Comparison of Genetic Programming Methods on Design of Cryptographic Boolean Functions

Jakub Husa(✉)

Faculty of Information Technology, IT4Innovations Centre of Excellence, Brno University of Technology, Brno, Czech Republic
ihusa@fit.vutbr.cz

Abstract. The ever-increasing need for information security requires a constant refinement of contemporary ciphers. One of these are stream ciphers which secure data by utilizing a pseudo-randomly generated binary sequence. Generating a cryptographically secure sequence is not an easy task and requires a Boolean function possessing multiple cryptographic properties. One of the most successful ways of designing these functions is genetic programming. In this paper, we present a comparative study of three genetic programming methods, tree-based, Cartesian and linear, on the task of generating Boolean functions with an even number of inputs possessing good values of nonlinearity, balancedness, correlation immunity, and algebraic degree. Our results provide a comprehensive overview of how genetic programming methods compare when designing functions of different sizes, and we show that linear genetic programming, which has not been used for design of some of these functions before, is the best at dealing with increasing number of inputs, and creates desired functions with better reliability than the commonly used methods.

Keywords: Genetic programming · Cartesian Genetic programming · Linear Genetic programming · Cryptographic Boolean functions · Comparative study

1 Introduction

In 1882 Frank Miller, and later Gilbert Vernam, came up with the concept of a one-time pad and created a cipher which could under the right conditions be entirely unbreakable [1]. However for the cipher to work, it required the creation of pads of numbers which would be unique, truly random and could never be reused. These strict conditions made the cipher unfeasible for use in everyday life. However, its concept had survived and given birth to a family of stream ciphers.

These ciphers replace the one-time pads, with a single generator able to create a near infinite sequence so long that none of its parts would ever need

L. Sekanina et al. (Eds.): EuroGP 2019, LNCS 11451, pp. 228–244, 2019.
https://doi.org/10.1007/978-3-030-16670-0_15

to be reused. However, a deterministically generated sequence cannot be truly random. Instead, stream ciphers focus on making the sequence so complex that it can not be crypto-analyzed in feasible time. This requires a use of well designed specialized Boolean functions with cryptographic properties.

There are three main ways how to these functions can be created, random search, algebraic constructions, and heuristic methods (and their combinations) [2]. One of the most efficient heuristic methods are Evolutionary Algorithms. Inspired by the natural evolutionary process, these maintain a population of individuals each representing a potential solution. The individuals are then combined and mutated in a cycle guided by a fitness function, which determines the quality of each solution and steers the evolutionary process towards its goal.

In this paper, we focus on a specific subset of evolutionary algorithms called genetic programming (GP), which has been shown to provide great results in evolving Boolean functions of varying sizes and properties. The two most commonly used GP methods, tree-based, and Cartesian GP have already been the subject of multiple studies, while linear GP is usually overlooked. Another contribution of our paper is that we use two different population schemes for each GP method. To the best of our knowledge, this is a type of comparison none of the related works have performed before. Our aim is to provide a fair and comprehensive comparison of the individual GP methods and gain insight into what approach is most suited for each of the many various tasks.

The rest of this paper is organized as follows. Section 2 describes preliminaries of what Boolean functions are, how are they used in stream ciphers, and what are their cryptographic properties. In Sect. 3 we describe the various GP methods and go over related works that used GP to create Boolean functions in the past. In Sect. 4 we define our objectives, what fitness functions we use, how our experiments are set up, and how have the parameters used by each GP method been optimized. Section 5 shows the results of our experiments and highlights the most interesting findings. The work concludes with Sect. 6 which provides a summary and outlines the possible future works.

2 Preliminaries

Boolean function is a function $B^n \rightarrow B$ where $B \in \{0, 1\}$ and $n \in \mathbb{N}$. In other words, it is a function which takes multiple binary inputs and provides a single binary output. The simplest way of representing a Boolean function is with a *truth table*, which assigns a specific output to every possible combination of inputs via a binary vector of length n^2, with n being the number of inputs [3].

Another way a binary function can be represented is the *Walsh spectrum*. It is defined as:

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

And shows the correlation (Hamming distance) between the examined function $f(\boldsymbol{x})$ and all linear functions $(\boldsymbol{a}.\boldsymbol{x})$ [4]. This representation is useful for determining the function's nonlinearity (defined below). It can be calculated

from the truth table representation using Fast Walsh–Hadamard transform in $O(n2^n)$, with n being the number of inputs [5].

The third way of representing Boolean functions is with the *algebraic normal form*, which uses a multivariate polynomial P defined as [3]:

$$P(x) = \oplus_{w \in \mathbb{F}_2^n} h(w).x^w \tag{2}$$

with $h(w)$ defined by the Möbius inversion principle [6]:

$$h(w) = \oplus_{x \preceq w} f(x), \text{ for any } w \in \mathbb{F}_2^n \tag{3}$$

In other words, algebraic normal form represents functions as a logical XOR of terms, which are themselves comprised of logical AND of the function's inputs. This representation allows to determine the function's algebraic degree (defined below), and for n inputs it can be calculated from the truth table representation in $O(n2^n)$.

2.1 Use of Boolean Functions

Boolean functions are used in stream ciphers when generating the pseudo-random sequence. The most commonly used type of generators use a Linear Feedback Shift Register (LFSR). It is comprised of a binary register of length m, initialized by a secret key and an initialization vector to some non-zero value. When generating a sequence, values in the register are shifted by one bit to the right. The right-most bit leaving the register is used as its output, and a new left-most bit is calculated by a (linear) generating polynomial from its current state. For a register of any common length, there is a well-known list of *primitive polynomials*, which will ensure that the generator will pass through every possible configuration except the state of all zeros before repeating itself, and thus generate a sequence with a period of $2^m - 1$. This means that for sufficiently long registers the sequence is effectively non-repeating [7].

LFSR generators are fast and easy to implement in both software and hardware and provide an output with good statistical properties [8,9]. However, its linear nature makes it susceptible to many cryptographic attacks. Given $2m$ bits of output, the initial setting of the register (and thus the secret key) can be reconstructed in $O(m^2)$ using the Berlekamp-Massey algorithm [10,11].

To ensure security the relationship between LFSR's internal state and its output needs to be obscured with a cryptographically sound Boolean function. Figure 1 shows the two main ways in which it can be applied, which determines what properties the function needs to possess.

The *combiner* model utilizes outputs of several short LFSRs of co-prime length and requires a Boolean function with a high degree of correlation immunity (defined below), equal to the number of LFSRs used [8]. The *filter* model utilizes several bits of a single long LFSR. This means that the Boolean function only requires a correlation immunity of 1, allowing for higher values of other cryptographic properties. However, the placement of input bits (tap positions) may open the generator to other types of cryptographic attacks, which are yet to be fully explored [12,13].

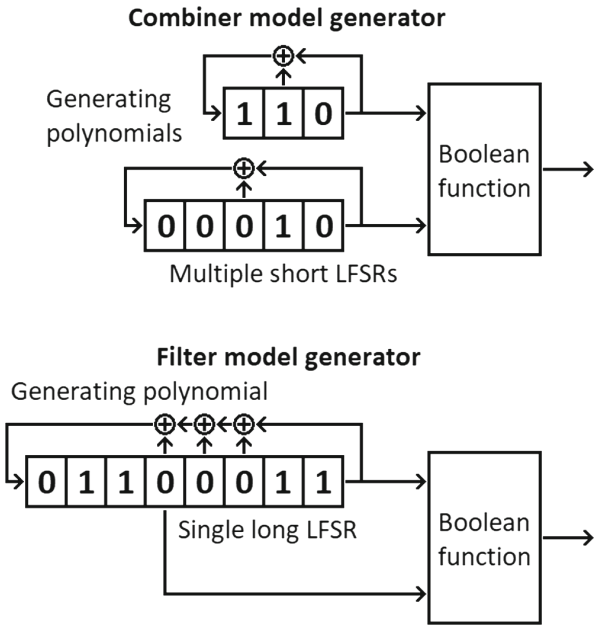

Fig. 1. Use of Boolean functions in combiner and filter model of LFSR generators.

2.2 Property Definitions

In this paper, we focus on Boolean functions suitable for LFSR generators and possessing four cryptographic properties. Balancedness, nonlinearity, correlation immunity, and algebraic degree. Each property determines how difficult it will be to break the cipher using a specific type of attack. The individual properties are mutually conflicting, and when designing Boolean functions one always looks for a compromise. For a function to be directly applicable in a cipher it should also possess a high degree of Algebraic and Fast-Algebraic immunity, and have at least 13 inputs [14], though the ideal number of inputs is considered to be at least 20 [8]. But functions with fewer inputs and possessing only some of the cryptographic properties are still important, as they are used as building blocks for larger more secure functions constructed by analytical approaches [15].

A Boolean function is balanced if its truth table contains an equal number of ones and zeros, making its output statistically indistinguishable from a random sequence. An unbalanced function would cause a statistical relationship between plaintext and ciphertext of the secured messages, and make them vulnerable to attack by frequency analysis [8,16].

Linear Boolean function is a function that can be created by logical XOR of its inputs (including a constant 0 function). An affine function is either a linear function or its complement. Nonlinearity N_f of a function f is the minimal Hamming distance between the function's truth table and the truth table of any of the affine functions. For functions with an even number of inputs the

maximum nonlinearity is given by equation:

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

Functions that reach this value are called *bent* functions. Nonlinearity of functions with an odd number of inputs has a known upper bound, but to the best of our knowledge, this limit has only been reached for functions with seven or fewer inputs [17]. Nonlinearity obscures the linear relationship between the state of LSFR and its output and protects the cipher against the Fast Correlation attack [16,18,19].

Function f has a correlation immunity CI_f of degree t if its output is statistically independent of any t inputs. In other words, if the function's truth table was split in half based on whether one of its inputs is 0 or 1, and both of the sub-tables would contain the same number of 1s, regardless of which input was used to create this split, then the function has a correlation immunity of the first degree. If the function's truth table could be quartered using a combination of any two inputs, it would have correlation immunity of the second degree, and so on. Correlation immunity protects against the Siegenthaler's correlation attack [20] and is in direct conflict with the function's algebraic degree Deg_f (defined below). If $t \in <2, n-1>$ then $Deg_f \leq n - CI_f - 1$, otherwise $Deg_f \leq n - CI_f$, where n is the number of function inputs. This limitation is known as Siegenthaler's Inequality [19]. If a function is both immune to correlation and balanced it is called *resilient*.

Algebraic degree Deg_f of function f is defined as the maximum number of elements in a single term when it's represented in the algebraic normal form (defined above). High algebraic degree protects the functions against the Berlekamp-Massey algorithm [10,11], Rønjom-Helleseth attack [21], and other algebraic attacks [22].

3 Genetic Programming Methods

Genetic programming is a subset of evolutionary algorithms focused on the evolution of executable structures. For Boolean functions, this usually means a logic equation used to calculate its truth table. This allows functions with many inputs to be stored in a compact form, and to be evolved with greater efficiency.

The most common form of GP is the Tree-based genetic programming (TGP). It represents its genotype using a syntactic tree consisting of two types of nodes. Internal (function) nodes that represent logical operators, and take their inputs from other function nodes or leaves. Leaf (terminal) nodes represent either the function's input variables or constants. TGP uses genotypes of variable length, but the maximum allowed depth of its tree may be restricted to prevent it from bloating to an unmanageable size [23].

Cartesian Genetic programming (CGP), represents the chromosome using a two-dimensional array of acyclically interconnected nodes. Each node contains a logical operator and several inputs that can be connected either to the function's input variables or a node in one of the previous columns. Because every node

can be utilized in multiple data paths, performing genetic crossover is difficult, and offsprings are usually created utilizing only the mutation operator. In software implementations, the grid is often implemented as a single row, with no restrictions on how many levels-back can a node's input reach. The output is usually taken from a node specified by the last gene(s) of the genotype [24].

Linear Genetic programming (LGP) represents the chromosome as a linearly-executed sequence of instruction operating over a finite set of registers. Each instruction specifies a logical operator, several operands chosen from the set of registers and function inputs, and an output selected from the set of registers. The numbers of instructions and registers are mutually independent, and the values left in the registers after all instructions have been executed serve as the function output [25].

3.1 Related Works

The earliest application of evolutionary algorithms for the design of cryptographic Boolean functions focused on finding functions with high nonlinearity with a genetic algorithm [26]. TGP has first been used to design cryptographically sound Boolean functions with 8 inputs [27], while CGP was used to search for bent Boolean functions with up 16 and later 18 inputs [28,29]. LGP was used to design bent functions with up to 24 inputs and has been shown to cope with increasing number of variables better than CGP [30].

There have been multiple studies comparing genetic algorithms, TGP, and CGP on the design of cryptographic Boolean functions applicable in stream ciphers, and functions with correlation immunity and minimal hamming weight useful in preventing side-channel attacks. All of these studies have shown that GP methods provide better results than the other evolutionary approaches [14,31–34].

In other works, TGP has been used to design algebraic constructions for combining existing bent Boolean functions into larger bent functions with up to 20 inputs [35], and a comparative study experimenting with designs of bent and resilient functions using CGP has shown that the latter may benefit from the use of various crossover operators [36].

4 Objectives

Our goal is to provide a comprehensive comparison of the three GP methods. For this reason, we choose several tasks of varying difficulty. To see how each method copes with growing search space we use Boolean functions of 6, 8, 10, and 12 inputs. To see how they handle increasingly restrictive criteria, we design four types of functions. Bent Boolean functions maximizing the nonlinearity property. Balanced functions with high nonlinearity. Resilient functions with correlation immunity of the first degree. And finally, resilient functions with high algebraic degree approaching the Siegenthaler's inequality (Siegenthaler functions).

For each type of a Boolean function, we define a fitness function that we try to maximize. Because the performance of evolutionary algorithms depends on how well is the fitness function able to guide its search, we use not only the raw values of cryptographic properties but define coefficients that show how close a solution is to meeting the specified criteria.

For balancedness we define coefficient:

$$BAL = 1 - \frac{|ONES - ZEROS|}{2^n} \tag{5}$$

Where ONES and ZEROS are the number of 1s and 0s in the function's truth table, and n is the number of function inputs. For Correlation immunity we define coefficient:

$$CRI = \frac{SPLIT}{n} \tag{6}$$

Where SPLIT is the number of inputs that can split the Boolean function's truth table and create two halves with an equivalent number of 1s. For algebraic degree we define coefficient:

$$DEG = max(1, \frac{Deg_f}{n-2}) \tag{7}$$

The value $n-2$ is one degree less than the limit determined by Siegenthaler's inequality to allow the evolved Boolean functions to also be balanced and highly nonlinear. Lastly, we define an acceptability coefficient:

$$ACC = \begin{cases} 1 & \text{if the function meets all criteria} \\ \frac{1}{2^{n-1}} & \text{otherwise} \end{cases} \tag{8}$$

This coefficient is used to reduce the fitness of Boolean functions that do not represent acceptable solutions to be within the range $<0, 1>$, which is worse than the fitness of any acceptable solution.

Using the coefficients and the raw value of nonlinearity, we define the fitness function for each of the four types of functions being evolved:

$$F_{Bent} = N_f * ACC \tag{9}$$

$$F_{Balanced} = N_f * ACC * BAL \tag{10}$$

$$F_{Resilient} = N_f * ACC * BAL * CRI \tag{11}$$

$$F_{Siegenthaler} = N_f * ACC * BAL * CRI * DEG \tag{12}$$

Another possible way of combining the multiple criteria would be to utilize a multi-objective algorithm. However, related works suggest that this approach is not competitive when designing Boolean functions with cryptographic properties [33,34], and so we leave this option out of the scope of this paper.

4.1 Experimental Setup

The various GP methods usually manage their population in different ways. To make our comparison fair, we perform all experiments using two different population schemes.

First scheme is the Steady-state Tournament (SST). Its initial population is generated randomly, and new individuals are created by randomly selecting thee individuals, and replacing the worst of them with a child of the better two. The child is created by a crossover followed by mutation. TGP uses standard subtree crossover that replaces a randomly selected function node (and its subtree) from one parent with a randomly selected function node (and its subtree) from the other parent. CGP and LGP use standard one-point crossover set to only split the chromosomes between whole nodes, respectively instructions.

Second scheme is the $(1 + \lambda)$ Evolution strategy (EST). Its initial population is randomly generated and new individuals are created in generations, by selecting the currently best individual and creating λ offsprings via mutation. If the parent and any of its offspring have the same fitness, the offsprings are preferred when choosing a new parent of the next generation, to promote fitness-neutral mutations. TGP mutates individuals by replacing randomly selected function node with a new, randomly generated subtree. CGP and LGP perform mutation by changing any of its individual genes to a randomized value with a small probability.

To increase the overall informative value of our comparison, we include a fourth evolutionary method. A genetic algorithm (GAL), whose chromosome represents the Boolean function's truth table directly, and not as an equation. It uses a one-point crossover and mutates individuals by flipping any number of individual bits in its chromosome to their opposite value, with a small probability.

Table 1. Fitness values desired for each type of function and number of inputs.

Inputs	Bent	Balanced	Resilient	Siegenthaler
6	28	24	24	24
8	120	112	112	112
10	496	480	480	480
12	2016	1984	1984	1984

For each evolved type of function, we choose a desired nonlinearity (and a corresponding fitness). For bent functions, we use the optimal value given by Eq. 4. For the rest of the functions we determine the desired nonlinearity experimentally by running each algorithm for 1 000 000 evaluations, and selecting the best value reached to be our goal, as shown in Table 1. The selected values are lower than some of the known lower bounds on maximum nonlinearity [37].

However, while we believe that these values can be reached using GP as well, the number of evaluations required would make our type comparison unfeasible.

All GP methods use {AND, OR, XOR, XNOR} as their set of operators, and had two logical constant {TRUE, FALSE} added to their Boolean function's inputs. The project is implemented in C++ using the Evolutionary Computation Framework[1], parallelized with Message Passing Interface[2]. During parallel execution, the main core maintains the population and performs the quick and easy tasks of selection, crossover, and mutation, while the computationally expensive task of obtaining each individual's truth table, cryptographic properties, and the resulting fitness, is passed to a number of worker cores.

4.2 Parameter Optimization

To determine each method's best performing setup, we use a one-at-a-time optimization method. GP methods were optimized on the task of evolving bent functions with 12 inputs. Because the size of a function's truth table is determined by the number of its inputs the chromosome length of GAL could not be optimized, and is included only for comparison. Because the ideal mutation rate is highly dependent on the chromosome length, it was optimized for each number of inputs separately. The examined ranges and the best values found are shown in Table 2.

For SST scheme, a medium sized population of around 15–25 individuals performed the best, with the exception of TGP method which performed even better

Table 2. Parameters optimized for each genetic programming method, the genetic algorithm with different number of inputs, and for both population schemes.

Optimized property	Examined range	TGP	CGP	LGP	GAL6	GAL8	GAL10	GAL12
$(1+\lambda)$ Evolution Strategy								
Population	$1+1$–1000	$1+5$	$1+5$	$1+5$	$1+5$	$1+5$	$1+5$	$1+5$
Mutation rate	0.00025–1.0	1.0	0.035	0.025	0.04	0.008	0.0025	0.001
Chrom. length	31–4095	–	511	511	(64)	(256)	(1024)	(4096)
Tree depth	5–12	9	–	–	–	–	–	–
Free registers	5–100	–	–	15	–	–	–	–
Steady-State Tournament								
Population	3–1000	40	20	20	20	20	20	20
Mutation rate	0.00025–1.0	1.0	0.02	0.0125	0.04	0.008	0.0025	0.001
Chrom. length	31–4095	–	511	511	(64)	(256)	(1024)	(4096)
Tree depth	5–12	9	–	–	–	–	–	–
Free registers	5–100	–	–	15	–	–	–	–

[1] http://ecf.zemris.fer.hr/.
[2] https://www.open-mpi.org/.

with a larger population of around 35–50 individuals, possibly due to utilizing a different crossover operator than the other methods. This result is somewhat surprising, as TGP commonly performs the best with a population containing hundreds or thousands of individuals [38]. For EST scheme, the smallest populations always performed the best, but we have chosen a population with 5 offsprings to allow for a reasonable degree of parallelization.

For TGP method the genotype size was set as the maximum depth of its tree. For CGP and LGP we have worked with the number of nodes (respectively instructions) and restricted our search to genotype lengths equivalent to a fully grown TGP tree ($2^n - 1$). For CGP we have used a single-row implementation with no limit on each node's reach. For all GP methods, the optimal result was equivalent to a maximum of 511 logical operators, implying that this value depends on the task itself, rather than the method being used.

For CGP and LGP, the ideal mutation rate depended on the population scheme and was higher for EST, which uses mutation as its only genetic operator. For TGP and GAL, the mutation rate was the same for both population schemes. For LGP we have also optimized the number of available registers and found the optimal value to be around 15, including the final register whose value is used as the function output.

5 Results

The experiments include a multitude of tasks. We use four Evolutionary algorithms, GAL, TGP, CGP, and LGP, two population schemes SST and EST, evolve four types of Boolean functions, bent, balanced, resilient, and Siegenthaler, and consider use 6, 8, 10, and 12 inputs for each. Combination of all these setups results in 128 individual tasks. For each of these, we have performed 100 independent runs. Each run is limited to a maximum of 1 000 000 evaluation, and if it does not find a solution before this limit, the run is considered unsuccessful.

Our basis of comparison for the results is the number of fitness function evaluations required to find a Boolean function with the desired fitness. The complete set of results is shown in Tables 3, 4, 5, and 6 (one for each type of Boolean function).

All four tables order the experiments based on the number of inputs, the GP method, and population scheme. We consider the median outcome to be the most telling indicator of success and the first and third quartile as secondary indicators. These values are shown in columns "Q1", "Median", and "Q3", rounded to the nearest integer value. The last column labeled "Suc." shows how many of the 100 runs have been successful.

Our results show that the GAL, which was included to provide a comparison between GP and non-GP evolutionary algorithms, performs poorly and was only able to create bent functions of 6, and balanced functions of 6 and 8 inputs. It fails completely when required to create functions possessing correlation immunity, and for all other setups fails to provide even a single viable result.

Table 3. Experiment results of designing bent Boolean functions.

Method	Q1	Median	Q3	Suc.	Method	Q1	Median	Q3	Suc.
6 inputs									
GAL-EST	14372	31131	67776	100	GAL-SST	9915	30385	70250	100
TGP-EST	399	894	2157	100	TGP-SST	365	485	605	100
CGP-EST	161	331	642	100	CGP-SST	245	425	685	100
LGP-EST	298	634	1333	100	LGP-SST	305	535	1325	100
8 inputs									
GAL-EST	1000000	1000000	1000000	0	GAL-SST	1000000	1000000	1000000	0
TGP-EST	922	2179	3853	100	TGP-SST	565	765	1045	100
CGP-EST	290	531	1159	100	CGP-SST	440	615	1010	100
LGP-EST	624	1359	2496	100	LGP-SST	860	1755	4030	100
10 inputs									
GAL-EST	1000000	1000000	1000000	0	GAL-SST	1000000	1000000	1000000	0
TGP-EST	2016	4311	8710	100	TGP-SST	805	1085	1415	100
CGP-EST	488	934	1718	100	CGP-SST	675	1165	1945	100
LGP-EST	1000	1716	2970	100	LGP-SST	1525	2725	4975	100
12 inputs									
GAL-EST	1000000	1000000	1000000	0	GAL-SST	1000000	1000000	1000000	100
TGP-EST	3220	6879	12757	98	TGP-SST	1125	1385	1815	100
CGP-EST	1237	1994	3211	100	CGP-SST	960	1495	2345	100
LGP-EST	2165	3701	5899	100	LGP-SST	2185	3715	7045	100

Table 4. Experiment results of designing balanced Boolean functions.

Method	Q1	Median	Q3	Suc.	Method	Q1	Median	Q3	Suc.
6 inputs									
GAL-EST	26	41	76	100	GAL-SST	25	65	105	100
TGP-EST	91	204	457	100	TGP-SST	125	185	245	100
CGP-EST	35	61	131	100	CGP-SST	185	265	405	100
LGP-EST	80	124	329	100	LGP-SST	105	155	225	100
8 inputs									
GAL-EST	1451	3444	7216	100	GAL-SST	1805	3045	4735	100
TGP-EST	370	701	1143	100	TGP-SST	325	445	565	100
CGP-EST	115	209	404	100	CGP-SST	305	425	715	100
LGP-EST	195	364	720	100	LGP-SST	265	465	855	100
10 inputs									
GAL-EST	1000000	1000000	1000000	0	GAL-SST	1000000	1000000	1000000	0
TGP-EST	731	1301	2066	99	TGP-SST	525	645	805	100
CGP-EST	210	334	522	100	CGP-SST	305	425	715	100
LGP-EST	321	674	1028	100	LGP-SST	565	985	1815	100
12 inputs									
GAL-EST	1000000	1000000	1000000	0	GAL-SST	1000000	1000000	1000000	0
TGP-EST	1522	2289	4082	100	TGP-SST	765	965	1175	100
CGP-EST	416	619	1095	100	CGP-SST	505	715	965	100
LGP-EST	700	1191	1814	100	LGP-SST	975	1635	2515	100

Table 5. Experiment results of designing resilient Boolean functions.

Method	Q1	Median	Q3	Suc.	Method	Q1	Median	Q3	Suc.
6 inputs									
GAL-EST	1000000	1000000	1000000	0	GAL-SST	1000000	1000000	1000000	0
TGP-EST	195	359	677	100	TGP-SST	205	325	455	100
CGP-EST	116	217	341	100	CGP-SST	145	255	410	100
LGP-EST	217	474	928	100	LGP-SST	245	535	1335	100
8 inputs									
GAL-EST	1000000	1000000	1000000	0	GAL-SST	1000000	1000000	1000000	0
TGP-EST	625	966	1537	100	TGP-SST	525	625	805	100
CGP-EST	246	441	877	100	CGP-SST	325	565	935	100
LGP-EST	688	1139	1929	100	LGP-SST	480	1095	2590	100
10 inputs									
GAL-EST	1000000	1000000	1000000	0	GAL-SST	1000000	1000000	1000000	0
TGP-EST	1231	1706	2387	100	TGP-SST	805	1045	1245	100
CGP-EST	451	861	1377	100	CGP-SST	605	1005	1905	100
LGP-EST	822	1546	2647	100	LGP-SST	1185	2015	3870	100
12 inputs									
GAL-EST	1000000	1000000	1000000	0	GAL-SST	1000000	1000000	1000000	0
TGP-EST	2091	3106	4901	100	TGP-SST	1085	1365	1735	100
CGP-EST	886	1509	2472	100	CGP-SST	1045	1795	3070	100
LGP-EST	1311	2184	4352	100	LGP-SST	1460	2755	5415	100

Table 6. Experiment results of designing Siegenthaler Boolean functions.

Method	Q1	Median	Q3	Suc.	Method	Q1	Median	Q3	Suc.
6 inputs									
GAL-EST	1000000	1000000	1000000	0	GAL-SST	1000000	1000000	1000000	0
TGP-EST	785	1944	3550	100	TGP-SST	1145	1745	3335	100
CGP-EST	507	1104	1874	100	CGP-SST	745	1315	2670	100
LGP-EST	826	1686	3822	100	LGP-SST	1080	2025	6400	100
8 inputs									
GAL-EST	1000000	1000000	1000000	0	GAL-SST	1000000	1000000	1000000	0
TGP-EST	4336	8189	17444	100	TGP-SST	8615	18085	35935	100
CGP-EST	4897	9219	18703	100	CGP-SST	7995	15465	41335	100
LGP-EST	7359	14892	25307	100	LGP-SST	8000	19975	52290	100
10 inputs									
GAL-EST	1000000	1000000	1000000	0	GAL-SST	1000000	1000000	1000000	0
TGP-EST	19424	56054	196702	85	TGP-SST	50765	116905	327265	93
CGP-EST	26188	63944	125741	100	CGP-SST	36360	76195	236620	95
LGP-EST	33291	67382	130232	100	LGP-SST	37445	77385	253780	96
12 inputs									
GAL-EST	1000000	1000000	1000000	0	GAL-SST	1000000	1000000	1000000	0
TGP-EST	81635	214414	1000000	72	TGP-SST	205905	529585	1000000	73
CGP-EST	112127	233049	529320	92	CGP-SST	165990	537935	1000000	72
LGP-EST	100285	198551	359006	97	LGP-SST	92080	262425	631395	87

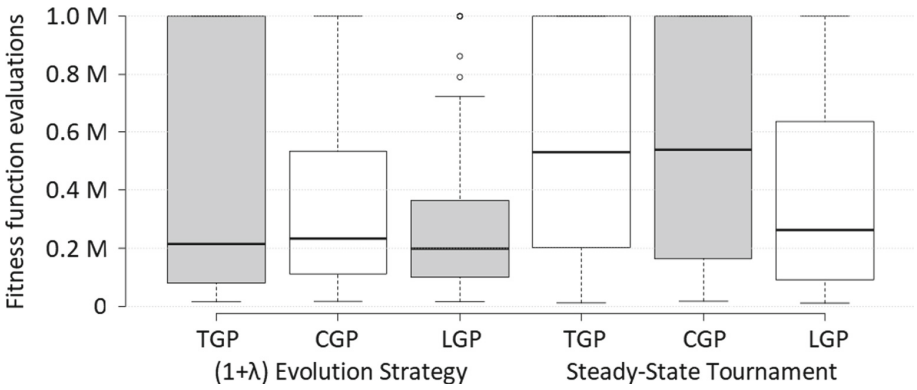

Fig. 2. Comparison of the three genetic programming methods on the task of designing Siegenthaler Boolean functions with 12 inputs, for two different population schemes.

Tables 3, 4, 5 show the importance of crossover for TGP. Making a comparison between the two population schemes, we see that only using mutation increases the median number of evaluations by 2–5 times, as well as severely impacting TGP's reliability, making it the only GP method with less than 100% success rate when designing bent and balanced Boolean functions. Conversely, the results shown in Table 6 illustrate that the use of SST scheme can also have a negative effect and that different population schemes may be ideal for different tasks, even when utilizing the same GP method.

The results provided by CGP show that it indeed performs better with a population scheme that uses mutation as its only genetic operator. The design of 12-input bent functions being the only task where CGP performed better using SST than the EST setup.

The most interesting results, however, are shown in Table 6. LGP which throughout most of the experiments performed in a manner similar to CGP but worse. It manages to be greatly successful with the design of Siegenthaler functions. Though still performing poorly when evolving Siegenthaler functions with a small a number of inputs, LGP managed to cope with additional inputs significantly better than other approaches and over-performed them both when designing Siegenthaller functions with 12 inputs, as is highlighted in Fig. 2. In addition, LGP had the greatest number of successful runs for both population schemes and functions of 10 and 12 inputs.

6 Conclusion and Future Work

In this paper, we have discussed the use of Boolean functions in stream ciphers and examined how suitable various GP methods are for designing them. We evolved functions possessing cryptographically significant properties of nonlinearity, balancedness, correlation immunity, and algebraic degree, with an even

number of 6 to 12 inputs. For each GP method, we have implemented two different population schemes, Steady-State Tournament and $(1 + \lambda)$ Evolutionary Strategy, and to provide a fair comparison, we have performed a one-at-a-time parameter optimization to find the most suitable setup for each method and scheme.

Our results have confirmed that genetic algorithms which evolve functions on the truth table level cope poorly with an increased number of function inputs and can not compete with GP approaches. All three examined GP methods have been shown as competitive, with TGP being best suited for bent and resilient functions, CGP for balanced functions, and LGP for resilient functions with algebraic degree approaching the Siegenthaler's inequality.

For CGP and LGP, use of the SST scheme failed to provide better performance than EST, implying that a simple one-point crossover is not sufficiently complex to improve their performance. Our results also do not confirm that LGP is better than CGP at dealing with an increasing number of function inputs when designing bent Booleans functions. However, they do show this to be the case when evolving resilient functions with a high algebraic degree approaching the Siegenthaler's inequality.

To the best of our knowledge, this is the first work to design balanced, resilient and Siegenthaler functions using LGP, and to experiment with multiple population schemes for each of the GP methods while designing cryptographic Boolean functions. Thanks to this we have shown that the ideal choice of population scheme depends not only on the GP method but also on the type of function being evolved.

Still, our work is just one step in the exploration of the diverse and difficult domain of cryptographic Boolean functions. Future works could expand upon it by including other cryptographic criteria like algebraic and fast algebraic immunity, or focus on the evolution of functions with other cryptographical uses, like the design of Boolean functions with high correlation immunity and minimal hamming weight that can provide protection against side channel cryptographic attacks. Lastly, it would be interesting to experiment with other population schemes and examine their influence on each GP method and type of Boolean function evolved.

Acknowledgments. This work was supported by Czech Science Foundation project 19-10137S.

References

1. Vernam, G.S.: Cipher printing telegraph systems: for secret wire and radio telegraphic communications. J. AIEE **45**(2), 109–115 (1926)
2. Goossens, K.: Automated creation and selection of cryptographic primitives. Master's thesis, Katholieke Universiteit Leuven (2005)

3. Picek, S., Marchiori, E., Batina, L., Jakobovic, D.: Combining evolutionary computation and algebraic constructions to find cryptography-relevant Boolean functions. In: Bartz-Beielstein, T., Branke, J., Filipič, B., Smith, J. (eds.) PPSN 2014. LNCS, vol. 8672, pp. 822–831. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10762-2_81
4. Forrié, R.: The strict avalanche criterion: spectral properties of Boolean functions and an extended definition. In: Goldwasser, S. (ed.) CRYPTO 1988. LNCS, vol. 403, pp. 450–468. Springer, New York (1990). https://doi.org/10.1007/0-387-34799-2_31
5. Fino, B.J., Algazi, V.R.: Unified matrix treatment of the fast Walsh-Hadamard transform. IEEE Trans. Comput. **C-25**(11), 1142–1146 (1976)
6. Meier, W., Pasalic, E., Carlet, C.: Algebraic attacks and decomposition of Boolean functions. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 474–491. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24676-3_28
7. Wu, H.: Cryptanalysis and design of stream ciphers. A Ph.D. thesis of Katholieke Universiteit Leuven, Belgium (2008)
8. Carlet, C.: Boolean functions for cryptography and error correcting codes. Boolean Models Meth. Math. Comput. Sci. Eng. **2**, 257–397 (2010)
9. Armknecht, F.: Algebraic attacks on certain stream ciphers. Ph.D. thesis, University of Rennes (2006)
10. Massey, J.: Shift-register synthesis and BCH decoding. IEEE Trans. Inf. Theory **15**(1), 122–127 (1969)
11. Norton, G.H.: The Berlekamp-Massey algorithm via minimal polynomials. arXiv preprint arXiv:1001.1597 (2010)
12. Didier, F.: Attacking the filter generator by finding zero inputs of the filtering function. In: Srinathan, K., Rangan, C.P., Yung, M. (eds.) INDOCRYPT 2007. LNCS, vol. 4859, pp. 404–413. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-77026-8_32
13. Hodžić, S., Wei, Y., Pašalić, E., Bajrić, S.: Optimizing the placement of tap positions. Ph.D. thesis, Univerza na Primorskem, Fakulteta za matematiko, naravoslovje in informacijske tehnologije (2015)
14. Picek, S., Jakobovic, D., Miller, J.F., Batina, L., Cupic, M.: Cryptographic boolean functions: one output, many design criteria. Appl. Soft Comput. **40**, 635–653 (2016)
15. Carlet, C., Feng, K.: An infinite class of balanced functions with optimal algebraic immunity, good immunity to fast algebraic attacks and good nonlinearity. In: Pieprzyk, J. (ed.) ASIACRYPT 2008. LNCS, vol. 5350, pp. 425–440. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89255-7_26
16. Chose, P., Joux, A., Mitton, M.: Fast correlation attacks: an algorithmic point of view. In: Knudsen, L.R. (ed.) EUROCRYPT 2002. LNCS, vol. 2332, pp. 209–221. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46035-7_14
17. Kavut, S., Maitra, S., Yücel, M.D.: There exist Boolean functions on n (odd) variables having nonlinearity $> 2^{n-1} - 2^{\frac{n-1}{2}}$ if and only if $n > 7$ (2006)
18. Canteaut, A., Trabbia, M.: Improved fast correlation attacks using parity-check equations of weight 4 and 5. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 573–588. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45539-6_40
19. Braeken, A.: Cryptographic properties of Boolean functions and S-boxes. Ph.D. thesis (2006)

20. Tarannikov, Y., Korolev, P., Botev, A.: Autocorrelation coefficients and correlation immunity of Boolean functions. In: Boyd, C. (ed.) ASIACRYPT 2001. LNCS, vol. 2248, pp. 460–479. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45682-1_27
21. Ronjom, S., Helleseth, T.: A new attack on the filter generator. IEEE Trans. Inf. Theory **53**(5), 1752–1758 (2007)
22. Courtois, N.T., Meier, W.: Algebraic attacks on stream ciphers with linear feedback. In: Biham, E. (ed.) EUROCRYPT 2003. LNCS, vol. 2656, pp. 345–359. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-39200-9_21
23. Cramer, N.L.: A representation for the adaptive generation of simple sequential programs. In: Proceedings of the First International Conference on Genetic Algorithms, pp. 183–187 (1985)
24. Kalkreuth, R., Rudolph, G., Droschinsky, A.: A new subgraph crossover for Cartesian genetic programming. In: McDermott, J., Castelli, M., Sekanina, L., Haasdijk, E., García-Sánchez, P. (eds.) EuroGP 2017. LNCS, vol. 10196, pp. 294–310. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-55696-3_19
25. Brameier, M.: On linear genetic programming. Ph.D. thesis, Universitätsbibliothek Technische Universität Dortmund (2004)
26. Millan, W., Clark, A., Dawson, E.: An effective genetic algorithm for finding highly nonlinear boolean functions. In: Han, Y., Okamoto, T., Qing, S. (eds.) ICICS 1997. LNCS, vol. 1334, pp. 149–158. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0028471
27. Picek, S., Jakobovic, D., Golub, M.: Evolving cryptographically sound Boolean functions. In: Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation, pp. 191–192. ACM (2013)
28. Hrbacek, R., Dvorak, V.: Bent function synthesis by means of Cartesian genetic programming. In: Bartz-Beielstein, T., Branke, J., Filipič, B., Smith, J. (eds.) PPSN 2014. LNCS, vol. 8672, pp. 414–423. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10762-2_41
29. Hrbacek, R.: Bent functions synthesis on Intel Xeon Phi coprocessor. In: Hliněný, P., et al. (eds.) MEMICS 2014. LNCS, vol. 8934, pp. 88–99. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-14896-0_8
30. Husa, J., Dobai, R.: Designing bent Boolean functions with parallelized linear genetic programming. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion, pp. 1825–1832. ACM (2017)
31. Picek, S., Jakobovic, D., Miller, J.F., Marchiori, E., Batina, L.: Evolutionary methods for the construction of cryptographic Boolean functions. In: Machado, P., et al. (eds.) EuroGP 2015. LNCS, vol. 9025, pp. 192–204. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-16501-1_16
32. Picek, S., Carlet, C., Jakobovic, D., Miller, J.F., Batina, L.: Correlation immunity of Boolean functions: an evolutionary algorithms perspective. In: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, pp. 1095–1102. ACM (2015)
33. Picek, S., Carlet, C., Guilley, S., Miller, J.F., Jakobovic, D.: Evolutionary algorithms for Boolean functions in diverse domains of cryptography. Evol. Comput. **24**(4), 667–694 (2016)
34. Picek, S., Guilley, S., Carlet, C., Jakobovic, D., Miller, J.F.: Evolutionary approach for finding correlation immune Boolean functions of order t with minimal hamming weight. In: Dediu, A.-H., Magdalena, L., Martín-Vide, C. (eds.) TPNC 2015. LNCS, vol. 9477, pp. 71–82. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26841-5_6

35. Picek, S., Jakobovic, D.: Evolving algebraic constructions for designing bent Boolean functions. In: Proceedings of the Genetic and Evolutionary Computation Conference 2016, pp. 781–788. ACM (2016)
36. Husa, J., Kalkreuth, R.: A comparative study on crossover in Cartesian genetic programming. In: Castelli, M., Sekanina, L., Zhang, M., Cagnoni, S., García-Sánchez, P. (eds.) EuroGP 2018. LNCS, vol. 10781, pp. 203–219. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-77553-1_13
37. Zhang, W., Pasalic, E.: Improving the lower bound on the maximum nonlinearity of 1-resilient boolean functions and designing functions satisfying all cryptographic criteria. Inf. Sci. **376**, 21–30 (2017)
38. Eiben, A.E., Smith, J.E., et al.: Introduction to Evolutionary Computing, vol. 53. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-662-05094-1