Evolving Algebraic Constructions for Designing Bent Boolean Functions

Stjepan Picek
KU Leuven, ESAT/COSIC and iMinds
Kasteelpark Arenberg 10, bus 2452, B-3001
Leuven-Heverlee, Belgium and
LAGA, UMR 7539, CNRS, Department of
Mathematics, University of Paris 8, France
stjepan@computer.org

Domagoj Jakobovic
Dept. of Electronics, Microelectronics, Computer
and Intelligent Systems,
Faculty of Electrical Engineering and Computing
University of Zagreb, Croatia
domagoj.jakobovic@fer.hr

ABSTRACT

The evolution of Boolean functions that can be used in cryptography is a topic well studied in the last decades. Previous research, however, has focused on evolving Boolean functions directly, and not on general methods that are capable of generating the desired functions. The former approach has the advantage of being able to produce a large number of functions in a relatively short time, but it directly depends on the size of the search space. In this paper, we present a method to evolve algebraic constructions for generation of bent Boolean functions. To strengthen our approach, we define three types of constructions and give experimental results for them. Our results show that this approach is able to produce a large number of constructions, which could in turn enable the construction of many more Boolean functions with a larger number of variables.

Keywords

Boolean Functions; Cryptography; Genetic Programming; Algebraic Constructions; Evolution; Secondary Constructions

1. INTRODUCTION

Boolean functions play an important role in several areas like coding theory, sequences, and **cryptography**. Cryptography can be defined as a science of secret writing with the goal of hiding the meaning of a message [21]. To ensure that goal (note there exist other relevant goals like authenticity or integrity, but they are out of scope in this research) one uses cryptographic algorithms, commonly known as **ciphers**. When all parties that participate in a secure communication use the same key (i.e., the parameter that determines the functional output of a cipher), we talk about symmetric key cryptography [15]. To encode a message (commonly known as plaintext) into a ciphertext, one uses **encryption** transformation and to revert the ciphertext back to plaintext one uses **decryption** transformation.

Symmetric key cryptography is usually divided into **block ciphers** and **stream ciphers**. The main difference between block ciphers and stream ciphers is in the way how they encrypt/decrypt the data. In block ciphers, encryption and decryption transformation is difficult while in stream ciphers that transformation is easy and changes for every symbol [8].

We concentrate now on stream ciphers. Stream ciphers usually work by producing a keystream that is added modulo two (XOR) with plaintext bits. To obtain such a keystream, one well researched way is to employ linear feedback shift register (LFSR). However, the output from an LFSR is linear and there exist easy cryptanalysis techniques against it [17]. To add nonlinearity to the cipher (and consequently make the cryptanalysis more difficult) one can for instance add one or more Boolean functions. Two well explored approaches are to use combiner or filter generators. In a combiner generator, outputs from several LFSRs serve as an input to a Boolean function. In a filter generator, the output is obtained by a nonlinear combination of a number of positions in one longer LFSR [5].

For a Boolean function to be useful in such constructions, it needs to satisfy a number of properties. One such property is the nonlinearity where the higher the value the more nonlinearity there is. Informally speaking, the nonlinearity property tells us how far is a function from all affine functions and therefore how difficult is to conduct the cryptanalysis. Boolean functions that have maximal nonlinearity are called **bent** functions. Although they are not balanced (i.e., their truth tables do not have the same number of zeros and ones) and therefore not suitable for cryptography, there are methods how to transform bent Boolean functions into balanced Boolean functions with high nonlinearity [5].

To build Boolean functions one has on his disposal three options: algebraic constructions, heuristics, and random search [25]. Naturally, it is also possible to combine the aforesaid techniques [25].

In this paper, we present a novel approach how to evolve bent Boolean functions that combines heuristics and algebraic constructions. Instead of evolving Boolean functions (e.g., their truth tables), we evolve algebraic constructions that are then used to generate bent Boolean functions. Therefore, here we evolve "generators" of Boolean functions. Such generators when provided with the adequate input, result in new bent Boolean functions that have larger number of

GECCO '16, July 20-24, 2016, Denver, CO, USA
© 2016 ACM. ISBN 978-1-4503-4206-3/16/07…$15.00
DOI: http://dx.doi.org/10.1145/2908812.2908915

variables. We note that we believe this technique can be extended to evolution of different types of algebraic constructions and Boolean functions with different properties.

In order to evolve algebraic constructions, we employ evolutionary algorithms (EAs), and more precisely Genetic Programming (GP). We show that this approach has many advantages, the main one being the ease of generation of larger Boolean functions. As such, we expect to open a new line of research on the evolution of Boolean functions for cryptography.

1.1 Motivation and Contributions

When considering heuristics, and more specifically evolutionary algorithms (EAs) to build Boolean functions with properties desirable for cryptographic usages, there exist a plethora of works as presented in Sec. 3. However, as far as we are aware, there exist no research up to now that considers evolving algebraic constructions of Boolean functions. We clarify the difference briefly. Evolutionary algorithms start with a number of individuals (Boolean functions) and evolve them in order to obtain properties as given in the fitness function. The expected end result is one or more Boolean functions that have good cryptographic properties. The examples from literature point us that this principle works very well. In fact, it is not difficult to observe that in most of the instances the results for heuristics are as good as those obtained with algebraic constructions.

However, heuristic approach has some drawbacks. These drawbacks are the search space size (coupled with the usual solution representation as a string of bits) and the evaluation cost. To better understand the size of the search space, we note that for a Boolean function with n inputs there are in total 2^{2^n} Boolean functions. Furthermore, to present a Boolean function as a string of bits we need a vector of size 2^n.

We can see that the search space size grows exponentially and it becomes difficult to work even with Boolean functions with a moderate number of inputs. However, there are some algorithms that are better suited for such large problems and there is no reason why not to use them. Furthermore, if the memory or the computational complexity is too high, one can use a single solution-based metaheuristics instead of the population-based metaheuristics [29]. Next, when the encoding of solutions in the truth table form becomes impractical (i.e., too large), one can use for instance Genetic Programming (GP) or Cartesian Genetic Programming (CGP) since the encoding is much more efficient in those cases. Therefore, although significant, the aforesaid problems can be resolved with a more appropriate choice of algorithms or encodings.

However, the evaluation cost of the fitness function is a problem that cannot be easily solved. If we consider only the nonlinearity property and a Boolean function with eight inputs, then to evaluate it we need up to a few milliseconds. If we want to evaluate the same property for a Boolean function with 16 inputs, this can now take several hundred milliseconds. Finally, if we evaluate a Boolean function with 24 inputs, it will last around 10 seconds.

One can ask if it is realistic to use such large Boolean functions. We consider it is, since it is known that to be able to withstand some cryptanalytic attacks, a Boolean function needs to have at minimum 13 inputs [5]. If the evaluation lasts several seconds, this still represents no problem for modern computers. However, let us now consider some more computationally complex properties – the algebraic immunity and the fast algebraic immunity properties [1, 7], which are properties one must consider when using Boolean functions in stream ciphers. Those properties for larger Boolean functions can easily take from several minutes to even several hours on a modern computer (for these measurements we used a PC with Intel i5 3470 processor and 6 GB of RAM). Therefore, if we try to evolve a Boolean function that has realistic size and good cryptographic properties (including the aforesaid two properties), the evaluation part will be often a bottleneck that is impossible to surpass. To conclude, to evolve such functions would be an extremely difficult and long process.

Therefore, in this work we aim to evolve algebraic constructions, which we believe is an approach with a number of benefits. Furthermore, to succeed in our investigation, we devise a model how to encode solutions and the appropriate fitness function.

Finally, we define several types of constructions that we believe can help in future research in order to better evaluate the efficiency of this technique.

1.2 Outline of the Paper

This paper is organized as follows. In Section 2, we first discuss how to represent Boolean functions and calculate the properties of interest. Furthermore, we elaborate on different types of constructions of Boolean functions (with an emphasis on bent Boolean functions). In Section 3, we give a short list of related work. Section 4 gives details about the GP approach we use and Section 5 presents the results of our experiments. In Section 6, we discuss the ramifications of our approach and we offer a possible roadmap for future work. Finally, in Section 7, we give a short conclusion.

2. ON BOOLEAN FUNCTIONS

In this section, we start with the background information on Boolean functions representations and properties and later we discuss various construction possibilities. However, first we give the notation we use in the rest of the paper.

Let $n, m \in \mathbb{N}$. The set of all n-tuples of the elements in the field $\mathbb{F}_2$ is denoted as $\mathbb{F}_2^n$. Here, $\mathbb{F}_2$ represents the Galois field with two elements. The set $\mathbb{F}_2^n$ represents all binary vectors of length n [5]. The inner product of vectors $\vec{a}$ and $\vec{b}$ is denoted as $\vec{a} \cdot \vec{b}$ and it equals $\vec{a} \cdot \vec{b} = \oplus_{i=1}^{n} a_i b_i$ with "$\oplus$" being addition modulo two.

The Hamming weight (HW) of a vector $\vec{a}$, where $\vec{a} \in \mathbb{F}_2^n$, is the number of non-zero positions in the vector. An (n, m)-function is any mapping F from $\mathbb{F}_2^n$ to $\mathbb{F}_2^m$ where Boolean functions represent $m = 1$ case.

2.1 Boolean Function Representations

There are several unique representations of Boolean functions where each one has its advantages and drawbacks. Here, we are interested in two unique representations, namely, the truth table representation and the Walsh-Hadamard representation.

A Boolean function f on $\mathbb{F}_2^n$ can be represented by a **truth table** (TT), which is a vector $(f(\vec{0}), ..., f(\vec{1}))$ that contains the function values of f, ordered lexicographically [5].

A Boolean function f can be represented by the **Walsh-Hadamard transform** W_f. The Walsh-Hadamard transform measures the correlation between $f(\vec{x})$ and the linear function $\vec{a} \cdot \vec{x}$ [5, 11]. The Walsh-Hadamard transform of a Boolean function f equals:

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

2.2 Boolean Function Properties

A Boolean function with n inputs is **balanced** if the Hamming weight of its truth table equals 2^{n-1}. Alternatively, a Boolean function f is balanced if the Walsh-Hadamard spectrum of a vector $\vec{0}$ equals zero [27]:

$$W_f(\vec{0}) = 0. \tag{2}$$

A Boolean function should lie at a large Hamming distance from all affine functions and the nonlinearity N_f of a Boolean function f is the minimum Hamming distance between the function f and affine functions [5]. The **nonlinearity** N_f of a Boolean function f expressed in terms of the Walsh-Hadamard coefficients equals [5]:

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

A Boolean function is **bent** if it has maximal nonlinearity that equals [5, 9, 28]:

$$N_f = 2^{n-1} - 2^{\frac{n}{2}-1}. \tag{4}$$

Alternatively, a Boolean function is bent if its Walsh-Hadamard spectrum is flat, i.e., if the spectrum has a value of $2^{\frac{n}{2}}$ for all $\vec{x} \in \mathbb{F}_2^n$. Bent functions are never balanced and they exist only if n is even. Since the values of bent functions are not uniformly distributed, they are not appropriate for direct usage in cryptography. However, as already mentioned, it is possible to transform bent functions into balanced Boolean functions with high nonlinearity [5].

To provide an insight on the number of bent Boolean functions, we give expressions for lower and upper bound. Note that in general, there are no known efficient bounds for an n-variable bent Boolean function. The lower bound equals:

$$lower\_bound = 2^{2^{\frac{n}{2}+log_2(n-2)-1}}, \tag{5}$$

and the upper bound equals [5]:

$$upper\_bound = 2^{2^{n-1}+\frac{1}{2}\binom{n}{n/2}}. \tag{6}$$

It is easy to calculate those values where we see that bent functions present only a small fraction of the whole solution space. For further information about Boolean functions, we refer interested readers to [5, 6, 9, 10].

2.3 Construction Techniques

Recall, there are three options how to generate Boolean functions: algebraic constructions, random search, and heuristics (and their combinations).

The main strength of algebraic constructions is that it can be proved they will generate functions with certain properties and is in general equally easy to construct functions of any dimension. The main drawback lies in the fact that they always result in the same functions (since they are deterministic) which means one is limited in a number of different functions one can obtain. A more general drawback is that it is usually quite difficult to devise a good algebraic construction, i.e., one that results in Boolean functions with desired cryptographic properties.

The main advantages of random search are that it produces an abundance of different Boolean functions and is a relatively fast method. However, the quality of such functions (with regards to their cryptographic properties) is most often suboptimal.

Finally, when discussing heuristic methods, they are usually positioned somewhere between the two aforesaid approaches: they generate a large number of good results in a relatively short time. However, as explained in Sec. 1.1, there are some drawbacks when considering the search space size and the evaluation cost.

It is also possible to divide construction techniques into **primary** constructions and **secondary** constructions. In primary constructions, one obtains new functions without using known functions. In secondary constructions, one uses already known functions to construct new functions [5].

Algebraic constructions can be either primary or secondary constructions. One example of a primary algebraic construction technique is the Maiorana-McFarland ($\mathcal{M}$) class [6]. There, $(\mathbb{F}_2^m)^2 \rightarrow \mathbb{F}_2$ of the form $f(\vec{x}, \vec{y}) = \vec{x} \cdot \vec{y} \oplus h(\vec{y})$, where h is any Boolean function with m inputs.

As an example of a secondary algebraic construction, we give the Rothaus construction [6, 10]. Let h_1, h_2, and h_3 be three bent functions with n inputs. Next, $h_1 \oplus h_2 \oplus h_3$ must give a bent function also. Then, to generate a bent function in $n+2$ variables one needs to calculate:

$$\begin{aligned} f(\vec{x}, x_{n+1}, x_{n+2}) = {} & h_1(\vec{x})h_2(\vec{x}) \oplus h_1(\vec{x})h_3(\vec{x}) \\ & \oplus h_2(\vec{x})h_3(\vec{x}) \oplus [h_1(\vec{x}) \oplus h_2(\vec{x})]x_{n+1} \\ & \oplus [h_1(\vec{x}) \oplus h_3(\vec{x})]x_{n+2} \oplus x_{n+1}x_{n+2}. \end{aligned} \tag{7}$$

For more details about primary and secondary algebraic constructions of bent Boolean functions, we refer interested readers to [3, 4, 10].

When discussing random search, we see it is a primary construction technique since it is not possible to start with some already generated functions.

On the other hand, heuristics are more complicated to classify despite the fact that it is usually considered they can serve both as the primary and the secondary construction technique [24]. For the sake of the simplicity, let us consider evolutionary algorithms as a representative of heuristic techniques. Since EAs always start with a population of solutions, they are in fact secondary construction techniques. Naturally, one can argue that it is just a set of random individuals where it is not important what properties those individuals have and from that perspective, we could include EAs as some sort of primary construction techniques. In the scenarios where the initial population for EAs is not randomly selected, but contains previously defined individuals, then EAs are for sure secondary construction techniques.

In this paper, we use the Rothaus construction as a motivation for our approach, where we aim to evolve **secondary algebraic constructions** that use four n-inputs bent Boolean functions to produce one $n+2$-input bent Boolean function.

Note that we use four bent functions without any additional constraints on those functions instead of three bent functions where their addition modulo two results in a bent function (as in the Rothaus method). We follow the former approach since it is easier to generate bent functions without the constraint that the result of the exclusive OR operation is also a bent function.

3. RELATED WORK

As already mentioned, there is a plenitude of papers dealing with heuristic generation of Boolean functions. However, since none of those papers aim to find algebraic construction methods for Boolean functions, we consider them related only with regards to the final goal, but not with regards to the construction techniques used. Therefore, here we give a subset of relevant work that investigates the evolution of (bent) Boolean functions with EAs.

We note that the first paper, as far as we know, that employs evolutionary algorithms to evolve cryptographic Boolean functions dates back to 1997. There, Millan et al. experimented with genetic algorithms (GAs) to evolve Boolean functions with high nonlinearity [18].

Millan, Clark, and Dawson used GAs to evolve Boolean functions that have high nonlinearity [19]. They used a combination of a GA and hill climbing together with a resetting step in order to find Boolean functions with high nonlinearity for sizes of up to 12 inputs.

Millan, Fuller, and Dawson proposed a new adaptive strategy for a local search algorithm for the generation of Boolean functions with high nonlinearity [20]. Additionally, they introduced the notion of the graph of affine equivalence classes of Boolean functions.

Izbenko et al. used a modified hill climbing algorithm to transform bent functions to balanced Boolean functions with high nonlinearity [14].

Picek, Jakobovic, and Golub experimented with GA and GP to find Boolean functions that possess several optimal properties [22]. As far as the authors know, this is the first application of GP for evolving cryptographically suitable Boolean functions.

Hrbacek and Dvorak used CGP to evolve bent Boolean functions of sizes up to 16 inputs [13] where the authors experimented with several configurations of algorithms in order to speed up the evolution process. They did not limit the number of generations and they succeeded in finding bent function in each run for sizes between 6 and 16 inputs.

Mariot and Leporati used GAs where the genotype consists of the Walsh-Hadamard values in order to evolve semi-bent (plateaued) Boolean functions [16].

A detailed analysis of the efficiency of a number of evolutionary algorithms when evolving Boolean functions satisfying different criteria is given in [23].

4. GENETIC PROGRAMMING APPROACH

In this section, we describe two approaches for generating Boolean functions with GP; the first one using only the Boolean variables as inputs (Sec. 4.1), and the second one which relies on the predefined Boolean functions as parts of the construction (Sec. 4.2). In all experiments, we use a configuration with Intel i5 3470 processor and 6 GB of RAM.

4.1 Evolving General Boolean Functions

Genetic Programming has already been extensively used in the evolution of Boolean functions as indicated in Sec. 3. Furthermore, GP and its variants (most notably Cartesian Genetic Programming) have been proven to be able to produce human-competitive results in this domain. In this work, we use GP to evolve a function in the form of a tree. Each tree is evaluated (e.g., for the nonlinearity property) according to the truth table representation it produces. The terminal set is in this case comprised of a given number of Boolean variables, which we denote $v_0, v_1, ..., v_n$.

The function set consists of several Boolean primitives necessary to represent any Boolean function. Here, we use the following function set: OR, XOR, AND, XNOR, and AND with the second input inverted, each of which take two input arguments. It is of course possible to use only a subset of those, but here we do not impose any implementation constraints.

4.2 Evolving Boolean Constructions

A slightly different approach is taken to evolve constructions with GP. Here, we presume the existence of a certain number of predefined Boolean functions which are included in the terminal set. In our experiments, four predefined Boolean functions are available as terminals, which are denoted with f_0, f_1, f_2, and f_3 (input functions). Additionally, the terminal set includes two Boolean variables, v_0 and v_1, while the function set remains the same.

With these parameters, if the number of variables of predefined input functions is n, the resulting construction (a GP tree) represents a new Boolean function with $n+2$ variables. With the goal to obtain a bent function of size $n+2$, the input functions are presumed to be bent themselves.

To be able to use this approach, the initial input functions must be given or evolved with the general GP method. In this paper, we obtain the initial set of bent functions with the general method (see Sec. 4.1), starting with a low number of variables (e.g., four variables), which is trivial to find. Then, the input functions are used to build constructions for a larger number of variables (e.g., six variables). The evolved constructions can be decoded and stored as a truth table. In each subsequent step, the outputs of the current stage in the form of a truth table may then be used as input functions in the next stage.

4.3 Fitness Function

With the goal of maximizing the nonlinearity property, the simplest approach is just to use the actual nonlinearity value N_f (see Eq. (3)) as the fitness function. However, this value represents only the single maximal value of the whole Walsh-Hadamard spectrum, and consequently there exist many different Boolean functions, with radically different Walsh-Hadamard spectra, that have the same nonlinearity value. This fact makes it more difficult for any optimization algorithm to move from one nonlinearity level to the next one.

Therefore, we use fitness function in a way that adds more information about the quality of a certain solution with regards to the Walsh-Hadamard spectrum:

$$fitness_1 = 2^n - freq(|W_f(\vec{a})|), \tag{8}$$

where $freq(|W_f(\vec{a})|)$ is the number of times the Walsh-Hadamard coefficient $W_f(\vec{a})$ value differs from the $n^{\frac{n}{2}}$ value. We see that in the case of a bent Boolean function, the fitness value equals 2^n.

The above fitness function aims to maximize the nonlinearity, and it is the function we use for evolving bent Boolean functions directly, with the approach in Sec. 4.1. However, when evolving Boolean constructions, we also aim to obtain a more general construction which will be able to produce a new bent function for every *combination* of input bent functions of lower order. For this purpose, we evaluate the constructions with several *groups* of input bent functions, which in each group define the values of terminals $f_0 - f_3$.

In our experiments, we use four groups of four input functions, where for each group i the value of Eq. (8) is calculated with the same tree. The resulting fitness for the evaluated construction is simply the sum of obtained values for each group:

$$fitness_2 = \sum_{i=1}^{4} fitness_{1,i}. \quad (9)$$

The optimal value a construction can attain is the maximum value of $fitness_1$ multiplied by four, which means that the construction produced a bent function of $n+2$ bits with every group of input bent functions of size n. This approach, of course, does *not* guarantee that the evolved construction will be general, i.e., that it will produce a bent function for every set of input bent functions. However, in all our experiments and in all subsequent analysis with different input groups, this has always shown to be true.

Finally, we note that when evolving constructions, the obtained trees with maximal fitness always include the two Boolean variables v_0 and v_1, but not necessarily the whole set of input functions $f_0 - f_3$. Since we said that we take as an inspiration the Rothaus construction (see Sec. 2.3), we need to ensure that there are indeed all four input functions in every construction. Therefore, we add a third configuration, in which a construction is penalized if it does not include all the input terminals. This fitness function is represented with the following equation:

$$fitness_3 = \frac{fitness_2}{1 + missing\_terminals}, \quad (10)$$

which simply equals to $fitness_2$ divided by the number of missing input terminals.

5. EXPERIMENTAL RESULTS

In this section, we present the results of our experiments. In the first part, we illustrate the difficulty of evolving bent Boolean functions of different sizes, while the second part presents the results for evolution of constructions with the same goal.

The GP parameters were the same for all configurations, and are based on our previous experiments as well as guidelines for the similar problems in the existing literature. The GP population size is 500 individuals and the stopping criteria is set to either 500 000 evaluations or to reaching the known optimal fitness value. The selection process is described in the Algorithm 1, where k equals 3 and the individual mutation probability is set to 0.5. The variation operators are simple tree crossover with 90% bias for functional nodes [26] and subtree mutation. If not stated otherwise, every experiment is repeated 30 times. For evolving general Boolean functions, we use a tree depth of 7 and for evolving constructions our experiments show that a tree depth of 5 is sufficient.

5.1 Evolving Bent Boolean Functions

The first part of the results only illustrates the increasing difficulty of the direct evolution of bent Boolean functions

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;
```

Table 1: Direct evolution of bent functions with GP.

Boolean variables	8	12	16	18
Maximal N_f	120	2 016	32 640	130 816
Success rate	100 %	100 %	0 %	0 %
Max. fitness	120	2 016	32 336	130 072
Avg. fitness	120	2 016	32 305.25	130 047

with increasingly larger number of variables. This section is not intended to provide a detailed performance evaluation, but rather to give some intuition on the difficulty of the problem.

Here, the approach in Sec. 4.1 is taken with a terminal set $v_0, v_1, ..., v_n$ where n is the number of Boolean variables. The results are given in Table 1; we calculate fitness as in Eq. (3) with the values in table showing the corresponding nonlinearity values.

The task is relatively simple for smaller sizes of Boolean functions (e.g., eight inputs), where GP is able to reach the optimal solution in under 10 000 evaluations. However, the situation changes drastically with increasing size: already for 16 inputs the success percentage drops to zero, since no optimal solution was found in 30 runs.

The outcome is the same for the 18 bits size, where we note it takes over 30 hours on a single processor to complete 500 000 evaluations. In fact, obtaining maximal nonlinearity already for 16 bits required about 600 seconds on a 40-node parallel environment employing distributed CGP evaluation, which is considered a human competitive result [13]. This shows us that it is possible to evolve bent Boolean functions even for larger dimensions, but the problem is difficult and requires a large number of evaluations.

5.2 Evolving Secondary Algebraic Constructions

In this section, we use secondary algebraic constructions to obtain bent Boolean functions in $n+2$ variables from four bent Boolean functions in n variables. First, we shorty discuss the construction types we expect to obtain from EAs. For the sake of simplicity, all construction types we denote as SEC which stands for Secondary Evolutionary Construction. By a correct result, we consider any bent Boolean function that has correct number of variables.

Definition 1. *Weak SEC is any construction that outputs the correct result for a specific input. However, Weak SEC will not output correct results for different inputs or Boolean function sizes.*

Definition 2. *Strong SEC Type I is any construction that outputs the correct results for an arbitrary number of different input combinations. Strong SEC Type I will not output correct results for different dimensions of Boolean functions.*

Definition 3. *Strong SEC Type II is any construction that outputs the correct results for an arbitrary number of different inputs and an arbitrary dimension of Boolean functions.*

As we can see, each following construction type has more constraints and represents a more difficult target to evolve. Ideally, we are interested in the Strong SEC Type II.

We start with only four Boolean variables, for which one can easily generate optimal solutions (i.e., bent Boolean functions). These are then randomly selected as input functions in evolving constructions leading to increasing number of bits (6, 8, etc.).

First, we briefly discuss what kind of constructions are desirable from a mathematical perspective. Note that instead of constructions, we can also consider the resulting Boolean functions.

The first criterion is that they must not be "trivial", i.e., the resulting Boolean functions cannot differ only by affine functions. The second and more important criterion is whether such functions are new, i.e., not constructed before with different algebraic constructions. We see that the fist criterion is a necessary one, but not a sufficient one, while the second criterion is sufficient for our purpose (since Boolean functions that differ only in affine function are known, and therefore cannot be new). Unfortunately, there is no easy (e.g., automatic) way of checking whether a construction is new, but one needs to compare the results with the results from other constructions.

We emphasize that the simpler construction we can evolve, the better it is, naturally maintaining the criteria we mentioned. Indeed, simpler constructions are easier to prove and faster to implement. Finally, their "simplicity" does not make any difference for the quality of underlying Boolean functions since those constructions generate a bent Boolean function, which is the only requirement we set.

The first test configuration uses only a single group of four input functions, i.e., with fitness as in Eq. (8), which aims to find at least weak constructions. We immediately note that the task of combining terminals as Boolean variables v_0 and v_1 with input bent functions $f_0 - f_3$ is next to trivial for GP, because it always succeeds in evolving a construction of a bent function. This result is achieved with under 1 000 evaluations in every run and seems not to be dependent on the actual Boolean function size (e.g., constructing from 16 to 18 bits is as easy as going from 4 to 6 bits).

The constructions obtained in this way, however, are guaranteed to produce a bent function only with the same four input functions, and are consequently able to give only a single new solution. An example of such solution is, for instance: (((v_1 AND2 v_0) XOR f_3) XOR (f_0 XOR f_1)), which provably results in a bent function only for certain input functions. Therefore, this belongs to the weak constructions, but is probably neither a strong construction of Type I nor Type II.

In the second configuration, we employ the $fitness_2$ as the objective and for each stage we use four different groups of input functions. Only if a construction (a GP tree) is able to produce a new bent function of size $n+2$ with each group of input functions of size n, it is considered optimal. Therefore, in this scenario we aim to reach strong constructions of at least strong SEC Type I.

Again, in this case we note that the GP is able to reach an optimal solution in every run (out of 30), and an optimal construction is always found within 10 000 evaluations. An example of such a construction from four to six bits is: ((v_1 AND2 v_0) XOR f_0), which produces a bent function for each input group. Although we do not prove that this is actually a strong construction of type I, this solution was able to produce a bent function with every input group it was subsequently tested with.

Finally, in the third configuration we use $fitness_3$ to ensure each optimal construction includes all the input terminals. The GP was again able to find an optimal construction in every run, and an example of such a solution is: ((((v_0 AND2 v_1) XOR f_0) XOR (f_1 XOR f_1)) XOR ((f_2 AND f_3) AND (v_0 AND2 f_2))).

However, notice that although presented solution has all terminals, the input function f_1 appears only in the subexpression f_1 XOR f_1, which results in a value of zero. Therefore, although officially conforming to our condition that a construction needs to use all input terminals, such construction cannot be accepted for further evaluation.

To counter this problem, in every evaluation we check whether there are such expressions (f XOR f, f XNOR f, etc.) in the tree and if there are, we do not count input terminals appearing there. Finally, one example of the third configuration where we also check the validity of subexpressions is ((((v1 XNOR f0) OR (f3 AND f0)) XOR ((f1 XOR v0) XNOR v1)) AND2 ((v0 AND2 f2) AND2 ((f0 XNOR f3) XOR (f1 AND2 v1)))).

We proceed with this approach to evolve constructions for bent functions of up to 20 variables with both $fitness_2$ and $fitness_3$. Note that it is not possible to apply the evolution of constructions for 20 bits without having previously defined input functions of 18 bits. That is why for every even number of variables (at each stage), we use the resulting constructions to obtain at least four groups of four bent functions (all 16 different) to use as input functions for the next stage. This turns out to be an easy task, because at each stage we complete 30 runs and have 30 optimal constructions; since each of them was evaluated on four input groups, there are 120 resulting functions to choose from.

We also make sure the selected functions are different; here we note that although it is impossible to predict which functions will be obtained, there is a very high percentage of unique functions among those (over 95%) at each stage.

The cumulative effort to reach the bent functions of 20 variables is therefore dependent on the previous stages. Since in each stage the observed success rate was 100%, this is a very promising way of obtaining solutions for large number of variables. The average computational effort is illustrated in Table 2, where the maximal recorded time needed to obtain an optimal solution in a single run is shown. Although the number of evaluations is roughly the same, there is an unavoidable increase in computation time due to increasing sizes of the truth table representation, which is needed for the evaluation of each potential solution.

Therefore, the total time to obtain a bent function of 20 bits is at least the sum of times in the appropriate column of Table 2. Compared to the regular approach which additionally exhibits a very poor success rate, this is an enormous improvement.

6. DISCUSSION AND FUTURE WORK

The results of the previous section point us to the unmistakable advantage of the construction method when com-

Table 2: Computational effort for evolving optimal constructions.

Boolean variables	$fitness_2$: max. eval	$fitness_2$: max. time	$fitness_3$: max. eval	$fitness_3$: max. time
6	5 000	<1s	6 000	<1s
8	5 000	<2s	6 000	<2s
10	5 000	4s	6 000	4s
12	5 000	4s	6 000	4s
14	5 000	6s	7 000	12s
16	5 000	30s	7 000	40s
18	5 000	42s	8 000	100s
20	5 000	90s	8 000	220s

pared with the general method. The computational effort needed to obtain a certain number of variables is negligible when compared with the time needed for a general method to reach an optimal solution. However, we only show that this is applicable for a certain objective, and that is the evolution of bent functions.

In the proposed method, one should not forget that existing bent functions of a smaller size n are needed to construct functions of size $n + 2$. We show that those are easily obtained, but we are ultimately limited with the number of unique bent functions that can be generated in this way. In a single run, one construction can be obtained, but in our experiments at least four bent functions are needed to reach the solution. Performing multiple runs can of course increase the diversity of the obtained solutions, at the cost of additional computation time. There is also the question of how the initially chosen/evolved input bent functions limit the type of the resulting constructions. However, as long as we are interested only in obtaining a solution with the desired properties, this does not seem to pose a problem.

Finally, although we are able to easily produce different bent functions, the question remains to which type do the evolved constructions belong. The solutions definitely exceed the weak construction type, and probably satisfy the strong Type I requirement, but the Type II is not verified during the evolution in our experiments.

Therefore, in the subsequent analysis, we take a randomly selected 3 out of 30 evolved constructions from each number of bits (6 to 20) and apply the same construction to check whether it will produce a bent function with four groups of input functions, for every number of Boolean variables from 6 to 24. The results show that every tested construction, either evolved using $fitness_2$ or $fitness_3$, succeeds in producing a bent function of up to 24 variables. This is a promising result, since it indicates that the evolved constructions belong to the Strong Type II, and may consequently be used with arbitrary input bent functions for an arbitrary number of variables.

If this is correct, one does not need to evolve new constructions for each larger number of variables at all; instead, a single set of evolved constructions may be used to generate bent functions, say of size n. Then the *same* set of constructions may be used with input bent functions of size n to acquire different bent functions with $n + 2$ variables, etc.

In general, the main question is whether this approach can be used to find constructions with different objectives. The first approach would be to try to evolve secondary constructions that for instance take as inputs *balanced* Boolean functions in n variables and with high nonlinearity and produce as an output balanced Boolean functions in $n + 2$ variables with high nonlinearity. A more difficult approach would be to try to evolve primary algebraic constructions, i.e., those that do not use already constructed functions as input values. This approach is interesting both for bent and balanced Boolean functions.

From the evolutionary perspective, the natural extension of this work is the application of related evolutionary algorithms that can be also used to represent a construction. Here, as the most promising candidate we see the Self Modifying Cartesian Genetic Programming [12].

From the mathematical perspective, this work opens several research fronts. The first one is to prove whether such constructions are valid for infinite class or at least infinite number of solutions [2]. Finally, it would be interesting to explore what constructions produce equivalent Boolean functions [5].

7. CONCLUSIONS

In this paper, we propose a novel way of evolution of Boolean functions. Instead of the direct evolution of Boolean functions (e.g., their truth tables), we evolve algebraic constructions that can be used to generate Boolean functions. Here, we concentrate on secondary constructions that produce bent Boolean functions. Our results show that this approach is highly successful and is independent of the Boolean function size one needs to find.

To better formalize our results, we also give definitions of several constructions types one can expect to find with evolutionary algorithms. The obtained results suggest it is possible to find all construction types, where even the one with the most constraints seems to be easily reachable.

We consider this work only as a first step in a new research direction when considering the evolution of Boolean functions for cryptography.

8. ACKNOWLEDGMENTS

This work has been supported in part by Croatian Science Foundation under the project IP-2014-09-4882. In addition, this work was supported in part by the Research Council KU Leuven (C16/15/058) and IOF project EDA-DSE (HB/13/020).

9. REFERENCES

[1] F. Armknecht, C. Carlet, P. Gaborit, S. Künzli, W. Meier, and O. Ruatta. Efficient Computation of Algebraic Immunity for Algebraic and Fast Algebraic Attacks. In S. Vaudenay, editor, *Advances in Cryptology - EUROCRYPT 2006, St. Petersburg, Russia, 2006. Proceedings*, pages 147–164. Springer Berlin Heidelberg, 2006.

[2] L. Budaghyan, C. Carlet, P. Felke, and G. Leander. An infinite class of quadratic APN functions which are not equivalent to power mappings. In *Information Theory, 2006 IEEE International Symposium on*, pages 2637–2641, 2006.

[3] C. Carlet. On the Secondary Constructions of Resilient and Bent Functions. In K. Feng, H. Niederreiter, and C. Xing, editors, *Coding,*

Cryptography and Combinatorics, pages 3–28. Birkhäuser Basel, Basel, 2004.

[4] C. Carlet. On Bent and Highly Nonlinear Balanced/Resilient Functions and Their Algebraic Immunities. In M. P. C. Fossorier, H. Imai, S. Lin, and A. Poli, editors, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes: 16th International Symposium, AAECC-16, Las Vegas, NV, USA, February 20-24, 2006. Proceedings*, pages 1–28. Springer Berlin Heidelberg, 2006.

[5] C. Carlet. Boolean Functions for Cryptography and Error Correcting Codes. In Y. Crama and P. L. Hammer, editors, *Boolean Models and Methods in Mathematics, Computer Science, and Engineering*, pages 257–397. Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[6] C. Carlet and S. Mesnager. Four decades of research on bent functions. *Des. Codes Cryptography*, 78(1):5–50, Jan. 2016.

[7] N. Courtois. Fast Algebraic Attacks on Stream Ciphers with Linear Feedback. In D. Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 176–194. Springer Berlin Heidelberg, 2003.

[8] J. Daemen. *Cipher and hash function design, strategies based on linear and differential cryptanalysis, PhD Thesis*. K.U.Leuven, 1995.

[9] J. Dillon. A Survey of Bent Functions. Technical report, Reprinted from the NSA Technical Journal. Special Issue, 1972. unclassified.

[10] J. F. Dillon. *Elementary Hadamard difference sets*. PhD thesis, University of Maryland, 1974.

[11] R. Forrié. The Strict Avalanche Criterion: Spectral Properties of Boolean Functions and an Extended Definition. In S. Goldwasser, editor, *Advances in Cryptology - CRYPTO' 88*, volume 403 of *Lecture Notes in Computer Science*, pages 450–468. Springer New York, 1990.

[12] S. Harding, J. F. Miller, and W. Banzhaf. Self Modifying Cartesian Genetic Programming: Fibonacci, Squares, Regression and Summing. In L. Vanneschi, S. Gustafson, A. Moraglio, I. Falco, and M. Ebner, editors, *Genetic Programming: 12th European Conference, EuroGP 2009 Tübingen, Germany, April 15-17, 2009 Proceedings*, pages 133–144. Springer Berlin Heidelberg, 2009.

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

[14] Y. Izbenko, V. Kovtun, and A. Kuznetsov. The design of boolean functions by modified hill climbing method. Cryptology ePrint Archive, Report 2008/111, 2008.

[15] L. R. Knudsen and M. Robshaw. *The Block Cipher Companion*. Information Security and Cryptography. Springer, 2011.

[16] 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*, pages 33–45, 2015.

[17] J. Massey. Shift-register synthesis and BCH decoding. *Information Theory, IEEE Transactions on*, 15(1):122–127, Jan 1969.

[18] W. Millan, A. Clark, and E. Dawson. An Effective Genetic Algorithm for Finding Highly Nonlinear Boolean Functions. In *Proceedings of the First International Conference on Information and Communication Security*, ICICS '97, pages 149–158, London, UK, UK, 1997. Springer-Verlag.

[19] W. Millan, A. Clark, and E. Dawson. Heuristic design of cryptographically strong balanced Boolean functions. In *Advances in Cryptology - EUROCRYPT '98*, pages 489–499, 1998.

[20] W. Millan, J. Fuller, and E. Dawson. New concepts in evolutionary search for Boolean functions in cryptology. *Computational Intelligence*, 20(3):463–474, 2004.

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

[22] 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*, GECCO '13 Companion, pages 191–192, New York, NY, USA, 2013. ACM.

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

[24] 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*, pages 192–204, 2015.

[25] S. Picek, E. Marchiori, L. Batina, and D. Jakobovic. Combining Evolutionary Computation and Algebraic Constructions to Find Cryptography-Relevant Boolean Functions. In T. Bartz-Beielstein, J. Branke, B. Filipič, and J. Smith, editors, *Parallel Problem Solving from Nature - PPSN XIII*, Lecture Notes in Computer Science, pages 822–831. Springer International Publishing, 2014.

[26] 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).

[27] B. Preneel, W. Van Leekwijck, L. Van Linden, R. Govaerts, and J. Vandewalle. Propagation characteristics of Boolean functions. In *Advances in Cryptology - EUROCRYPT '90*, pages 161–173, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[28] O. Rothaus. On "bent" functions. *Journal of Combinatorial Theory, Series A*, 20(3):300 – 305, 1976.

[29] E.-G. Talbi. *Metaheuristics: From Design to Implementation*. Wiley Publishing, 2009.