of a digital circuit which computes a simple hashing function. Based on the FPGA architecture, the circuit was synthesized automatically through simulated evolution. More recently in [**13**], the authors used Genetic Algorithms (GAs) to construct Universal Hash Functions to efficiently hash a given set of keys. The Hash Functions generated in this way provide a lesser number of collisions as compared to selecting them randomly from a family of Universal Hash Functions. The proposed algorithm could be used in the scenarios where the input distribution of keys frequently changes and the hash function needs to be modified often to rehash the values to reduce collisions. Finally, the use of the Genetic Programming (GP) heuristic was successfully applied in [**14**] for the automated design of cryptographic block ciphers and hash functions. In this article, we extend the work proposed in [**14**] by exploiting a hybrid meta-heuristic for the evolutionary process called GEP [**2**] that appeared far more efficient computationally speaking compared to the GP paradigm used in the previous papers. The next section details this heuristic. Also, while the work presented in [**14**] focuses on a single objective (the avalanche effect captured by the propagation criteria $PC_k(t)$ presented in §4.1), we add a set of complementary objectives to direct the traversal of the search space toward solutions which not only optimize the propagation criteria but also the randomness, the complexity and the efficiency of the evolved compression functions.

3 Gene Expression Programming (GEP) heuristic

Among different classes of EAs, John Koza in [**1**] proposed to use GA in the so called Genetic Programming (GP) where the individuals represent a function or a program. Gene Expression Programming (GEP) was proposed by Cândida Ferreira in [**2**] as an extension of GP. As an EA, GEP uses the populations of individuals, selects the individuals according to their fitness, and introduces genetic variation using one or more genetic operators. The fundamental difference between these three classes of EAs resides in the nature of the individuals:

- in GAs the individuals are symbolic strings of fixed length (chromosomes);
- in GP the individuals are nonlinear entities of different sizes and shapes called parse trees that represent a program (see Fig. 3);
- in GEP, the individuals are also nonlinear entities of different sizes and shapes (expression trees), but these complex entities are encoded as simple strings of the fixed length (chromosomes).

This avoids possible divergence in the size of the parse trees that can be observed within GP and represents the main weakness of this approach: a large number of computational resources can be used to edit huge illegal structures. On the contrary,

the chromosomes in GEP are simple entities: linear, compact, relatively small, easy to manipulate genetically (replicate, mutate, recombine, transpose, etc.). In addition, any modification made in the genome always results in syntactically correct expression trees or programs. Hence, GEP has been chosen as the base of heuristic for finding good candidates for hash functions (or more precisely, compression functions).

Overview. In GEP, the genome or chromosome consists of a linear, symbolic string of the fixed length composed of one or more genes. This string contains two kinds of symbols: functions (which are typically elements from the set Δ_f) and terminals (belonging to a set Δ_{term}). Each gene represents a relation between the function and one can represent it as a tree called *Expression Tree (ET)*. The string reflects, in fact, the traversal of the tree from the left to the right and from the top to the bottom, which is neither the prefix nor the postfix traversal sometimes found in the GP implementations. For instance, let us assume that $\Delta_f = \{\sqrt{}, +, -, *\}$ and $\Delta_{term} = \{a, b, c\}$. Fig. 3 provides the gene and the expression tree associated to the algebraic expression $(a+b) * \sqrt{b-c}$. Here, the gene starts at position 0 and ends at position 7.

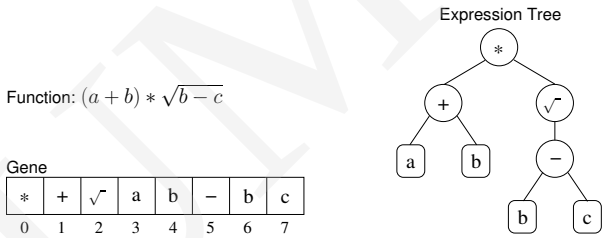

Fig. 3. Gene and expression tree associated to the function $(a+b) * \sqrt{b-c}$

In fact, the genes in GEP are composed of a head (containing both functions and terminals) together with a tail containing only terminals. Let h (resp. t) be the size of the head (resp. the tail). Then GEP encodes each gene with a string of the length $h+t$. More precisely, let a_{max} be the maximum number of arguments taken by the functions in Δ_f, then $t = h(a_{max}-1)+1$. Note that in the previous example, $a_{max} = 2$ meaning that $t = h+1$ and each gene is encoded with a string of the length $2h+1$. For instance, let us take $h = 7$. Then the complete gene considered previously could be written in GEP as depicted in Fig. 4.

In this case, the expression tree finishes at position 7 whereas the gene ends at position 14. If a mutation occurs at position 4 that changes 'b' into '+', then the gene presented in Fig. 5 is obtained, where the corresponding expression tree is also provided. It now finishes at position 9. So despite its fixed length, each gene has the potential to code for the expression trees of different sizes and shapes.

The way of cross-over operation is similar. Finally, GEP chromosomes are usually composed of more than one gene of equal length. For each problem or run, the number of genes, as well as the length of the head, is chosen. Each gene then codes the sub-ET

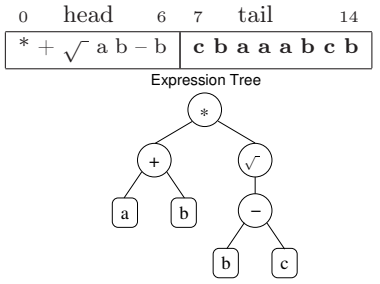

Fig. 4. Original gene: $(a + b) * \sqrt{b - c}$

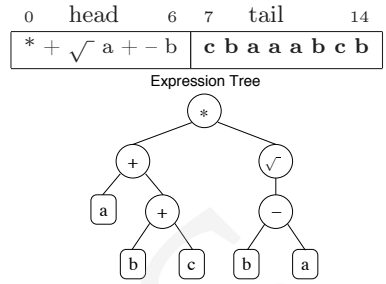

Fig. 5. Mutation example: $(a + b + c) * \sqrt{b - a}$

and the sub-ETs interact with one another forming a more complex expression tree. Detailing more closely GEP is clearly out of the topic for this paper (for this purpose, please refer to [**15**]).

4 GEPHashSearch: a GEP-based framework to find "good" compression functions

This article presents GEPHashSearch, a framework designed to evolve a population of individuals that are candidates for a compression function used in an iterative or incremental hash scheme. The objective is to design basic blocks of sufficiently good quality to expect that the global scheme remains a strong candidate. Note that many modern Hashes such as the current NIST finalists for the SHA-3 competition have a preprocessing stage; this is to prevent attacks that take advantage of a greater freedom of input (as opposed to being forced to append the length of the message). Some Hashes have expansion functions in addition to that prior to hashing. GEPHashSearch does neither of these as it simply focuses on the compression functions.

Table 1. Δ_f set of basic building operators used in GEPHashSearch

Operator	**Description**	**# Args**	**Example/Comment**
~	negates the input	1	~01=10
&	bitwise AND	2	0110 & 1100 = 0100
|	bitwise OR	2	0110 | 1100 = 1110
^	bitwise XOR	2	0110 ^ 1100 = 1010
$\gg_k$	cyclic right rotation of k bits	1	$10110 \gg_3 = 11010$
$\ll_k$	cyclic left rotation of k bits	1	$10110 \ll_3 = 10101$
$\texttt{addmod}_p$	Addition modulo p	2	$1011\ \texttt{addmod}_8\ 1100 = 0111$
$\texttt{multmod}_p$	Multiplication modulo p	2	$1011\ \texttt{multmod}_8\ 1100 = 0100$

Basic building operators. As we saw in §2.1, contemporary hash schemes consist in a set of binary operators combined together to form a more complex function applied in the different stages of the hash function evaluation. In GEPHashSearch, the basic building operators under considention are described in Table 1. They will form the set of functions Δ_f used by the GEP individuals in their head. The choices we made were governed by the operators generally used in the known hash schemes. Table 2 presents an overview of the basic operators used in MD5, SHA-1 and the finalists of the current SHA-3 hash competition. In the framework of GEPHashSearch, we focus on the cheapest operators (in terms of gates equivalents and required cycles to compute).

Table 2. Basic building operators used in MD5, SHA-1 and the five finalists of the current SHA-3 challenge (apart from permutations and Sboxes).

MD5 [**4**]	$\oplus$	&	|	~	`swap`	
SHA-1 [**5**]	$\oplus$	&	|	~	$\gg_5$	$\gg_{30}$ div
BLAKE [**16**]	$\oplus$	$\gg_k$	$\ll_k$	$\texttt{addmod}_{2^{32}}$	$\texttt{addmod}_{2^{64}}$	
Grøstl [**17**]	$\oplus$	trunc_k	Four AES operators on $\mathcal{F}_{256}$			
JH [**18**]	$\oplus$	&	|	~		
Keccak [**19**]	$\oplus$	&	|	~	$\gg_k$	
Skein [**20**]	$\oplus$	$\texttt{addmod}_{2^{64}}$	$\gg_k$			

GEP Individuals. In GEPHashSearch, each GEP individual $\texttt{Ind}_i$ that composes the population to evolve is assumed to be a potential candidate for a compression function $\texttt{compress}_i$ that intervenes in the iterative construction of a hash function H. This function is assumed to take two input parameters: a message chunk M of b bits and a state bloc S on n bits (corresponding to the fingerprint generated in the previous stage). This setup is illustrated in Fig. 6. The associated ET corresponds to the expression of $\texttt{compress}_i$ as a function composed by the basic building operators mentioned in Table 1 that form the set Δ_f. Remember that the objective of our work is to generate "good" candidates for the compression functions hoping that it will lead to good hash functions. In this context, the considered individuals are constrained by the following parameters:

- $n = 32 \times N$: size of the fingerprint, seen as N words *i.e.* N blocks of 4 bytes;
- $b = 32 \times B$: size of the successive message chunks, seen as B words;
- `head`: size of the head of the GEP chromosome.

The set of terminals Δ_{term} are all considered as word values corresponding to a block within the fingerprint or the message chunks. They are labelled accordingly as follows: $\Delta_{term} = \{b_1, \ldots, b_N\} \cup \{m_1, \ldots, m_B\}$. Also, rather than having a single GEP gene per individual, we setup N genes (to characterize the N threads responsible for each part of the state block $\{b_i\}_{1 \leq i \leq N}$). These genes are linked together by so-called *linking functions* that belongs to the set Δ_f and evolve throughout the generations. Finally, at every evaluation of an individual `Ind`, we compute a *terminal diversity* metric $T_D(\texttt{Ind})$

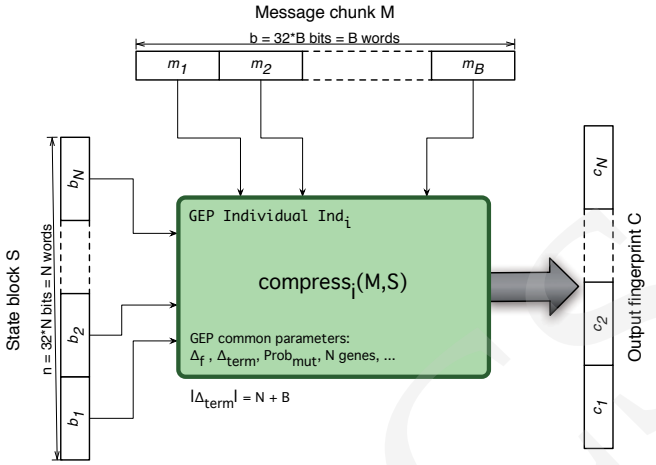

Fig. 6. GEP individual in the GEPHashSearch framework.

defined as the number of *distinct* terminals that are really expressed in the individual (*i.e.* that belong to the ET) divided by the total number of terminals *i.e.* $|\Delta_{term}| = N + B$. In particular, it is crucial that $T_D(\texttt{Ind}) = 1$ for the final individual chosen at the end of a GEPHashSearch process, meaning that none of the input blocks of the compression function (whether from the fingerprint or the message chunks) are ignored. Now that the GEPHashSearch individuals are defined, we can detail the four fitness functions used to evaluate them.

4.1 Fitness functions used in GEPHashSearch

Propagation criteria $PC_k(t)$. The propagation criterion $PC(t)$ has been proposed in [**21**] as a statistical test to check the effect of some bit flips on the output. Given a function f to evaluate, some random input x is generated and the result $y = f(x)$ is computed. Then the input is changed slightly into x' such that up to t random bits are flipped. In other words, assuming $d_H(.,.)$ denotes the Hamming distance, $d_H(x, x') \leq t$. Then the function is evaluated again to obtain $y' = f(x')$. It is expected that for a "good" function, each bit in the output is modified with probability $\frac{1}{2}$, thus the hamming distance $d_H(y, y')$ is expected to follow the Binomial law $\mathcal{B}(\frac{1}{2}, n)$ where n is the length of y (or y'). The χ^2 distribution (*i.e.*, a left skewed curve) is used to compare the goodness-of-fit of the observed frequencies of the k sample measures $\{d_H(y_i, y'_i)\}_{0 \leq i < k}$ to the corresponding expected frequencies of the hypothesized distribution (*i.e.* $\mathcal{B}(\frac{1}{2}, n)$).

If the successive values $d_H(y_i, y'_i)$ respect a binomial distribution, then this results in a low χ^2 value that permits to conclude that the tested function has good propagation proprieties. This χ^2 value is the measured fitness of the tested function and is referred to as $PC_k(t)$.

Randomness criteria RC**.** Apart from having good propagation properties, GEPHashSearch considers also the random nature of the fingerprint generated by the evaluated individuals. In this perspective, GEPHashSearch makes use of the NIST statistical Test Suite [**22**]. From a global perspective, this statistical package consists of 15 tests $\mathcal{T}_1, \ldots, \mathcal{T}_{15}$ that were developed to test the randomness of (arbitrarily long) binary sequences produced by either hardware or software based cryptographic random or pseudorandom number generators. Each of the tests $\mathcal{T}_i$ operates on a sufficiently long binary sequence s as an input, and computes a specific P-value which, when compared to the selected level of confidence α (for instance, $\alpha = 0.01$ (1%)), permits to conclude whether the sequence is non-random (P-value $< \alpha$) or if, on the contrary, it can be considered as random (P-value $\geq \alpha$). In GEPHashSearch, these tests are used to compute for each individual Ind_i at a given generation t the randomness criteria RC defined as a result of the function described in algorithm 2.

Algorithm 2. Randomness criteria evaluation RC for a given GEP individual.

Require: `Ind`, the GEP individual to test against the randomness criteria.
Require: l, the length of the sequence to pass to the NIST tests
Require: $\Delta_x = \{x_1, \ldots, x_k\}$, a set of random message chunks / $\forall i, |x_i| = b$ bits and $k \times n \geq l$
Require: IV, a fixed state block of n bits, used for the evaluation of all compression functions
Require: α, the level of confidence to apply to each NIST test
Ensure: $0 \leq RC \leq 15$
  function RC(`Ind`, l, Δ_x, IV, α)
    $RC \leftarrow 0$; $s \leftarrow$ ""; ▷ At worst, $RC = 0$. Initialize s as an empty string.
    while $|s| < l$ bits ▷ Generates the sequence s of at least l bits.
      $y_i \leftarrow$`Ind`(IV, x_i); ▷ Fingerprint of the message chunk x_i.
      $s \leftarrow s \| y_i$ ▷ append y_i (of n bits) to s.
    end while
    for $i \leftarrow 1..15$ ▷ Test s randomness against each NIST test.
      $RC \leftarrow RC + \mathcal{T}_i(s, \alpha)$; ▷$\mathcal{T}_i(s, \alpha) = 1$ if $\mathcal{T}_i$ concludes that s is random, 0 otherwise.
    end for
    return RC
  end function

To permit a fair comparison for this criteria of all GEP individuals, we assume that the set of random message chunks Δ_x used to build the bit-string sequence to be evaluated by the NIST tests is constructed at the beginning of each generation and is kept the same for all RC evaluations. Also, as these tests focus on the randomness of the output fingerprints generated by a given individual, we also fixed the value of

the input state block for the whole compression function evaluation. This fixed value, denoted IV, is defined at the beginning of a GEPHashSearch execution and is kept constant until the end of the run. A brief overview of the considered tests is now provided (see [**22**] for more details):

- $\mathcal{T}_1$**: Frequency (Monobit) Analysis:** the objective is to determine whether the number of ones and zeros in the tested sequence is approximately the same as would be expected for a truly random sequence.
- $\mathcal{T}_2$**: Frequency Analysis within each block:** this test determines the proportion of ones within each b-bit blocks y_i: it should be approximately $\frac{b}{2}$, as would be expected under the assumption of randomness.
- $\mathcal{T}_3$**: Longest sequence of identical bits (or runs):** this test analyzes the total number of runs in the sequence s, where a run is an uninterrupted sequence of identical bits, to determine whether the number of runs of ones and zeros of various lengths is as expected for a random sequence.
- $\mathcal{T}_4$**: Longest sequence of ones within each block:** as $\mathcal{T}_3$ yet focusing on the longest run of ones within each b-bit blocks y_i.
- $\mathcal{T}_5$**: Binary Matrix Rank Test:** the focus of this test is the rank of disjoint sub-matrices of the entire sequence, to check linear dependence among the fixed length sub-strings of the original sequence.
- $\mathcal{T}_6$**: Discrete Fourier Transform (DFT) (Spectral) Test**: this test analyzes the peak heights in the DFT of the sequence in order to detect periodic features (*i.e.*, repetitive patterns that are near each other) that would indicate deviation from the assumption of randomness.
- $\mathcal{T}_7$**: Non-overlapping Template Matching Test:** the purpose of this test is to detect the production of too many occurrences of a given non-periodic (aperiodic) pattern.
- $\mathcal{T}_8$**: Overlapping Template Matching Test:** this test complements the test $\mathcal{T}_7$ to check the number of occurrences of prespecified target strings.
- $\mathcal{T}_9$**: Maurer's "Universal Statistical" Test:** it detects whether or not the sequence can be significantly compressed without loss of information. A significantly compressible sequence is considered to be non-random.
- $\mathcal{T}_{10}$**: Linear Complexity Test:** the focus of this test is the length of Linear Feedback Shift Register (LFSR) to determine whether or not the sequence is complex enough to be considered random (random sequences are characterized by longer LFSRs).
- $\mathcal{T}_{11}$**: Serial Test:** the purpose of this test is to determine whether the number of occurrences of the 2^m m-bit overlapping patterns is approximately the same as would be expected for a random sequence. Random sequences have uniformity; that is, every m-bit pattern has the same chance of appearing as every other m-bit pattern.

Table 3. Minimal size of the tested sequence given to each NIST test to fulfill both the GEPHashSearch context and the NIST recommendations.

Test	min $\|\mathbf{s}\|$ (bits)	**Test**	min $\|\mathbf{s}\|$ (bits)	**Test**	min $\|\mathbf{s}\|$ (bits)
$\mathcal{T}_1$	100	$\mathcal{T}_6$	1,000	$\mathcal{T}_{11}$	1,024
$\mathcal{T}_2$	100	$\mathcal{T}_7$	10^6	$\mathcal{T}_{12}$	1,024
$\mathcal{T}_3$	100	$\mathcal{T}_8$	10^6	$\mathcal{T}_{13}$	100
$\mathcal{T}_4$	6,272	$\mathcal{T}_9$	904,960	$\mathcal{T}_{14}$	10^6
$\mathcal{T}_5$	38,912	$\mathcal{T}_{10}$	10^6	$\mathcal{T}_{15}$	10^6

$\mathcal{T}_{12}$**: Approximate Entropy Test:** similarly to $\mathcal{T}_{11}$, this test compares the frequency of overlapping blocks of two consecutive/adjacent lengths against the expected result for a random sequence.

$\mathcal{T}_{13}$**: Cumulative Sums (Cusum) Test:** this test determines whether the cumulative sum of the partial sequences occurring in the tested sequence is too large or too small relative to the expected behaviour of that cumulative sum for random sequences.

$\mathcal{T}_{14}$**: Random Excursions Test:** the focus of this test is the number of cycles having exactly K visits in a cumulative sum random walk.

$\mathcal{T}_{15}$**: Random Excursions Variant Test:** this test detects deviations from the expected number of visits to various states in the random walk.

In algorithm 2, the minimum length l of the tested sequence should be selected carefully, in accordance with the NIST recommendations for each test. We adapt these recommendations with regards of the GEPHashSearch context to produce the results presented in Table 3. Based on this analysis, GEPHashSearch uses the value $l = \mathbf{10^6}$ **bits** for the computation of RC in algorithm 2.

Complexity W_1**.** This metric measures the complexity of the compression function seen as the accumulated number of cycles require to execute the operators that compose the evaluated individual i.e. $W_1 = \sum_{\texttt{op} \in \texttt{Ind} \bigcap \Delta_f} W_1(\texttt{op})$. This assumes that the cost (in terms of a number of cycles) of all the operators that compose the set Δ_f is known. The examples of such evaluations can be found in various speed benchmarks of hash functions. For instance, those made in Blake [**16**] estimate that $W_1(\gg_7) = 12$ cycles, $W_1(\texttt{addmod}_{2^{32}})= 24$ cycles etc. The notation W_1 is used as this criterion reflects the sequential work required to perform the execution of the compression function.

Efficiency E**.** This metric makes a raw estimation of the efficiency of the tested individual by translating automatically its ET into an C code which is compiled and executed on a reference platform to determine the average time required by the associated compression function to get the fingerprint of random message chunks.
In practice, GEPHashSearch benchmarks the time t required to generate consecutively fingerprints of the e first elements of the set Δ_x (the one used for the evaluation

of RC, each of n bits). Then, the average hash efficiency (expressed in kB/s) of the individual is returned as follows: $E = \frac{e \times n}{1000 \times t}$

4.2 Multi objective optimization in GEPHashSearch

If the first implementation of the GEPHashSearch framework targets the optimization of each fitness object $PC_k(t)$, RC, W_1 and E independently, the real goal is to effectively explore and measure the trade-off that might be selected among these four objectives. In practice, we plan to build the set of optimal solutions (largely known as the Pareto-optimal solutions) using the NSGA-II algorithm [**23**], which works expressly with the notion of dominance. Just as all the individuals in the Pareto front are dominant by definition, NSGA-II is an elitist algorithm that selects across the pareto dominant individuals. One point worth noting is that evaluating over multiple criteria takes a greater amount of time for each additional test, as does checking for dominance over other individuals. While classical Multi-objective EAs that use non-dominated sorting and sharing have been criticized for their non-elitism approach and their computational complexity – $\mathcal{O}(|Pop|^3)$ in our case (more precisely $\mathcal{O}(4 \times |Pop|^3)$ where $|Pop|$ is the population size as there are 4 objectives to optimize), NSGA-II is faster as this non-dominated sorting approach has a computational complexity of $\mathcal{O}(|Pop|^2)$ in the GEPHashSearch context which explains why it received our preference.

5 Experiments

As mentioned before, we present here a work in progress and we are now in the process of validating the design choices proposed in the article. The implementation of GEPHashSearch consists in two fundamental components: (1) `LibGEP`[3] (version 0.4.1), a C++ library partly developed by one of the authors which provides a convenient interface to the GEP heuristic; (2) `ParadisEO`[4] (version 1.3), a portable C++ middleware for the manipulation of EAs heuristics. While at the moment of writing this implementation this has not been finished, we can propose here the first experimental results that were obtained in a mono-objective context. They were obtained by running GEPHashSearch on the resources of the computing cluster of the University of Luxembourg (see `http://hpc.uni.lu`). The parameters of the evolutionary processes executions are as follows:

- 100 executions on a population of 100 individuals, 20 generations
- Fingerprint size $n = 128$ bits (*i.e.* $N = 4$)
- Message chunk size $b = 128$ bits (*i.e.* $B = 4$)
- Level of confidence $\alpha = 0.05$
- Probability of *linking function* mutation: 0.05
- Probability of individual (resp. gene) mutation: 0.3 (resp. 0.1)

[3]See `http://libgep.gforge.uni.lu`
[4]See `http://paradiseo.gforge.inria.fr/`

- Probability of two-point crossover: 0.5

Figure 7 depicts the mono-objective optimization of the propagation criteria $PC_{1000}(4)$. More precisely, it illustrates the evolution of the average fitness of the best individual (over 50 executions) – Mean Best Fitness (MBF). We can see that GEPHashSearch converges quickly to optimal solutions (for which the χ^2 statistic observed at the end (42.02559) is below the expected value (46) for the selected level of confidence. This is obviously encouraging and we are now finalizing the implementation of the other criteria, namely RC, W_1 and E to permit concrete runs over NSGA-II withing GEPHashSearch.

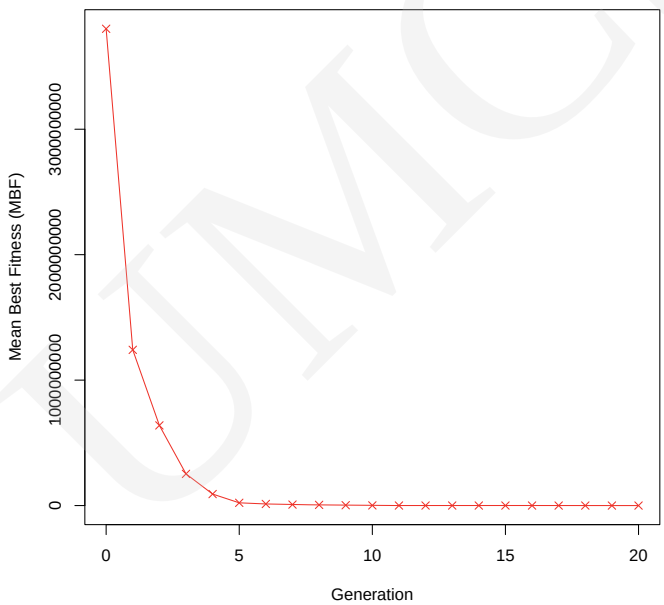

Fig. 7. Mono-objective optimization of the Propagation Criteria $PC_k(t)$ (with $k = 1000$ and $t = 4$) in GEPHashSearch.

6 Conclusion

Cryptographic hash functions are fundamental primitives in modern cryptography and have many security applications. In this paper, the GEPHashSearch framework was proposed. The objective is to build compression functions with reasonably "good" properties by means of the Gene Expression Programming (GEP) heuristic – an efficient alternative to the classical Genetic Programming (GP). As it is still a work in progress, this article focuses on the design aspects of this framework rather

than on complete implementation details and validation results. In particular, the complete description of the GEP individuals encoding together with four fitness objectives has been detailed. While the way GEP individuals are encoded reflects the organization of the most known hash functions (MD5, SHA-1 but also the five finalists of the SHA-3 contest organized by the NIST were studied), the defined objectives try to catch the expected properties of the underlying compression functions: a descent propagation of small modifications in the input message (captured by the propagation criteria $PC_k(t)$), reasonably good randomness (attested by the RC fitness value which reflects the successful passing of the 15 tests provided by the NIST Statistical Test Suite) and good performances (as measured by the complexity W_1 and the efficiency of the hashing E). The first experimental results are quite promising. Without doubts, the multi-objective optimization of this problem over these four criteria will lead to fruitful contributions to the cryptographic community.

References

[1] Koza J. R., Genetic Programming, MIT Press, (1992)
[2] Ferreira C., Gene expression programming: A new adaptive algorithm for solving problems, Complex Systems 13(2) (2001): 87–129.
[3] Menezes A. J., Vanstone S. A., Van Oorschot P. C., Handbook of Applied Cryptography, Computer Sciences Applied Mathematics Engineering, CRC Press Inc., 1st edition (1996).
[4] Rivest R., The MD5 Message-Digest Algorithm, Request for Comments (RFC) 1321, Network Working Group (1992).
[5] Eastlake D., Jones P., US Secure Hash Algorithm 1 (SHA1), Request for Comments (RFC) 3174, Network Working Group (2001).
[6] Bellare M., Micciancio D., A new paradigm for collision-free hashing: Incrementality at reduced cost, Advances in Cryptology—EUROCRYPT 97 volume 1233 of Lecture Notes in Computer Science, Springer-Verlag (1997): 163–192.
[7] Darwin C., The Origin of Species, John Murray (1859).
[8] Hernandez-Castro J. C., Viñuela P. I., Evolutionary computation in computer security and cryptography, New Gen. Comput. 23(3) (2005): 193–199.
[9] Clark A. J., Optimisation heuristics for cryptology, PhD thesis, Queensland University of Technology (1998).
[10] Daemen J., Govaerts R., Vandewalle J., A framework for the design of one-way hash functions including cryptanalysis of damgård's one-way function based on a cellular automaton, Advances in Cryptology ASIACRYPT '91, volume 739 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg (1993): 82–96.
[11] Damgå rd I., A design principle for hash functions, Advances in Cryptology CRYPTO'89 Proceedings, volume 435 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg (1990): 416–427.
[12] Damiani E., Tettamanzi A. G. B., Liberali V., Automatic synthesis of hashing function circuits using evolutionary techniques, Institute of electrical and electronics engineers, Los Alamitos (1998): 42–45.
[13] Safdari M., Evolving universal hash functions using genetic algorithms, In Proc. of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers, GECCO '09, New York, NY, USA, ACM (2009): 2729–2732.

[14] Estevez-Tapiador J. M., Hernandez-Castro J. C., Peris-Lopez P, Ribagorda A., Automated Design of Cryptographic Hash Schemes by Evolving Highly-Nonlinear Functions, Journal of Information Science and Engineering 24(5) (2008): 1485–1504.

[15] Ferreira C., Gene Expression Programming: Mathematical Modeling by an Artificial Intelligence, Springer-Verlag, 2nd edition (2006).

[16] Aumasson J.-P., Henzen L., Meier W., Phan R. C.-W., SHA-3 proposal BLAKE, Technical report, NIST (2010); http://www.131002.net/blake/blake.pdf

[17] Knudsen L. R., Gauravaram P., Matusiewicz K., Mendel F., Rechberger C., Schläffer M., Thomsen Søren S., `Grøstl` a SHA-3 candidate, Technical report, NIST (2011); http://www.groestl.info/Groestl.pdf

[18] Wu H., The Hash Function JH, Technical report, NIST (2011); http://www3.ntu.edu.sg/home/wuhj/research/jh/jh_round3.pdf

[19] Daemen J., Bertoni G., Peeters M., Van Assche G., Van Keer R., Keccak implementation overview, Technical report, NIST (2012); http://keccak.noekeon.org/Keccak-implementation-3.2.pdf

[20] Schneier B., Ferguson N., Lucks S., Whiting D., Bellare M., Kohno T., Walker J., Callas J., The Skein Hash Function Family, Technical report, NIST (2010); http://www.skein-hash.info/sites/default/files/skein.pdf

[21] Preneel B., Van Leekwijck W., Van Linden L., Govaerts R., Vandewalle J., Propagation characteristics of boolean functions (1990).

[22] Rukhin A. and al., NIST SP 800-22 – A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications, Technical report, NIST (2010); http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html

[23] Deb K., Agrawal S., Pratap A., Meyarivan T., A fast and elitist multiobjective genetic algorithm: NSGA-II, IEEE Trans. Evolutionary Computation 6(2) (2002): 182–197.