On the Evolution of Bent (n, m) Functions

Abstract—**Boolean functions as well as their generalizations, vectorial Boolean functions are extremely active areas of research. Their applications can be found in domains such as error correcting codes, communication, and cryptography. Accordingly, various methods how to obtain Boolean functions are explored where one group belongs to heuristic techniques and more precisely, evolutionary algorithms. In this paper we explore how to evolve (vectorial) Boolean functions with specific properties by utilizing several different algorithms and encodings. As far as we are aware, we are the first to explore the topic of evolution of vectorial Boolean functions where the output dimension is strictly smaller than the input dimension. Our results show that evolutionary algorithms represent a valuable option to produce vectorial Boolean functions where good results are obtained for various sizes. On the other hand, as the number of outputs grow, we can observe that evolutionary algorithms are still able to obtain high quality results but with much more difficulty.**

I. Introduction

The role of Boolean functions is diverse and spans across several research domains such as communication, coding theory, and **cryptography**. Within cryptography area, Boolean functions again have a number of applications that result in a need for construction techniques able to produce Boolean functions of various sizes and properties.

For instance, in cryptography, Boolean functions and their generalizations vectorial Boolean functions (also called S-boxes) have a prominent usage in symmetric key cryptography [1]. Symmetric key cryptography can be divided into stream ciphers and block ciphers where again (vectorial) Boolean functions have different usages. However, their goal is always the same: to improve the resiliency of cryptographic algorithms (commonly known as ciphers) against various cryptanalyses. For instance, when discussing block ciphers we want to have resistance against differential [2] and linear [3] cryptanalysis. On the other hand, with stream ciphers some attacks we want to have resilience against are fast correlation attack [4], Berlekamp-Massey attack [5], and algebraic attack [6].

The resilience against aforementioned attacks stems from smartly chosen (vectorial) Boolean functions where in block ciphers we mostly use S-boxes where the input size is equal to the output size, i.e., (n, n) functions like one that can be found in the AES cipher [7] or where the output size is somewhat smaller than the input size like in the DES cipher [8]. On the other hand, in stream ciphers one usually uses Boolean functions or S-boxes where the output dimension is strictly smaller than the input dimension [9].

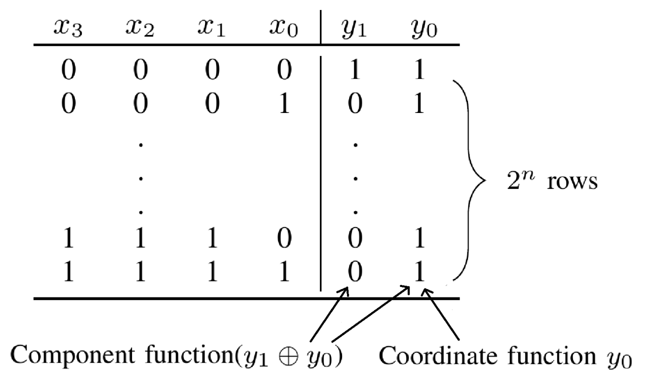
Naturally, since there is a plethora of different ciphers, the requirements on those Boolean functions or S-boxes differ (e.g., different sizes or cryptographic properties they need to possess). Accordingly, to produce such diversity of functions, researchers developed over the course of years a number of construction techniques. Such techniques can be divided into algebraic constructions, random search, heuristics, and combinations of those techniques [10]. Here we are interested in investigating how one type of heuristics, namely evolutionary algorithms (EA) can be used in the evolution of (vectorial) Boolean functions with good cryptographic properties. Naturally, as already said, this area is very active research domain and this also holds when applying heuristic techniques. Results obtained up to now (see Section III) suggest that evolutionary algorithms are very good choice for the evolution of Boolean functions where it is possible to obtain optimal results with respect to a number of properties and Boolean function sizes. On the other hand, when considering S-boxes, we observe that most of the works consider functions where the input and output dimensions are of the same size (which is also the combination mostly used in practical applications). However, except for the smallest size of practical importance (4×4), the results suggest that EAs are not able to compete with algebraic constructions nor to achieve optimal results.

In this paper, we investigate how to evolve vectorial Boolean functions ((n, m) functions) where the output size is strictly smaller than the input size. Moreover, we are interested in the evolution of bent (n, m) functions which necessitates that the input dimension n is always even and that the output dimension m is smaller or equal to $n/2$. As far as we are aware, we are the first to consider this problem that has practical applications in authentications schemes [11] and secret sharing [12], [13]. However, we emphasize that this problem is also interesting from the combinatorial optimization perspective. In order to provide extensive results, we experiment with three fitness functions, three encodings, and a number of function sizes.

The rest of the paper is organized as follows. In Section II we introduce the notation we follow and give relevant theory about vectorial Boolean functions. In Section III we enumerate related work. Section IV presents our experimental setting as well as the obtained results. In the same section we provide a discussion on the results as well as several possible future research directions. Finally, Section V gives a brief conclusion.

II. Preliminaries

Let n, m be positive integers, i.e., $n, m \in \mathbb{N}^+$. We denote by $\mathbb{F}_2^n$ the n-dimensional vector over $\mathbb{F}_2$ and by $\mathbb{F}_{2^n}$ the finite field with 2^n elements. The set of all n-tuples of elements in the field $\mathbb{F}_2$ is denoted by $\mathbb{F}_2^n$, where $\mathbb{F}_2$ is the Galois field with two elements. Further, for any set S, we denote $S \backslash \{0\}$ by S^*. The usual inner product of a and b equals



x_3	x_2	x_1	x_0	y_1	y_0
0	0	0	0	1	1
0	0	0	1	0	1
		.		.	
		.		.	
		.		.	
1	1	1	0	0	1
1	1	1	1	0	1

2^n rows
Component function($y_1 \oplus y_0$)
Coordinate function y_0

Fig. 1: An example of $(4, 2)$ function.

$a \cdot b = \bigoplus_{i=1}^{n} a_i b_i$ in F_2^n. The Hamming weight (w_H) of a vector a, where $a \in \mathbb{F}_2^n$, is the number of non-zero positions in the vector. An (n, m)-function is any mapping F from $\mathbb{F}_2^n$ to $\mathbb{F}_2^m$. An (n, m)-function F can be defined as a vector $F = (f_1, \cdots, f_m)$, where the Boolean functions $f_i : \mathbb{F}_2^n \rightarrow \mathbb{F}_2$ for $i \in \{1, \cdots, m\}$ are called the coordinate functions of F. The component functions of an (n, m)-function F are all the linear combinations of the coordinate functions with non all-zero coefficients. In the rest of the paper we use a capital letter F when discussing vectorial Boolean functions and small letter f when discussing Boolean functions. We give a small example of a $(4, 2)$ function F in Figure 1 where one can see the difference between the coordinate and component function.

A Boolean function f on $\mathbb{F}_2^n$ can be uniquely represented by a truth table (TT), i.e., a vector $(f(0), ..., f(1))$ that contains the function values of f, ordered lexicographically [9].

The second unique representation of a Boolean function is the Walsh-Hadamard transform W_f that measures the correlation between $f(x)$ and all linear functions $a \cdot x$ [9], [14]. The Walsh-Hadamard transform of a Boolean function f equals:

$$W_f(a) = \sum_{x \in \mathbb{F}_2^n} (-1)^{f(x) \oplus a \cdot x}. \tag{1}$$

The Walsh-Hadamard transform of an (n, m)-function F is a set of values [15]:

$$W_F(a, v) = \sum_{x \in \mathbb{F}_2^m} (-1)^{v \cdot F(x) \oplus a \cdot x}, \ a, v \in \mathbb{F}_2^m. \tag{2}$$

A Boolean function with n inputs is balanced if its Hamming weight equals 2^{n-1}. An (n, m)-function F is balanced if it takes every value of $\mathbb{F}_2^m$ the same 2^{n-m} number of times.

A Boolean function f should lie at a large Hamming distance (HD) from all affine functions and the nonlinearity N_f of a Boolean function is the minimum Hamming distance between the function f and affine functions [9]. The nonlinearity N_f of a Boolean function f expressed in terms of the Walsh-Hadamard coefficients equals [9]:

$$N_f = 2^{n-1} - \frac{1}{2} \max_{a \in \mathbb{F}_2^n} |W_f(a)|. \tag{3}$$

The nonlinearity N_F of an (n, m)-function F equals the minimum nonlinearity of all its component functions $v \cdot F$, where $v \in \mathbb{F}_2^{m*}$ [15]:

$$N_F = 2^{n-1} - \frac{1}{2} \max_{\substack{a \in \mathbb{F}_2^n \\ v \in \mathbb{F}_2^{m*}}} |W_F(a, v)|. \tag{4}$$

The Parseval's relation equals:

$$\sum_{a \in \mathbb{F}_2^n} W_f(a)^2 = 2^{2n}, \tag{5}$$

and it implies that the mean of $W_f(a)^2$ equals 2^n, and $\max_{a \in \mathbb{F}_2^n} |W_f(a)|$ is then at least equal to the square root of this mean.

From Eq. (5), it follows that the maximal value of the Walsh-Hadamard spectrum equals at least $2^{\frac{n}{2}}$, which occurs with equality in the case of bent Boolean functions. From this equation we see that the nonlinearity of any Boolean function is less or equal to:

$$N_f \leq 2^{n-1} - 2^{\frac{n}{2}-1}. \tag{6}$$

This bound is called the covering radius bound and is strict for bent Boolean functions. Furthermore, since this bound is valid for any Boolean function it is even more valid for vectorial Boolean functions. Therefore, in order for an (n, m) function to be bent, all of the component functions $v \cdot F, v \neq 0$ of a function F must be bent. Since bent n-dimensional Boolean functions can exist only when n is even, then bent (n, m) functions can exist only when n is even. However, this condition has been shown not to be sufficient. K. Nyberg showed that bent (n, m) functions can exist only when $m \leq \frac{n}{2}$ [16]. Bent (n, m) functions are also called perfect nonlinear functions [15]. For further information on vectorial Boolean functions and their applications in cryptography, we refer interested readers to [9], [15].

III. Related Work

As stated, there are many successful applications of evolutionary algorithms when constructing vectorial Boolean functions. Next, we give a list of relevant works, first for the Boolean functions and then for the vectorial Boolean functions.

McLaughlin and Clark use simulated annealing to generate Boolean functions that have optimal values of a number of properties: algebraic immunity, fast algebraic resistance, and algebraic degree [17]. In their work, they experiment with Boolean functions with sizes of up to 16 inputs.

Picek, Jakobovic, and Golub use genetic algorithms and genetic programming to find Boolean functions that possess several optimal properties where one of the scenarios is the evolution of bent Boolean functions [18].

Hrbacek and Dvorak use Cartesian genetic programming to evolve bent Boolean functions of sizes up to 16 inputs [19]. The authors do not limit the number of generations and therefore they succeed in finding bent function in each run for sizes between 6 and 16 variables.

Picek et al. investigate a number of different evolutionary algorithms and fitness functions for Boolean functions of 8 inputs [20]. They show that genetic programming and Cartesian genetic programming outperform genetic algorithms and evolution strategies in a number of relevant test scenarios where one of them is the evolution of bent Boolean functions.

Clark et al. experiment with simulated annealing in order to design Boolean functions by using spectral inversion [21]. Similarly, Mariot and Leporati use genetic algorithm where the genotype consists of the Walsh-Hadamard values in order to evolve semibent (plateaued, or 3-valued) Boolean functions [22].

Picek and Jakobovic use genetic programming in order to evolve algebraic constructions that are then used to construct bent Boolean functions [23]. With this approach, the authors are able to find bent Boolean functions for sizes up to 26 inputs.

Picek, Sisejkovic, and Jakobovic explore the efficiency of several immunological algorithms when constructing bent or balanced, highly nonlinear Boolean functions [24]. In this work, the authors use among others the floating-point encoding of solutions.

Next, we list several works considering the evolution of vectorial Boolean functions. Clark et al. use the principles from the evolutionary design of Boolean functions to evolve S-boxes with desired cryptographic properties. They use simulated annealing (SA) and hill climbing algorithm to evolve bijective S-boxes of sizes up to 8×8 with high nonlinearity values [25].

Millan et al. work with genetic algorithms to evolve S-boxes with high nonlinearity and low autocorrelation value. Moreover, the authors discuss the selection of the appropriate genetic algorithm parameters [26].

P. Tesar uses a combination of a special genetic algorithm with a total tree searching to generate 8×8 S-boxes with nonlinearity equal up to 104 [27].

Picek et al. explore how to generate S-boxes of size 8×8 with a better resistance against side-channel attacks as measured with the transparency order and modified transparency order properties [28], [29].

Finally, Picek, Rotim, and Cupic develop a new cost function able to reach high nonlinearity values for a number of different S-box sizes [30]. With this fitness function, they eliminated the need for several parameters that usually required extensive tuning phase.

IV. Experiments and Results

Before going into discussion on experimental setting and presenting results, we briefly discuss the difficulty of the problem we consider. A Boolean function f of n variables can be represented with a string of 2^n values and the search space size is equal to 2^{2^n}, as given in Table I for several sizes of interest. Note that to calculate nonlinearity and the Walsh-Hadamard spectrum, we always need to transform our solutions into the truth table representation. Next, in Table II we give optimal values for the nonlinearity property and each size of Boolean functions we consider in this paper. Note that the component-wise nonlinearity for vectorial Boolean function needs to be the same as for a single Boolean function.

TABLE I: The search space size for the investigated sizes of functions.

m \ n	4	6	8	10	12
1	2^{16}	2^{64}	2^{256}	$2^{1\,024}$	$2^{4\,096}$
2	2^{32}	2^{128}	2^{512}	$2^{2\,048}$	$2^{8\,192}$
3	-	2^{192}	2^{756}	$2^{3\,072}$	$2^{12\,288}$
4	-	-	2	$2^{4\,096}$	$2^{16\,384}$
5	-	-	-	$2^{5\,120}$	$2^{20\,480}$
6	-	-	-	-	$2^{24\,576}$

TABLE II: Nonlinearity of a bent function of n variables.

n	4	6	8	10	12
Nonlinearity	6	28	120	496	2 106

A. Algorithms and Representations

We experiment with three different representations for encoding of a (vectorial) Boolean function: bitstring, floating-point, and tree representation. The simplest genotype to use is the bitstring since in that case there is no need for mapping between it and the truth table representation of a Boolean function. However, as given in Table I we can observe that the genotype size increases exponentially with the number of variables of a Boolean function. Moreover, this number needs to be multiplied by the number of output variables when $m > 1$. To state it differently, quite soon one can expect problems with the size of solutions if using the truth table representation.

In the second encoding we use the floating-point genotype, which is defined as a vector of continuous variables. Therefore, once a solution is obtained, the first step is to transform it into the truth table representation of a Boolean function. Here, we use each continuous variable to decode a subset of bits from the truth table. In this way, we can compress the size of a solution, i.e., the number of genes we need to represent a solution. The transformation between real values and the bitstring is done as follows. First, we enforce that all real values are in the range $[0, 1]$. Then, the number of bits that are represented with a single floating-point value, $decode\_by$, can vary:

$$decode\_by = \frac{2^n}{dimension}, \tag{7}$$

where the parameter $dimension$ denotes the floating-point vector size (number of real values). Note that this parameter can be modified as long as the size of the truth table is divisible with it, so that each real value represents the same number of bits.

In the process of transformation of real values we first convert the floating-point values into integers. Each floating-point value falls into a specific *interval* between 0 and 1. Since each real value must represent $decode\_by$ bits, the size of that interval is given as:

$$interval = \frac{1}{2^{decode\_by}}. \quad (8)$$

For instance, if each real value encodes 2 bits from the truth table, the interval size is 0.25. To obtain a distinct integer for a given floating-point value, every element d_i of the floating-point vector is divided by the calculated interval size, generating a sequence of integers:

$$int\_value_i = \left\lfloor \frac{d_i}{interval} \right\rfloor. \quad (9)$$

In the example, the corresponding integer value is obtained by dividing the real value with $2^{-2} = 0.25$ and truncating to nearest smaller integer. The second step is to decode the integer values into a binary sequence to be used in evaluation. Here, we opted to use the Gray encoding, due to the proportional solution distance in both floating-point and binary search space.

For the algorithm we use a simple GA with the tournament selection where its size equals 3 [31] and the population size is 100. The mutation is selected uniformly at random between a simple mutation, where a single bit is inverted, and a mixed mutation, which randomly shuffles the bits in a randomly selected subset. The crossover operators are one-point and uniform crossover, performed at random for each new offspring.

As the third encoding, we use tree-based GP in which a Boolean function is represented by a tree of nodes [32]. The function set for GP in all the experiments is OR, NOT, XOR, AND, XNOR, AND with one input inverted and IF, which takes three arguments and returns the second one if the first evaluates to true, and the third one otherwise. The terminals correspond to n Boolean variables. Boolean functions can be represented only in XOR and AND operators, but it is quite easy to transform the function from one notation to the other. Note that when $m > 1$ we use m independent trees to represent a solution. GP also uses tournament selection with the tournament size equal to 3.

The crossover is performed with five different tree-based crossover operators selected at random: a simple tree crossover with 90% bias for functional nodes, uniform crossover, size fair, one-point, and context preserving crossover [33]. We use a single mutation type, a subtree mutation, and use maximum tree depth of 5. The population size for the GP equals 200.

B. Common Parameters

In all the experiments the number of independent trials N for each configuration is 30 and the stopping criterion for all algorithms equals 500 000 evaluations. For each of the algorithms and representations we use the individual mutation probability of 0.5. It is important to note that we use the mutation probability to select whether an individual would be mutated or not, and the mutation operator is executed only once on a given individual; e.g. if the mutation probability is 0.5, then on average 5 out of every 10 new individuals will be mutated (see Algorithm 1), and one mutation will be performed on that individual.

Algorithm 1 Steady-state tournament selection

```
randomly select $k$ individuals;
remove the worst of $k$ individuals;
$child$ = crossover (best two of the tournament);
perform mutation on $child$, with given individual mutation probability;
insert $child$ into population;
```

C. Fitness Functions

In the next section, we present fitness functions we consider in our experiments. Note that the first two functions are well established and used in related work while the third fitness function is, as far as we know, explored for the first time in the evolution of vectorial Boolean functions.

1) Fitness Function 1: In the simplest version of the fitness function we aim to maximize the nonlinearity value:

$$fitness_1 = N_F. \quad (10)$$

2) Fitness Function 2: The second version of fitness function improves on Eq. (10) and adds the second term where we aim to minimize the number of occurrences of values different from $2^{\frac{n}{2}}$ as given by the Walsh-Hadamard spectrum. Note that this is a version of the fitness that usually gives better results for vectorial Boolean functions where the input and output dimension are the same (however, there it is mostly used with the permutation encoding). Here we are again interested in the maximization of the following expression:

$$fitness_2 = N_F + \frac{1}{T}, \quad (11)$$

where T equals the number of occurrences of the value different from $2^{\frac{n}{2}}$ in the Walsh-Hadamard spectrum. Since we still consider nonlinearity as the primary parameter, we scale the second term in the fitness function in the range $[\frac{1}{2^n}, 1]$. Note that if $T = 0$ then the fraction is set to 1.

3) Fitness Function 3: Finally, in the third fitness function we try to explore the relation previously not considered when evolving bent (vectorial) Boolean functions. A Boolean function is bent if all its derivatives are balanced [9]. Here, a derivative D of a Boolean function f in the direction of b equals:

$$D_b f(x) = f(x) \oplus f(x \oplus b). \quad (12)$$

For a vectorial Boolean function to be bent, all its linear components need to be balanced and therefore we aim to maximize the following expression:

$$fitness_3 = \sum_{i=1}^{2^n} \sum_{b=1}^{2^n} D_b f(x_i), \quad (13)$$

where x_i represents the value of a function f for input x at position i.

TABLE III: Results for the tree representation, fitness 1.

m \ n	4	6	8	10	12
1	6/6	28/28	120/120	496/496	2 016/2 016
2	6/6	28/28	120/120	496/496	2 016/1 984
3	-	28/28	120/106.5	480/463	1 984/1 920
4	-	-	112/104	480/448	1 944/1 878
5	-	-	-	448/447	1 920/1 792
6	-	-	-	-	1 888/1 765

TABLE IV: Results for bitstring representation, fitness 1.

m \ n	4	6	8	10	12
1	6/6	28/26	114/113.5	478/476	1 964/1 961
2	6/6	26/25	112/110	472/470	1 956/1 952
3	-	24/24	110/108	468/466	1 948/1 946
4	-	-	108/106	464/464	1 944/1 940
5	-	-	-	462/460	1 936/1 934
6	-	-	-	-	1 930/1 929

TABLE V: Results for the floating-point representation, fitness 1.

m \ n	4	6	8	10	12
1	6/6	26/26	112/112	476/475	1 950/1 936
2	6/6	26/24	110/109	472/470	1 934/1 920.5
3	-	24/24	108/106	468/466	1 924/1 912
4	-	-	106/104	464/464	1 918/1 900.5
5	-	-	-	462/460	1 914/1 895
6	-	-	-	-	1 912/1 893

TABLE VI: Results for the tree representation, fitness 2.

m \ n	4	6	8	10	12
1	6/6	28/28	120/120	496/496	2 016/2 016
2	6/4	28/28	120/120	496/480	2 016/1 984
3	-	28/24	120/106.5	480/460	2 016/1 920
4	-	-	112/104	480/448	1 920/1 856
5	-	-	-	448/440	1 920/1 792
6	-	-	-	-	1 856/1 768

D. Results

We give the results in the *max value/median value* notation since the first number has greater relevance from the practical perspective (we are interested in a single function with as good as possible nonlinearity value) while the second number has more relevance from the optimization perspective, to assess what is the average behavior of the algorithms we tested. Note that we opted to use median rather than the average value so not to assume normal distribution of data. In Table III we give the results for tree encoding and fitness function 1. We see that when $m = 1$ (single Boolean function case) GP is able to find optimal results for all tested input size in 100% of cases. Moreover, GP is able to reach optimal values in all the runs for when considering input dimension of 6 variables. On the other hand, starting with $n = 8$, GP is not able to reach optimal value when the output dimension is equal or greater than 3.

Next, in Table IV we give the results for the bitstring representation and fitness function 1. Note that here, except for the smallest size (i.e., when $n = 4$), we cannot reach the optimal values. Therefore, our results corresponds to the ones from related work for cases when $m = 1$ or $n = m$. On the other hand, for dimensions $(10, 5), (12, 5), (12, 6)$ bitstring representation outperforms the tree representation.

Finally, in Table V we give results for the fitness function 1 and floating-point representation. As we can see, the results are somewhat worse than for the bitstring encoding. Such results are also in accordance with the results from [24] where the authors note that floating-point representation is in the most of the cases the worst one.

In the second scenario we improve our fitness function to give more information and consequently to reach better values. We emphasize that for the fitness function 2, we give in tables the nonlinearity values and not the fitness values (with the decimal part from the second term in Eq. (11)). Surprisingly, we can observe that in a number of scenarios the median value is even lower than for the fitness function 1 (cf. Table III). However, we note that with the fitness function 2 we are able to reach optimal value for the $(12, 3)$ case which we could not reach with any representation and fitness function 1.

In Table VII we give results for the fitness function 2 with the bitstring encoding. There, for the dimension $(6, 1)$ the results are slightly worse when compared to the fitness function 1, but for $(8, 2), (10, 4), (12, 1), (12, 5), (12, 6)$ dimensions, the fitness value improves although it does not reach optimal values. What is interesting to notice is that the bitstring representation outperforms the tree encoding for sizes $(10, 5), (12, 4), (12, 5), (12, 6)$ which is similar situation as with the fitness function 1. On the other hand, smaller dimensions reach higher fitness values when the tree encoding is used.

When working with the floating-point encoding, we can observe that the improved fitness function does not help and actually for a number of scenarios we reach even worse values when compared to the fitness function 1. Moreover, the results with the floating-point encoding and fitness function 2 are the worst ones when compared to the other encodings.

In Figures 2a until 2f we give statistics in the boxplot form for fitness functions 1 and 2 and for function input sizes $n =$

TABLE VII: Results for the bitstring representation, fitness 2.

m \ n	4	6	8	10	12
1	6/6	26/26	114/114	478/476	1 968/1 962
2	6/6	26/25.5	114/110	472/470	1 956/1 952.5
3	-	24/24	110/108	468/466	1 948/1 946
4	-	-	108/106	466/464	1 944/1 940
5	-	-	-	462/460	1 938/1 934
6	-	-	-	-	1 932/1 928.5

TABLE VIII: Results for the floating-point representation, fitness 2.

m \ n	4	6	8	10	12
1	6/6	26/26	112/111	477/476	1 952/1 936
2	6/6	26/24	110/108	472/470	1 928/1 913
3	-	24/24	108/106	468/466	1 968/1 910
4	-	-	106/104	464/464	1 922/1 906
5	-	-	-	462/460	1 918/1 898.5
6	-	-	-	-	1 906/1 898

TABLE IX: Theoretical maximal values for fitness 3.

m \ n	4	6	8
1	15	63	255
2	45	189	765
3	-	441	1 785
4	-	-	3 825

$8, 10, 12$. To denote the scenarios we use the notation: input size – output size encoding (T for tree, B for bitstring, and F for the floating-point encoding). We also use a color coding where the boxplot in red color represents the tree encoding, in green color the bitstring encoding, and finally in blue color the floating-point encoding. From the graphs it is easy to see that when the output dimension is small (e.g., $m = 1$), the tree encoding is by far superior when compared to the other two encodings. Furthermore, for input sizes 10 and 12, tree encoding performs the best on average with a clear advantage over other encodings while for the input dimension 12 the results are much more similar over all encodings (especially when $m > 1$).

Next, we present results when experimenting with the fitness function where the objective is to obtain all balanced derivatives of linear combinations of a vectorial Boolean function. First, in Table IX we give the theoretical best results. Note that here the higher the value, the more derivatives are balanced. In Table X we give the results for tree encoding and input dimension in the range $[4, 8]$ and the output dimension in the range $[1, 4]$. We do not display the results for larger sizes nor for the other encodings since the results obtained are inferior when compared to the first two fitness functions. Nevertheless, what can be seen here is that the results for evolving bent Boolean functions ($m = 1$) are very promising and deserves further investigation.

TABLE X: Results for the tree representation, fitness 3.

m \ n	4	6	8
1	15/15	63/63	255/255
2	45/45	189/189	765/765
3	-	438/435	1 779/1 776
4	-	-	3 813/3 795

E. Future Work

In the future work, we plan to explore two research avenues. In the first one, we plan to explore Cartesian genetic programming (CGP) since the tree-based genetic programming gave very good results in our experiments. With CGP we would avoid the need to have m independent trees and would rather have a graph with m outputs. The second research direction involves exploring larger S-box sizes and especially bitstring representation that we observed to perform the best for size $(12, 6)$. Finally, since we observe a good performance when using fitness function 3 (Eq. (13)) we plan to explore its behavior on larger Boolean functions.

V. Conclusion

In this paper we investigate the evolution of bent vectorial Boolean functions. Differing from the case when considering single Boolean functions (i.e., where the number of outputs equals 1) or S-boxes as used in block ciphers (where the input and output dimensions are usually the same), here we investigate S-boxes where the output is strictly smaller than the input. Such S-boxes have practical applications in authentication codes or secret sharing schemes, but are also interesting as combinatorial optimization problems and could be used as benchmarks.

Our results suggest that evolutionary algorithms can be used to evolve a number of different sizes of vectorial Boolean functions where as the best performing algorithm we consider genetic programming. Naturally, that successfulness stems from the representation rather than using some specific selection strategy. Finally, the results show that when the number of outputs is strictly smaller than the number of inputs, this problem is more reminiscent of a Boolean function case than the S-box case. Moreover, differing from results usually obtained for S-boxes with $m = n$, here evolutionary algorithms can be regarded as a viable option when a number of different functions are needed.

References

[1] C. Paar and J. Pelzl, *Understanding Cryptography - A Textbook for Students and Practitioners*. Springer, 2010.

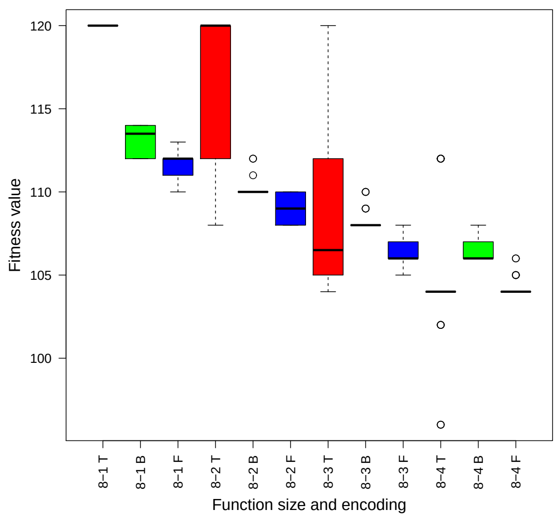
[2] E. Biham and A. Shamir, "Differential Cryptanalysis of DES-like Cryptosystems," in *Proceedings of the 10th Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO '90. London, UK, UK: Springer-Verlag, 1991, pp. 2–21.

[3] M. Matsui and A. Yamagishi, "A new method for known plaintext attack of FEAL cipher," in *Proceedings of the 11th annual international conference on Theory and application of cryptographic techniques*, ser. EUROCRYPT'92. Berlin, Heidelberg: Springer-Verlag, 1993, pp. 81–91.

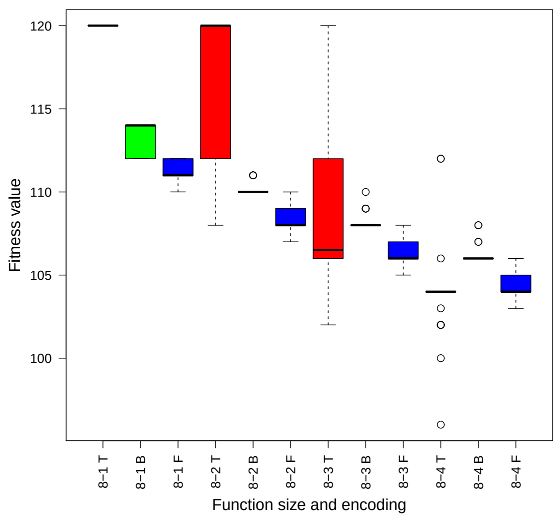
[4] W. Meier and O. Staffelbach, "Fast Correlation Attacks on Stream Ciphers," in *Advances in Cryptology - EUROCRYPT '88*, ser. Lecture Notes in Computer Science, D. Barstow, W. Brauer, P. Brinch Hansen, D. Gries, D. Luckham, C. Moler, A. Pnueli, G. Seegmüller, J. Stoer, N. Wirth, and C. Günther, Eds. Springer Berlin Heidelberg, 1988, vol. 330, pp. 301–314.

[5] J. Massey, "Shift-register synthesis and BCH decoding," *Information Theory, IEEE Transactions on*, vol. 15, no. 1, pp. 122–127, Jan 1969.

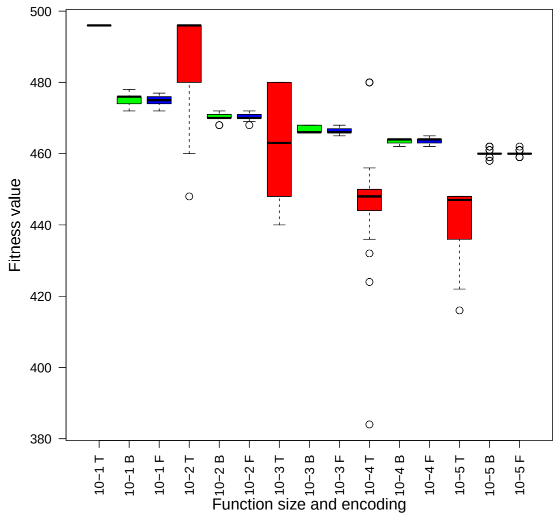
[6] N. Courtois and W. Meier, "Algebraic Attacks on Stream Ciphers with Linear Feedback," in *Advances in Cryptology - EUROCRYPT 2003*, ser. Lecture Notes in Computer Science, E. Biham, Ed. Springer Berlin Heidelberg, 2003, vol. 2656, pp. 345–359.



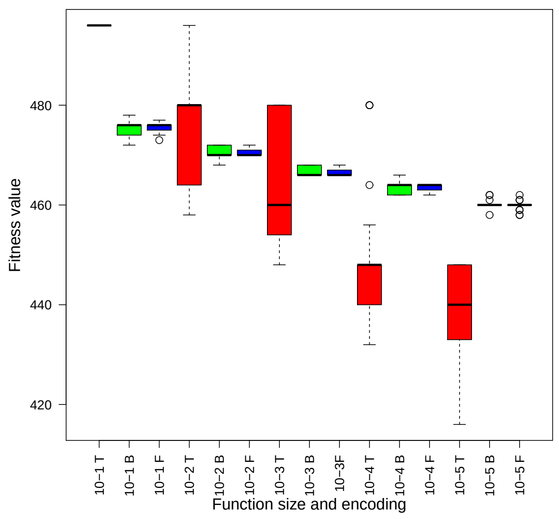
(a) Fitness function 1, $n = 8$



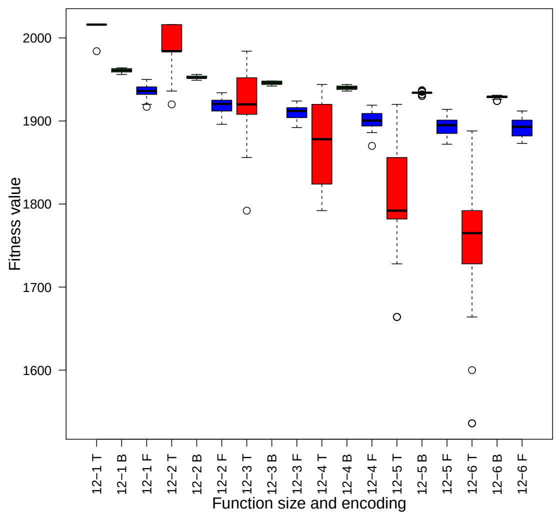
(b) Fitness function 2, $n = 8$



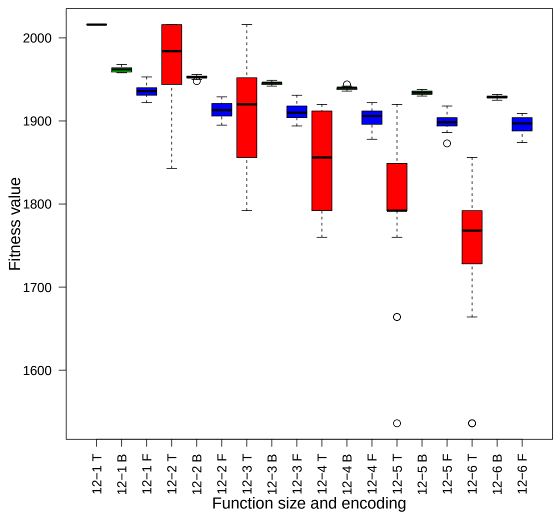
(c) Fitness function 1, $n = 10$



(d) Fitness function 2, $n = 10$



(e) Fitness function 1, $n = 12$



(f) Fitness function 2, $n = 12$

[7] J. Daemen and V. Rijmen, "Probability distributions of correlation and differentials in block ciphers," *J. Mathematical Cryptology*, vol. 1, no. 3, pp. 221–242, 2007.

[8] "FIPS 46-3, Data Encryption Standard (DES)," National Institute for Standards and Technology (NIST), Gaithersburg, MD, USA, 1999.

[9] C. Carlet, "Boolean Functions for Cryptography and Error Correcting Codes," in *Boolean Models and Methods in Mathematics, Computer Science, and Engineering*, 1st ed., Y. Crama and P. L. Hammer, Eds. New York, NY, USA: Cambridge University Press, 2010, pp. 257–397.

[10] S. Picek, D. Jakobovic, J. F. Miller, E. Marchiori, and L. Batina, "Evolutionary methods for the construction of cryptographic boolean functions," in *Genetic Programming - 18th European Conference, EuroGP 2015, Copenhagen, Denmark, April 8-10, 2015, Proceedings*, 2015, pp. 192–204.

[11] S. Chanson, C. Ding, and A. Salomaa, "Cartesian Authentication Codes from Functions with Optimal Nonlinearity," *Theor. Comput. Sci.*, vol. 290, no. 3, pp. 1737–1752, Jan. 2003. [Online]. Available: http://dx.doi.org/10.1016/S0304-3975(02)00077-4

[12] C. Carlet, C. Ding, and J. Yuan, "Linear codes from perfect nonlinear mappings and their secret sharing schemes," *IEEE Transactions on Information Theory*, vol. 51, no. 6, pp. 2089–2102, June 2005.

[13] C. Carlet and C. Ding, "Highly Nonlinear Mappings," *J. Complex.*, vol. 20, no. 2-3, pp. 205–244, Apr. 2004. [Online]. Available: http://dx.doi.org/10.1016/j.jco.2003.08.008

[14] R. Forrié, "The Strict Avalanche Criterion: Spectral Properties of Boolean Functions and an Extended Definition," in *Advances in Cryptology - CRYPTO' 88*, ser. Lecture Notes in Computer Science, S. Goldwasser, Ed. Springer New York, 1990, vol. 403, pp. 450–468.

[15] C. Carlet, "Vectorial Boolean Functions for Cryptography," in *Boolean Models and Methods in Mathematics, Computer Science, and Engineering*, 1st ed., Y. Crama and P. L. Hammer, Eds. New York, NY, USA: Cambridge University Press, 2010, pp. 398–469.

[16] K. Nyberg, "Perfect Nonlinear S-Boxes," in *Advances in Cryptology - EUROCRYPT '91, Workshop on the Theory and Application of of Cryptographic Techniques, Brighton, UK, April 8-11, 1991, Proceedings*, ser. Lecture Notes in Computer Science, vol. 547. Springer, 1991, pp. 378–386.

[17] J. McLaughlin and J. A. Clark, "Evolving balanced Boolean functions with optimal resistance to algebraic and fast algebraic attacks, maximal algebraic degree, and very high nonlinearity," Cryptology ePrint Archive, Report 2013/011, 2013, http://eprint.iacr.org/.

[18] S. Picek, D. Jakobovic, and M. Golub, "Evolving Cryptographically Sound Boolean Functions," in *Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation*, ser. GECCO '13 Companion. New York, NY, USA: ACM, 2013, pp. 191–192.

[19] R. Hrbacek and V. Dvorak, "Bent Function Synthesis by Means of Cartesian Genetic Programming," in *Parallel Problem Solving from Nature - PPSN XIII*, ser. Lecture Notes in Computer Science, T. Bartz-Beielstein, J. Branke, B. Filipič, and J. Smith, Eds. Springer International Publishing, 2014, vol. 8672, pp. 414–423.

[20] S. Picek, D. Jakobovic, J. F. Miller, L. Batina, and M. Cupic, "Cryptographic boolean functions: One output, many design criteria," *Applied Soft Computing*, vol. 40, pp. 635 – 653, 2016.

[21] J. A. Clark, J. Jacob, S. Maitra, and P. Stănică, "Almost Boolean functions: the design of Boolean functions by spectral inversion," in *Evolutionary Computation, 2003. CEC '03. The 2003 Congress on*, vol. 3, Dec 2003, pp. 2173–2180 Vol.3.

[22] L. Mariot and A. Leporati, "A Genetic Algorithm for Evolving Plateaued Cryptographic Boolean Functions," in *Theory and Practice of Natural Computing - Fourth International Conference, TPNC 2015, Mieres, Spain, December 15-16, 2015. Proceedings*, 2015, pp. 33–45.

[23] S. Picek and D. Jakobovic, "Evolving Algebraic Constructions for Designing Bent Boolean Functions," in *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference, Denver, CO, USA, July 20 - 24, 2016*, 2016, pp. 781–788.

[24] S. Picek, D. Sisejkovic, and D. Jakobovic, "Immunological algorithms paradigm for construction of Boolean functions with good cryptographic properties," *Engineering Applications of Artificial Intelligence*, 2016.

[25] J. A. Clark, J. L. Jacob, and S. Stepney, "The design of S-boxes by simulated annealing," *New Generation Computing*, vol. 23, no. 3, pp. 219–231, Sep. 2005.

[26] W. Millan, L. Burnett, G. Carter, A. Clark, and E. Dawson, "Evolutionary Heuristics for Finding Cryptographically Strong S-Boxes," in *Information and Communication Security*, ser. Lecture Notes in Computer Science, V. Varadharajan and Y. Mu, Eds. Springer Berlin Heidelberg, 1999, vol. 1726, pp. 263–274.

[27] P. Tesař, "A New Method for Generating High Non-linearity S-Boxes," *Radioengineering*, vol. 19, no. 1, pp. 23–26, Apr. 2010.

[28] S. Picek, B. Ege, L. Batina, D. Jakobovic, L. Chmielewski, and M. Golub, "On Using Genetic Algorithms for Intrinsic Side-channel Resistance: The Case of AES S-box," in *Proceedings of the First Workshop on Cryptography and Security in Computing Systems*, ser. CS2 '14. New York, NY, USA: ACM, 2014, pp. 13–18.

[29] S. Picek, B. Mazumdar, D. Mukhopadhyay, and L. Batina, "Modified Transparency Order Property: Solution or Just Another Attempt," in *Security, Privacy, and Applied Cryptography Engineering - 5th International Conference, SPACE 2015, Jaipur, India, October 3-7, 2015, Proceedings*, 2015, pp. 210–227.

[30] S. Picek, M. Cupic, and L. Rotim, "A New Cost Function for Evolution of S-Boxes," *Evolutionary Computation*, vol. 24, no. 4, pp. 695–718, 2016.

[31] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing*, 2nd ed. Springer-Verlag, Berlin Heidelberg New York, USA, 2015.

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

[33] R. Poli, W. B. Langdon, and N. F. McPhee, *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008, (With contributions by J. R. Koza).