Evolving Constructions for Balanced, Highly Nonlinear Boolean Functions

Claude Carlet
claude.carlet@gmail.com
University of Bergen
Bergen, Norway

Marko Djurasevic
marko.durasevic@fer.hr
University of Zagreb
Zagreb, Croatia

Domagoj Jakobovic
domagoj.jakobovic@fer.hr
University of Zagreb
Zagreb, Croatia

Luca Mariot
luca.mariot@ru.nl
Radboud University
Nijmegen, The Netherlands

Stjepan Picek
stjepan.picek@ru.nl
Radboud University
Nijmegen, The Netherlands

ABSTRACT

Finding balanced, highly nonlinear Boolean functions is a difficult problem where it is not known what nonlinearity values are possible to be reached in general. At the same time, evolutionary computation is successfully used to evolve specific Boolean function instances, but the approach cannot easily scale for larger Boolean function sizes. Indeed, while evolving smaller Boolean functions is almost trivial, larger sizes become increasingly difficult, and evolutionary algorithms perform suboptimally.

In this work, we ask whether genetic programming (GP) can evolve constructions resulting in balanced Boolean functions with high nonlinearity. This question is especially interesting as there are only a few known such constructions. Our results show that GP can find constructions that generalize well, i.e., result in the required functions for multiple tested sizes. Further, we show that GP evolves many equivalent constructions under different syntactic representations. Interestingly, the simplest solution found by GP is a particular case of the well-known indirect sum construction.

CCS CONCEPTS

• **Software and its engineering** → **Genetic programming**; • **Theory of computation** → **Cryptographic primitives**; • **Security and privacy** → *Block and stream ciphers*; *Mathematical foundations of cryptography*.

KEYWORDS

Evolutionary Algorithms, Boolean Functions, Balancedness, Nonlinearity, Secondary Constructions

ACM Reference Format:
Claude Carlet, Marko Djurasevic, Domagoj Jakobovic, Luca Mariot, and Stjepan Picek. 2022. Evolving Constructions for Balanced, Highly Nonlinear Boolean Functions. In *Genetic and Evolutionary Computation Conference (GECCO '22), July 9–13, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3512290.3528871

GECCO '22, July 9–13, 2022, Boston, MA, USA

ACM ISBN 978-1-4503-9237-2/22/07.
https://doi.org/10.1145/3512290.3528871

1 INTRODUCTION

Evolutionary algorithms (EAs) are successfully applied in various domains like rostering [2], design of neural networks [10], and cryptography [20]. While they provide no guarantee to reach optimal solutions, the obtained solutions are often relevant enough to maintain the claim that EAs are viable approaches for many real-world applications. Considering the cryptography perspective, common examples of successful applications are evolution of Boolean functions [5], evolution of S-boxes [13], attacks on PUFs [3], hardware Trojan detection [25], and side-channel attacks [26]. From those applications, the evolution of Boolean functions is one of the most successful examples. Indeed, various EAs (as discussed in Section 3) have managed to evolve Boolean functions fulfilling diverse cryptographic conditions. Nevertheless, we can recognize one major problem with those works: they evolve specific Boolean functions, but such an approach often does not generalize to Boolean functions with a larger size. To circumvent this problem, one needs to use algebraic constructions that generalize for multiple sizes. There are two types of constructions: primary, where functions are created from scratch, and secondary, where previously constructed functions are used as building blocks [4].

For a Boolean function to be useful in cryptography, it should be large enough (e.g., minimum 13 inputs), be balanced, and have the highest possible nonlinearity. While these conditions seem reasonable, there are no known algebraic constructions to reach larger nonlinearity than what is obtained with simple quadratic functions (while the existence of sporadic functions shows that it is possible to do so). The reasons for this can be multiple. The most obvious reason is that it is not even known what is the best possible nonlinearity for balanced Boolean functions with more than seven inputs. Also, designing a construction is, in general, a difficult task. As one does not know what kinds of constructions are possible, the search space is prohibitively large.

Thus, finding such algebraic constructions would be very useful. Besides being a source of Boolean functions for direct applications in cryptography, any developments would also be highly relevant for the research in Boolean functions and error-correcting codes.

The literature is abundant with examples for primary or secondary constructions of bent Boolean functions. Bent functions have the best possible nonlinearity, but they exist only when the number of variables is even, and moreover they are unbalanced. Some of the existing constructions for bent functions have been

adapted to generate balanced functions with good nonlinearity (see e.g. [4] for an overview). EAs also showed capable of evolving constructions of bent Boolean functions [19]. Still, evolving bent Boolean functions seem to be simpler and less practically relevant as bent functions do not have a direct usage in cryptography.

This paper aims to investigate whether genetic programming (GP) can evolve algebraic constructions that, in turn, provide balanced, highly nonlinear Boolean functions. To the best of our knowledge, this is the first work that addresses such an optimization problem using EAs. In our approach, a candidate construction in the GP population is encoded by a tree whose leaves are either seed functions of n variables with high nonlinearity or additional independent variables. The output of the tree is a $(n+k)-$variable Boolean function (with k being the number of additional variables), which is, in turn, evaluated for its balancedness and nonlinearity. We experimentally test our approach using different sizes for the seed functions, using either 1 or 2 additional variables. Our main findings are as follows:

(1) GP can evolve many optimal constructions that achieve the target nonlinearity for relatively small sizes, with the addition of two variables generally performing better than adding a single one.
(2) In the experiments where GP obtains optimal constructions with full success rate, many solutions turn out to be the same after minimizing the corresponding circuits and checking for pairwise equivalence.
(3) One of the optimal solutions that occur in all considered experiments, with more or less bloated variants, is a particular case of the well-known indirect sum construction [4].

We finally provide a possible explanation for the third finding, which relates to the way the additional variables are used in the constructions. This prompts us with interesting directions for future research on this optimization problem, which we overview in the conclusions of the paper.

2 BACKGROUND

2.1 Notation

Let n be a positive integer, i.e., $n \in \mathbb{N}^+$. We will denote the set of all n-tuples of elements in the finite field $\mathbb{F}_2 = \{0, 1\}$ as $\mathbb{F}_2^n$. We denote the inner product of two vectors a and b by $a \cdot b$, and it equals $a \cdot b = \bigoplus_{i=0}^{n-1} a_i b_i$. Here, "$\oplus$" denotes the addition modulo two (bitwise XOR). The support ($supp$) of a Boolean function f is the set containing the non-zero positions in the truth table representation, i.e., $supp(f) = \{x : f(x) = 1\}$. The Hamming weight $w_H(f)$ of a Boolean function f equals the size of its support.

2.2 Boolean Functions

A Boolean function of n variables is a mapping f from $\mathbb{F}_2^n$ to $\mathbb{F}_2$, and it can be uniquely represented by a truth table. The truth table of f is the vector $(f(0, \cdots, 0), \ldots, f(1, \cdots, 1))$ containing the function values of f, with the input vectors ordered lexicographically.

The Walsh-Hadamard transform W_f is another unique representation of a Boolean function. It measures the correlation between $f(x)$ and the linear functions $a \cdot x$ [4]:

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

A Boolean function f is balanced if its truth table vector has the same number of 0s and 1s, or equivalently if $|supp(f)| = 2^{n-1}$.

The minimum Hamming distance between a Boolean function f and all affine functions $a \cdot x \oplus b$ is called the nonlinearity of f. The nonlinearity Nl_f of a Boolean function f can be expressed in terms of the Walsh-Hadamard coefficients as [4]:

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

The nonlinearity of a Boolean function with n inputs is bounded above by the following inequality:

$$Nl_f \leq 2^{n-1} - 2^{\frac{n}{2}-1}. \quad (3)$$

This bound is usually called the Covering Radius Bound. Note that the so-called *bent* functions satisfy with equality this bound, and therefore they are maximally nonlinear. However, bent functions cannot be balanced and exist only for even n, limiting their applicability in cryptography. When n is odd, the bound given in Eq. (3) cannot be tight and the maximal nonlinearity lies between $2^{n-1} - 2^{\frac{n-1}{2}}$ and $2^{n-1} - 2^{\frac{n}{2}-1}$. Here, $2^{n-1} - 2^{\frac{n-1}{2}}$ is also called the quadratic bound because it is the best nonlinearity achievable by quadratic functions (i.e. functions of algebraic degree 2). The maximum possible nonlinearity for balanced Boolean functions is unknown for all $n > 7$. Table 1 recaps the optimal and best-known nonlinearities values for such functions of several sizes.

If a Boolean function is not balanced, it cannot be used in cryptography as it causes a statistical bias. Similarly, if a Boolean function is not highly nonlinear, it will not provide optimal (or near to optimal) resilience against linear cryptanalysis.

2.3 Construction Techniques

There are three viable options to create Boolean functions: algebraic constructions, random search, and heuristic approaches. The main strength of algebraic constructions is that they generate functions with certain properties, and it is equally easy to construct functions of any number of variables. The main drawback lies in the fact that they are deterministic and always result in the same functions up to affine equivalence (that is, up to the composition by an affine automorphism), which means the number of different functions one can obtain is limited. Furthermore, it is quite difficult to devise an algebraic construction that results in Boolean functions with the desired properties. Heuristic methods are known to generate a large number of good results in a relatively short time. However, the search space size grows exponentially with the number of variables, and it is difficult to work even with a moderate number of inputs.

The construction techniques can be divided into *primary* constructions and *secondary* constructions. In primary constructions, one obtains new functions without using known ones. In secondary constructions, one uses existing functions to construct new ones [4].

The Rothaus construction represents an example of a secondary algebraic construction [6]. Let h_1, h_2, and h_3 be three bent functions with n inputs, with $h_1 \oplus h_2 \oplus h_3$ also being a bent function.

Table 1: Best known nonlinearity values for balanced Boolean functions. Entries in bold are optimal.

Variables	4	5	6	7	8	9	10	11	12	13	14	15	16
Max NL	**4**	**12**	**26**	**56**	116	240	492	992	2012	4036	8120	16272	32638

A new bent function of $n+2$ variables is generated as:

$$f(x, x_{n+1}, x_{n+2}) = h_1(x)h_2(x) \oplus h_1(x)h_3(x) \oplus h_2(x)h_3(x) \oplus [h_1(x) \oplus h_2(x)]x_{n+1} \oplus [h_1(x) \oplus h_3(x)]x_{n+2} \oplus x_{n+1}x_{n+2}. \quad (4)$$

This kind of construction is a motivation for our approach, where we aim to evolve secondary algebraic constructions that use a number of n-variable highly nonlinear balanced Boolean functions to produce either $(n+1)$ or $(n+2)$-input balanced functions having high nonlinearity.

3 RELATED WORKS

The history of using evolutionary algorithms to evolve Boolean functions with good cryptographic properties is already 25 years long. As far as we know, the first work using EAs for Boolean functions with specific cryptographic properties happened in 1997. There, the authors used genetic algorithms to evolve Boolean functions with high nonlinearity [15]. Next, various algorithms were tested to obtain even better results. For instance, Millan et al. used GA in combination with hill climbing and a resetting step to evolve highly nonlinear Boolean functions up to 12 inputs [16]. On the other hand, Clark and Jacob used simulated annealing and hill-climbing with a cost function motivated by the Parseval theorem to find Boolean functions with high nonlinearity and low autocorrelation [7]. Aguirre et al. were the first to consider multi-objective optimization for this problem [1]. The authors used a random bit climber to find balanced, highly nonlinear Boolean functions.

Picek et al. considered various EAs (genetic algorithms and genetic programming) to find Boolean functions that fulfill multiple cryptographic properties [21]. Mariot and Leporati experimented with Particle Swarm Optimization [9] to find Boolean functions with good trade-offs of cryptographic properties for sizes up to 12 inputs [12]. Picek et al. investigated various immunological algorithms to evolve highly nonlinear Boolean functions up to 16 inputs [22]. Clark et al. [8] pioneered the spectral inversion approach where pseudo-Boolean functions are represented by Walsh spectra that satisfy good cryptographic properties; the optimization objective is to find a spectrum that corresponds to a true Boolean function. Mariot and Leporati [11] further investigated this approach by proposing a genetic algorithm to evolve such spectra.

The above-listed works make only a small part of the research done but show how most of the works manage to find highly fit Boolean functions (whatever the properties required). Still, there was always the problem of using such computationally heavy approaches for Boolean functions with more inputs or cryptographic properties that are more expensive to evaluate. Additionally, such approaches resulted in specific Boolean function instances, running searches for every new size required.

Picek and Jakobovic considered an approach where instead of evolving Boolean functions, they evolved constructions resulting in Boolean function with the required properties [19]. They used genetic programming to evolve secondary algebraic constructions of bent (thus, imbalanced but maximally nonlinear) Boolean functions [19]. Carlet et al. used genetic programming to improve Boolean functions obtained through algebraic constructions [5]. This approach resulted in Boolean functions obtained through the Hidden Weight Boolean Function construction with higher nonlinearity than previously known. Finally, Mariot et al. [14] investigated a secondary construction based on cellular automata (CA), using evolutionary strategies to search for CA local rules that result in bent and semi-bent functions when plugged into the construction.

For a more detailed overview of EAs and Boolean functions in cryptography, we refer interested readers to [20].

4 METHODOLOGY

This section describes how to evolve Boolean functions from scratch using GP and then evolve secondary constructions that rely on predefined Boolean functions in a smaller size.

4.1 Evolving Boolean Functions with GP

GP and its variants (most notably Cartesian Genetic Programming [17]) have already been extensively used in the evolution of Boolean functions as indicated in Sec. 3 and have been able to produce human-competitive results. As a baseline approach, we use GP to evolve a function in the symbolic form, using a tree representation. According to the truth table it produces, each tree is evaluated for the nonlinearity property. The terminal set is comprised of a given number of Boolean variables, which we denote with $v_0, v_1, ..., v_{n-1}$. The function set consists of several Boolean primitives, which can be used to represent any Boolean function. Our experiments use the following function set: OR, XOR, AND, AND2, XNOR, and function NOT that takes a single argument. The function AND2 behaves the same as the function AND but with the second input inverted. Additionally, we included the function IF, which takes three arguments and returns the second one if the first one evaluates to true and the third one otherwise.

4.2 Evolving Boolean Constructions

To evolve constructions with GP, we take a slightly different approach. Firstly, we presume the existence of a certain number of predefined Boolean functions (seed functions) that are included in the terminal set. In our experiments, up to four predefined Boolean functions are available as terminals, which are denoted with f_0, f_1, f_2, and f_3. The number of variables of seed functions is taken to be n, and they are given by their truth tables. Additionally, the terminal set includes a number of independent Boolean variables; if a single variable is added (v_0), then the resulting construction (a GP tree) represents a new Boolean function with $n+1$ variables. Likewise, with two Boolean variables, v_0 and v_1, the construction obtains an $(n+2)$-variable function. The function set remains the same as

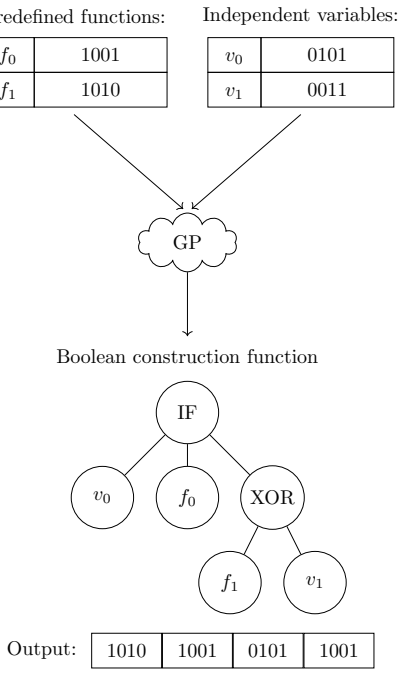


Figure 1: Outline of evolving Boolean constructions (2 seed functions of 2 variables, construction resulting in $n + 2 = 4$ variable Boolean function)

in the previous method. Figure 1 shows the outline of the entire construction process, in which the GP based on a set of predefined functions and input variables constructs a Boolean function.

To be able to apply this approach, the seed functions must be given or previously evolved with the general GP method. Since we optimize for high nonlinearity, the seed functions are presumed to be optimally (where possible) or highly nonlinear. The initial set of seed functions is obtained with the general method, starting with a low number of variables (e.g., four variables), which is trivial to find. Then, the seed functions are used to find constructions for a larger number of variables (e.g., six). The evolved constructions can be decoded and stored as a truth table; that way, the outputs of the previous stage may then be used as seed functions in the next stage, in a kind of bootstrap procedure.

4.3 Fitness Functions

The algorithm should find constructions that give balanced, highly nonlinear functions. To facilitate this, we distinguish the *objective value* of the resulting Boolean function and the *fitness value of the construction* that obtained this function.

We use a two-stage objective function in which a bonus equal to the nonlinearity is awarded only to a perfectly balanced function; otherwise, the objective value is only described by the balancedness penalty. The balancedness penalty BAL is defined as the difference up to the balancedness (i.e., the number of bits to be changed to make the function balanced). This difference is included in the objective function with a negative sign to act as a penalty in maximization scenarios. The delta function $\delta_{BAL,0}$ assumes the value one when $BAL = 0$ and is zero otherwise.

$$objective_1 : -BAL + \delta_{BAL,0} \cdot Nl_f. \tag{5}$$

The second objective function extends the first one to consider the whole Walsh-Hadamard spectrum:

$$objective_2 : -BAL + \delta_{BAL,0} \cdot (Nl_f + Indicator). \tag{6}$$

In this expression, the $Indicator$ term is the normalized number of occurrences of the largest nonlinearity value in the whole spectrum (denoted $\#max\_values$). The smaller the number of these largest values, the easier it is for the algorithm to reach the next nonlinearity value: $Indicator = 1 - \frac{\#max\_values}{2^n}$.

When evolving constructions, we aim to obtain a general construction that will be able to produce a highly nonlinear function for every *combination* of seed functions of lower order. For this purpose, we evaluate the constructions with several *groups* of seed functions, where each group consists of different values of terminals $f_0 - f_3$. In our experiments, we use four groups of seed functions, where for each group i, the value of the objective function is calculated with the same tree (i.e., the same construction). The resulting fitness for the evaluated construction is then defined in three ways.

A) The first method considers the objective value obtained with the first seed group; only if the nonlinearity reaches a predefined level (e.g., the best-known value), then the other seed groups are used, and their obtained objective value is added to the first one to obtain the fitness value:

$$fit_1 : val_1 + \delta_{val_1,targetVal} \cdot \sum val_i. \tag{7}$$

B) The second approach sums the objective value obtained by all the seed groups; we denote this as *sum of all groups*:

$$fit_2 : \sum val_i. \tag{8}$$

C) The third method considers the *minimum* objective value among all seed groups, which is maximized as a consequence:

$$fit_3 : \min val_i. \tag{9}$$

Naturally, this approach does not guarantee that the evolved construction will be general; thus, every evolved construction is subsequently evaluated with a separate test set of seed functions.

Finally, we observe 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 $f_0 - f_3$. To find meaningful expressions that can be candidates for general construction (see Sec. 2.3), we need to ensure that all input seed functions are contained in every construction. Therefore, we add a penalty step, in which a construction is penalized if it does not include all the input terminals. This penalization is applied to all the fitness functions and can be represented with the following equation:

$$fitness_i = \frac{fitness_i}{1 + missing\_terminals}, \tag{10}$$

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

Table 2

Parameter description	Parameter value
Number of variables of target Boolean function	5, 6, 7, 8
Independent variables	1, 2
Number of seed functions	2, 4
Number of seed function groups	4
Seed functions type	balanced, bent
Objective value	nonlinearity (5), nonlinearity with spectrum (6)
Type of fitness function	first group (7), sum of all groups (8), minimum of all groups (9)

4.4 Experimental Settings

The parameters for the GP are the same for all configurations and are based on our previous experience, as well as guidelines from the existing literature addressing similar problems. The population size is set to 500, and the maximal tree depth to 5. We employ a steady-state selection operator with a 3-tournament elimination, which in each iteration randomly selects three individuals for the tournament and eliminates the worst one. A new individual is created immediately by crossing over the remaining two from the tournament, which then undergoes mutation with a probability of 0.5. The variation operators used for GP are simple tree crossover, uniform crossover, size fair, one-point, and context preserving crossover [23] (selected at random) and subtree mutation.

Common parameters for all the experiments include the termination condition of 500 000 fitness evaluations. We chose this particular bound because our preliminary tests showed that final solutions are mostly found before reaching this number of evaluations. Finally, each experiment is repeated 30 times.

5 EXPERIMENTAL RESULTS

In this section, we present the results of the described experiments. First, we applied a canonical GP to evolve balanced, highly nonlinear Boolean functions. We limit to functions with a larger number of variables; for $n < 8$, the optimal nonlinearity values are known (see Table 1), and GP has no difficulties in finding functions with this property. For this experiment only, the number of runs was set to 100, and the results are shown in Table 3.

We can see that, for some function sizes, the search always converges to the same level of nonlinearity (i.e., for $n = 9, 11, 17$). In most cases, however, the GP managed to obtain the best-found value only with a lower rate (i.e., one or two out of 100 runs).

For the constructions, the experiments were performed in two phases. In the first phase, we conducted experiments in which we explored all the configuration variants presented in Table 2. For example, to get to construction that results with 8 variables, we used seed functions of both 6 variables (adding two independent ones, $n+2$) and 7 variables (plus a single independent one, $n+1$); the number of seed functions was 2 or 4, and they were either balanced or bent; objective value used two options, and all three fitness functions were tested. In cases where balanced highly nonlinear seed functions were used, their nonlinearity was equal to that from Table 1, since they are relatively easy to obtain with GP. Also, in all cases when using the first fitness function, best known nonlinearity value from Table 1 is used as a $targetVal$ in 9.

The first phase aimed to identify configurations that allow us to obtain "locally optimal" constructions; we define those as the ones that manage to reach the target nonlinearity value for a given number of variables (5-8). The first thing to note is that there is no comparative advantage in using the objective value (5), which only considers nonlinearity; all the experiments show that (6) is never worse and almost always a better choice. Secondly, it seems that constructions of the form $n + 1$ are generally worse than the ones introducing two variables ($n + 2$), as locally optimal solutions are seldomly found in the former. Using bent seed functions produces the same results as the balanced seeds for target sizes 5 and 6 but obtains worse results in sizes 7 and 8. Finally, when only 2 input seed functions are used, the first fitness function does not find locally optimal construction; otherwise (for 4 seed functions), there is no difference between them. It is interesting that no construction has produced functions with nonlinearity 26 (which is optimal) for 6 variables. This is surprising since those functions can be found relatively easily with a search-based approach.

In the second phase, we take all the configurations that were able to produce locally optimal constructions and evaluate them on a separate test set of seed functions. The test set comprises of 8 groups of balanced highly nonlinear functions, obtained either with search-based GP (up to 13 variables) or with the constructions themselves, using a bootstrap approach. In this phase, the *same* construction (taken from a single run) is used to produce the resulting Boolean function whose nonlinearity is then evaluated on the whole range of variables from $n = 6$ to $n = 18$.

In this experiment, it became evident that the $n+1$ constructions we found are not general since they do not reach target values where the number of variables is different from the one on which they were evolved; the same can also be observed for constructions using bent seeds. All the other configurations (i.e., with two independent variables ($n + 2$), using balanced seeds) could produce "general" constructions in at least several runs.

The results for those constructions are presented in Table 4; the first row lists the target number of variables ($n + 2$), the middle row shows the nonlinearity of seed functions in n variables, and the last row shows the obtained nonlinearity. The most important finding in this phase is that, although the evolved constructions look different (with a different genotype) and produce different resulting Boolean functions, all the constructions we tested always produce the same nonlinearity for a given number of variables. We try to analyze this behavior in the next section.

From the results, it is evident that the obtained constructions are "general", in the sense that they always produce balanced Boolean functions with very high nonlinearity. In some cases, that nonlinearity is optimal (which is known for $n < 8$) and in some cases equal to the one obtained by search-based GP. It is interesting to note that, in cases where constructions produce lower nonlinearity than simple GP, the GP has a significantly lower probability of finding better solutions.

Table 3: Results for search-based GP in finding balanced highly nonlinear Boolean functions

GP search	n = 9	n = 10	n = 11	n = 12	n = 13	n = 14	n = 15	n = 16	n = 17	n = 18
min NL	240	480	992	1 953	3 905	8 001	16 192	32 512	65 280	130 561
avg NL	240	484.24	992	1 996.69	4 028.39	8 087.3	16 253.5	32 542.2	65 280	130 622
max NL	240	492	992	2 008	4 032	8 120	16 256	32 608	65 280	130 753
occurrence of max	100%	2%	100%	14%	96%	1%	97%	2%	100%	2%

Table 4: Results for secondary constructions producing balanced highly nonlinear Boolean functions

constructions	n = 7	n = 8	n = 9	n = 10	n = 11	n = 12	n = 13	n = 14	n = 15	n = 16	n = 17	n = 18
seed NL	12	26	56	116	240	488	992	2 000	4 032	8 096	16 256	32 576
resulting NL	56	116	240	488	992	2 000	4 032	8 096	16 256	32 576	65 280	130 688

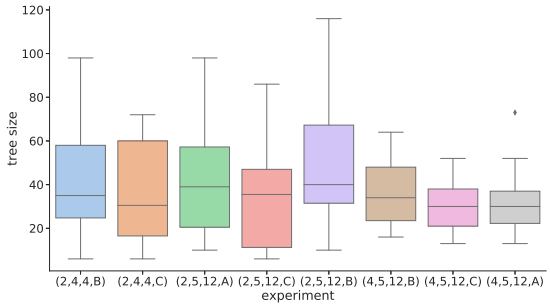


Figure 2: Tree size distribution for the eight experiments that always converged to a general solution.

6 DISCUSSION

We now investigate in detail the constructions produced by GP for those experiments that always converged to a general solution in each run. In particular, we considered five experiments with two seed functions and three experiments with four seed functions. In all considered experiments, the constructions extended the seed functions by two additional variables. In what follows, an experiment is synthetically identified by the tuple (s, n, nl, ev), where s is the number of seed functions used as input for the constructions, n is the number of variables of each seed function (which means that the construction will generate functions of $n + 2$ variables), nl is the nonlinearity of the seed functions, and ev is the evaluation method used by GP to evolve the constructions. In particular, we denote by A, B, C respectively the three fitness functions fit_1, fit_2 and fit_3 defined in Section 4.3.

6.1 Solutions Size

We start by analyzing the size of the trees generated by GP, considering it as an interpretability proxy. In particular, we define the size of a tree as the number of its nodes, independently of the type of functional or input variable (i.e., additional independent variable or seed function). Figure 2 plots the distributions of the tree sizes for all eight experiments.

The first interesting remark is that the distributions of the five experiments with two seed functions are quite dispersed, with extreme values ranging as low as 6 nodes and as high as 120 nodes. Further, the upper quartiles of these distributions make up a large part of the interquartile ranges, except for the experiment $(2, 5, 12, C)$. Considering also that the median tree size is around 35-40 nodes, we can conclude that most of the trees evolved by GP with two seed functions are too unwieldy to be interpreted by hand. The situation seems better with the three experiments using four seed functions, where the upper and lower quartiles are more balanced, and there is also a smaller difference between the minimum and maximum values. However, the median tree sizes are similar to those of the experiments with two seed functions, and the minimum sizes are significantly higher. Therefore, we end up with constructions that are quite difficult to interpret also with four seed functions.

6.2 Solutions Diversity

As a next step, we employed the ESPRESSO heuristic logic minimizer [24] to simplify the GP trees obtained in all eight experiments, with a twofold objective. First, we determined the simplest possible circuit of each construction by performing an exact minimization and verified if its size was small enough to elicit a manual interpretation. Unfortunately, the resulting minimized expressions were still too complex for a deeper analysis. As a second objective, we checked for *equivalent* circuits to investigate how many different solutions GP can generate within each experiment. In particular, the ESPRESSO tool allows checking if two different circuits are equivalent by comparing their truth tables and applying basic equivalence relations such as output negation or permutation of the input variables. We performed a pairwise equivalence test among all solutions evolved by GP in each considered experiment and built the corresponding graphs. Hence, each graph is composed of 30 nodes (one node per solution), and two nodes are connected by an undirected edge if and only if the two circuits were marked as equivalent by the ESPRESSO minimizer.

Figure 3 displays the adjacency matrices of the equivalence relation graphs, while Table 5 reports for each experiment the number of distinct solutions (or equivalence classes), the size of the largest equivalence class, and the number of seeds effectively used.

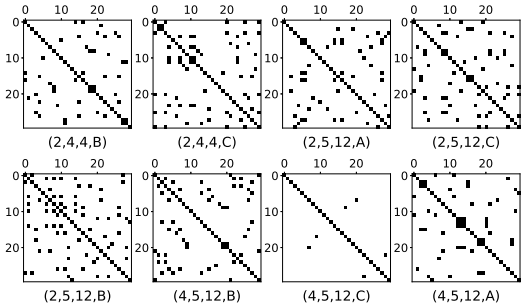


Figure 3: Adjacency matrices for the equivalence relation graphs of all eight experiments.

Table 5: Equivalence classes summary for all experiments. Bold values represent experiments where all constructions use *less* seeds than allowed.

Exp.	#classes	max_size	seeds_used
(2,4,4,B)	15	4	2
(2,4,4,C)	13	5	2
(2,5,12,A)	15	5	2
(2,5,12,C)	13	5	2
(2,5,12,B)	11	5	2
(4,5,12,B)	14	4	**2**
(4,5,12,C)	27	2	4
(4,5,12,A)	14	4	**2**

In general, one can see from Figure 3 and Table 5 that using two seed functions generally leads GP to evolve many equivalent functions, with $(2, 5, 12, B)$ yielding the smallest number of distinct solutions and the largest equivalence classes. This is somewhat expected since the number of ways to combine the two additional variables of the constructions is smaller with two seed functions than with four. Accordingly, the experiment $(4, 5, 12, C)$ is the one giving the highest diversity among solutions, with only 6 equivalent solutions grouped in three equivalences classes of size 2.

Contrarily, the distributions of equivalence classes for the other two experiments with four seed functions, namely $(4, 5, 12, B)$ and $(4, 5, 12, A)$, are closer to those with two seeds. What is more surprising is that all solutions evolved by GP in these two experiments actually use *less* seeds than expected, as shown by the entries in bold of Table 5. Although the original GP trees of each solution in these two experiments use all seeds functions, the ESPRESSO tool always returned minimized circuits where the seeds f_2 and f_3 are never used. This is interesting since, as explained in Section 4.3, we adopted a penalty factor in all our fitness functions in order to force the occurrence of all seed functions in the candidate trees. However, in this particular experiment GP was able to circumvent this penalty by using all four seed terminals at a syntactic level, but encoding two of them in subtrees that do not affect the output of the constructions. The cause of this phenomenon likely resides in the underlying fitness functions, which are fit_2 and fit_1 respectively for the experiments $(4, 5, 12, B)$ and $(4, 5, 12, A)$. In both cases, each seed group can contribute in a non-uniform way to the fitness value of an individual. This might, in turn, lead the GP evolutionary process to favor general constructions with fewer active seed functions. On the other hand, the experiment $(4, 5, 12, C)$, where all four seed functions partake in the minimized circuits, is based on the fitness function fit_3, which maximizes the minimum objective value among all seed groups. In this case, GP is forced to evolve constructions that yield highly nonlinear balanced functions for each tested group, possibly increasing the chances that all seed functions are combined uniformly.

Table 6: Simplest GP constructions selected for analysis.

exp.	size	construction
(2,4,4,B)	6	IF(v_0, f_0, (v_1 XOR f_1))
(2,4,4,C)	6	IF(v_0, f_0, (f_1 XOR v_1))
(2,5,12,A)	10	IF(v_0, f_1, ((v_1 XOR f_0) OR (v_1 AND v_0)))
(2,5,12,C)	6	IF(v_1, f_1, (f_0 XOR v_0))
(2,5,12,B)	10	IF(NOT(NOT(v_0)), NOT((f_0 XOR NOT(v_1))), f_1)
(4,5,12,B)	17	IF(v_1, (v_0 XOR (f_1 AND v_1)), IF(v_1, (f_2 OR (f_2 AND (f_2 OR f_3))), f_0))
(4,5,12,C)	17	IF(v_0, (f_1 XOR v_1), (((f_0 OR f_3) AND2 IF(f_3, f_2, v_1)) AND v_0) OR f_3))
(4,5,12,A)	22	IF(v_0, (v_0 AND2 f_1), ((v_0 AND ((f_2 XOR v_1) XOR f_3)) XOR (NOT(f_0) XOR IF((v_0 XNOR v_1), v_1, v_0))))

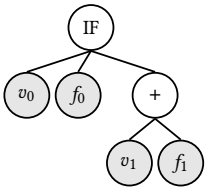
6.3 Interpreting Simple Constructions

So far, we discussed the general constructions concerning their sizes and diversity, which gave us some insights on the GP's behavior for this particular optimization problem. We now investigate the specific nature of these constructions to determine if they are new or already known in the literature of Boolean functions.

As remarked in Section 6.2, the minimized circuits obtained through ESPRESSO are, on average, still too complex to allow a manual interpretation. For this reason, here we analyze in detail only some of the simplest constructions evolved by GP. In particular, we selected one construction for each of the eight experiments investigated in the previous sections. Our selection criteria for "simplicity" were as follows:

(1) Small tree size, by considering the lower quartiles of the tree size distributions in Figure 2 as an upper bound.
(2) IF node at the root, so the construction is piecewise-defined.
(3) Condition at the root IF composed of a single literal (independent additional variable or seed function). This helps avoiding bloated expressions that control the output of the functions resulting from the construction.

Table 6 reports the selected constructions for each experiment, as evolved by GP. The notation used for the expressions includes v_0, v_1 for the two additional variables and $f_0 - f_3$ for the seed functions. Regarding the constructions with two seed functions, one can easily see that the smallest solutions of size 6 all correspond to the same construction, up to a swap of the XOR operands or a renaming of the leaf nodes. Figure 4 depicts the tree of this construction and the



$$F(v_0, v_1, v) = \begin{cases} f_0(v) \ , & \text{if } v_0 = 1 \ , \\ f_1(v) \oplus v_1 \ , & \text{if } v_0 = 0 \ . \end{cases} \tag{11}$$

Figure 4: Smallest GP construction. XOR is denoted by +.

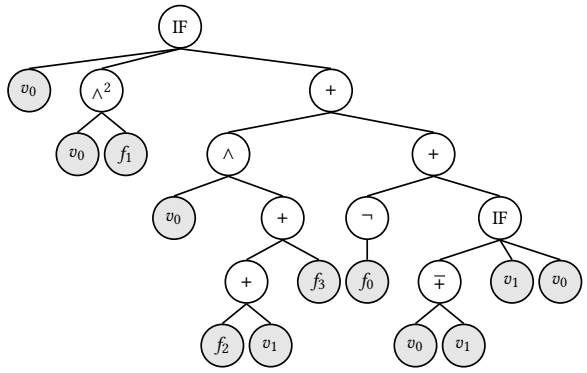


Figure 5: Example of bloated GP construction. AND2 is denoted by $\wedge^2$, XNOR by $\overline{+}$.

corresponding mathematical definition, taking as a reference the solution of the experiment $(2, 4, 4, B)$. Considering the truth table representation, this construction basically concatenates the two seed functions as $f_0||f_0||\overline{f_1}||f_1$, where $\overline{f_1}$ denotes the negation of f_1. For this reason, in what follows we refer to this expression as the concatenation construction.

It is interesting to remark that *the remaining five expressions in Table 6 still correspond to the concatenation construction.* This can be easily verified for the expressions with two seed functions. In particular, for the experiment $(2, 5, 12, A)$ the innermost AND always maps to 0, since we are in the subtree where $v_0 = 0$, and thus the subsequent OR evaluates to v_1 XOR f_0. Similarly, for the expression selected in experiment $(2, 5, 12, B)$, the condition on the root IF is simply a double negation of v_0. Further, the subtree NOT(f_0 XOR NOT(v_1)) which is selected when $v_0 = 1$ is equivalent to f_0 XOR v_1, by comparing the respective truth tables.

Concerning the three constructions with four seed functions, the simplification process is more convoluted, so we do not report it here in full for the sake of brevity. As an example, we only show in detail the largest tree in Figure 5, namely the construction selected for $(4, 5, 12, A)$. The subtree with AND2 always evaluates to NOT(f_1), since $v_0 = 1$ in that branch of the root IF. On the contrary, the AND subtree can be pruned since $v_0 = 0$, and thus it always evaluates to 0. Hence, the seed functions f_2 and f_3 are effectively discarded, since they only occur in this prunable subtree. Replacing the other occurrences of v_0 with 0 in the remaining XOR subtree, one finally gets NOT(f_1) XOR v_1, which is a trivial variation of $f_1(v) \oplus v_1$ when $v_0 = 0$. Therefore, this tree is equivalent to the concatenation construction as well.

Since all solutions analyzed up to now are equivalent, it makes sense to determine whether the concatenation construction corresponds to a known result in the related literature. In particular, taking the expression in Figure 4 and exchanging the indices of v_0 and v_1, one can see that this construction is a particular case of the *indirect sum construction* [4], where only two additional variables are used to extend the functions.

7 CONCLUSIONS AND FUTURE WORK

This paper proposed for the first time a GP approach to evolve secondary constructions of Boolean functions that are both balanced and highly nonlinear, which are particularly relevant in the design of symmetric ciphers. A candidate construction is encoded by a tree where the internal nodes are Boolean operators, while the leaves represent either seed functions or additional independent variables. The fitness functions evaluate the generality of construction by measuring the balancedness and the nonlinearity of the resulting Boolean functions starting from a set of optimal seeds. Our experiments show that, for certain parameters combinations, GP always converges to a general construction. A closer inspection of these solutions reveals that GP actually finds many equivalent constructions, and the solutions that we analyzed in detail turned out to be a particular case of the indirect sum construction [4].

Our findings seem to indicate that GP cannot find novel constructions with our current formulation of the optimization problem. However, it is still remarkable that GP always finds the same simple construction in all considered experiments, albeit under different syntactic forms. One possible explanation for this behavior could be related to the genotype representation adopted in our experiments. Indeed, the additional two variables are always used *externally to the seed functions*; in other words, v_0 and v_1 are never employed as inputs to the seed functions themselves, but rather their values are combined with the outputs of the seeds. Remark this is not a real restriction from the semantic point of view, since all Boolean functions of $n + 2$ variables can be expressed as the combination of two n-variable functions f, g with the additional two variables; However, considering also that we enforce a maximum depth on the trees, the representations that GP can evolve in this way are quite constrained. In particular, we formulate the hypothesis that the concatenation construction is the *only* general construction discoverable by GP under this encoding and that differences arise only at a syntactic level, with more or less bloated constructions. We plan to investigate this hypothesis in future research, following two complementary future directions. The first direction is to investigate if, under the given encoding constraints, the concatenation is the only optimal solution in the *semantic space* of constructions. This could be accomplished by analyzing the space of all constructions in terms of their truth tables. It would also be interesting to consider the use of geometric semantic GP [18] for this particular problem. Finally, the second direction is to experiment with GP encodings that are less constrained, by either allowing the additional variables to partake in the input of the seed functions, or by adopting a more general approach with independent additional variables. Indeed, the indirect sum construction is more symmetric in its structure than what we experimented with in this paper, and this the reason why GP could not evolve it in its most general form.

REFERENCES

[1] Hernan Aguirre, Hiroyuki Okazaki, and Yasushi Fuwa. 2007. An Evolutionary Multiobjective Approach to Design Highly Non-linear Boolean Functions. In *Genetic and Evolutionary Computation Conference (GECCO)*. 749–756.

[2] Ruibin Bai, Edmund K. Burke, Graham Kendall, Jingpeng Li, and Barry McCollum. 2010. A Hybrid Evolutionary Approach to the Nurse Rostering Problem. *IEEE Transactions on Evolutionary Computation* 14, 4 (2010), 580–590. https://doi.org/10.1109/TEVC.2009.2033583

[3] Georg T. Becker. 2015. The Gap Between Promise and Reality: On the Insecurity of XOR Arbiter PUFs. In *Cryptographic Hardware and Embedded Systems – CHES 2015*, Tim Güneysu and Helena Handschuh (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 535–555.

[4] Claude Carlet. 2021. *Boolean Functions for Cryptography and Coding Theory*. Cambridge University Press. https://doi.org/10.1017/9781108606806

[5] Claude Carlet, Domagoj Jakobovic, and Stjepan Picek. 2021. *Evolutionary Algorithms-Assisted Construction of Cryptographic Boolean Functions*. Association for Computing Machinery, New York, NY, USA, 565–573. https://doi.org/10.1145/3449639.3459362

[6] Claude Carlet and Sihem Mesnager. 2016. Four Decades of Research on Bent Functions. *Des. Codes Cryptography* 78, 1 (Jan. 2016), 5–50.

[7] John A Clark and Jeremy L Jacob. 2000. Two-Stage Optimisation in the Design of Boolean Functions. In *Information Security and Privacy*. LNCS, Vol. 1841. Springer, 242–254.

[8] John A. Clark, Jeremy L. Jacob, Subhamoy Maitra, and Pantelimon Stanica. 2004. Almost Boolean Functions: The Design of Boolean Functions by Spectral Inversion. *Comput. Intell.* 20, 3 (2004), 450–462.

[9] J. Kennedy and R. Eberhart. 1995. Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, Vol. 4. 1942–1948 vol.4. https://doi.org/10.1109/ICNN.1995.488968

[10] Zhichao Lu, Ian Whalen, Yashesh Dhebar, Kalyanmoy Deb, Erik D. Goodman, Wolfgang Banzhaf, and Vishnu Naresh Boddeti. 2021. Multiobjective Evolutionary Design of Deep Convolutional Neural Networks for Image Classification. *IEEE Transactions on Evolutionary Computation* 25, 2 (2021), 277–291. https://doi.org/10.1109/TEVC.2020.3024708

[11] Luca Mariot and Alberto Leporati. 2015. 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 (Lecture Notes in Computer Science, Vol. 9477)*, Adrian-Horia Dediu, Luis Magdalena, and Carlos Martín-Vide (Eds.). Springer, 33–45.

[12] Luca Mariot and Alberto Leporati. 2015. Heuristic Search by Particle Swarm Optimization of Boolean Functions for Cryptographic Applications. In *Genetic and Evolutionary Computation Conference, GECCO, Companion Material Proceedings*. 1425–1426.

[13] Luca Mariot, Stjepan Picek, Alberto Leporati, and Domagoj Jakobovic. 2019. Cellular automata based S-boxes. *Cryptography and Communications* 11, 1 (2019), 41–62. https://doi.org/10.1007/s12095-018-0311-8

[14] Luca Mariot, Martina Saletta, Alberto Leporati, and Luca Manzoni. 2021. Heuristic Search of (Semi-)Bent Functions based on Cellular Automata. *CoRR* abs/2111.13248 (2021).

[15] William Millan, Andrew Clark, and Ed Dawson. 1997. An Effective Genetic Algorithm for Finding Highly Nonlinear Boolean Functions. In *First Int. Conference on Information and Communication Security (ICICS '97)*. Springer, 149–158.

[16] William Millan, Andrew Clark, and Ed Dawson. 1998. Heuristic Design of Cryptographically Strong Balanced Boolean Functions. In *Advances in Cryptology - EUROCRYPT '98*. 489–499.

[17] Julian F. Miller. 1999. An Empirical Study of the Efficiency of Learning Boolean Functions Using a Cartesian Genetic Programming Approach. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 2* (Orlando, Florida) *(GECCO'99)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1135–1142.

[18] Alberto Moraglio, Krzysztof Krawiec, and Colin G. Johnson. 2012. Geometric Semantic Genetic Programming. In *Parallel Problem Solving from Nature - PPSN XII - 12th International Conference, Taormina, Italy, September 1-5, 2012, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 7491)*, Carlos A. Coello Coello, Vincenzo Cutello, Kalyanmoy Deb, Stephanie Forrest, Giuseppe Nicosia, and Mario Pavone (Eds.). Springer, 21–31.

[19] Stjepan Picek and Domagoj Jakobovic. 2016. 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*, Tobias Friedrich, Frank Neumann, and Andrew M. Sutton (Eds.). ACM, 781–788.

[20] Stjepan Picek and Domagoj Jakobovic. 2020. Evolutionary Computation and Machine Learning in Cryptology. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion* (Cancún, Mexico) *(GECCO '20)*. Association for Computing Machinery, New York, NY, USA, 1147–1173. https://doi.org/10.1145/3377929.3389886

[21] Stjepan Picek, Domagoj Jakobovic, and Marin Golub. 2013. Evolving Cryptographically Sound Boolean Functions. In *Genetic and Evolutionary Computation Conference (GECCO) (GECCO '13 Companion)*. ACM, 191–192.

[22] Stjepan Picek, Dominik Sisejkovic, and Domagoj Jakobovic. 2017. Immunological algorithms paradigm for construction of Boolean functions with good cryptographic properties. *Eng. Appl. of AI* 62 (2017), 320–330.

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

[24] Richard L. Rudell and Alberto L. Sangiovanni-Vincentelli. 1987. Multiple-Valued Minimization for PLA Optimization. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 6, 5 (1987), 727–750.

[25] Sayandeep Saha, Rajat Subhra Chakraborty, Srinivasa Shashank Nuthakki, Anshul, and Debdeep Mukhopadhyay. 2015. Improved Test Pattern Generation for Hardware Trojan Detection Using Genetic Algorithm and Boolean Satisfiability. In *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*. 577–596. https://doi.org/10.1007/978-3-662-48324-4_29

[26] Zhenbin Zhang, Liji Wu, An Wang, Zhaoli Mu, and Xiangmin Zhang. 2015. A novel bit scalable leakage model based on genetic algorithm. *Security and Communication Networks* 8, 18 (2015), 3896–3905. https://doi.org/10.1002/sec.1308 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/sec.1308