Designing Bent Boolean Functions With Parallelized Linear Genetic Programming

Jakub Husa
Faculty of Information Technology
Brno University of Technology
Božetěchova 1/2
Brno, Czech Republic 612 66
ihusa@fit.vutbr.cz

Roland Dobai
Faculty of Information Technology
Brno University of Technology
Božetěchova 1/2
Brno, Czech Republic 612 66
dobai@fit.vutbr.cz

ABSTRACT

Bent Boolean functions are cryptographic primitives essential for the safety of cryptographic algorithms, providing a degree of nonlinearity to otherwise linear systems. The maximum possible nonlinearity of a Boolean function is limited by the number of its inputs, and as technology advances, functions with higher number of inputs are required in order to guarantee a level of security demanded in many modern applications. Genetic programming has been successfully used to discover new larger bent Boolean functions in the past. This paper proposes the use of linear genetic programming for this purpose. It shows that this approach is suitable for designing of bent Boolean functions larger than those designed using other approaches, and explores the influence of multiple evolutionary parameters on the evolution runtime. Parallelized implementation of the proposed approach is used to search for new, larger bent functions, and the results are compared with other related work. The results show that linear genetic programming copes better with growing number of function inputs than genetic programming, and is able to create significantly larger bent functions in comparable time.

CCS CONCEPTS

• **Software and its engineering** → **Genetic programming**; • **Theory of computation** → *Cryptographic primitives*; • **Computing methodologies** → Parallel algorithms;

KEYWORDS

boolean functions, bent functions, cryptography, genetic programming, island model, linear genetic programming, nonlinearity

ACM Reference format:
Jakub Husa and Roland Dobai. 2017. Designing Bent Boolean Functions With Parallelized Linear Genetic Programming. In *Proceedings of GECCO '17 Companion, Berlin, Germany, July 15-19, 2017,* 8 pages.
DOI: http://dx.doi.org/10.1145/3067695.3084220

GECCO '17 Companion, Berlin, Germany

DOI: http://dx.doi.org/10.1145/3067695.3084220

1 INTRODUCTION

Evolutionary algorithms (EAs) are a nature-inspired method of computation, which solves problems using a population of candidate solutions evolved using evolutionary operators. For over two decades EAs have been used to provide designs, comparable and sometimes surpassing those created by human designers. One of the areas where this can be seen is cryptography, where EAs were used to design block cyphers, hash functions, S-boxes, pseudo-random sequences, Boolean functions, and many others [1].

Boolean functions are one of the basic cryptographic primitives. They posses many interesting attributes, which can provide resistance against various types of attacks [2]. One of the attributes is nonlinearity, which allows the Boolean functions to provide nonlinearity to otherwise linear systems. They are often the only nonlinear element in stream cyphers [3]. One of the types of functions possessing extremely high nonlinearity are called bent functions [4]. Because of the enormous size of the space of all possible Boolean functions, bent functions are extremely rare, and there is no known way of how to construct them all [4].

Linear genetic programming (LGP) is one of the forms of genetic programing and broader category of EAs [5]. The program is represented as a linearly executed list of instructions, operating over a set of registers [6]. Each instruction takes two registers as inputs, performs a single operation selected from a predetermined set of operations, and returns the result into one of the registers. The inputs of the program determine the initial value of the registers, while outputs are determined based on the values which are left in the registers after the execution of the program.

Bent functions have been designed by genetic programming (GP) or its specific variant called Cartesian genetic programming (CGP) in particular [7]. A trait common to all of these approaches is the difficulty of designing bent Boolean functions with high number of inputs.

In this paper we propose to search for bent Boolean functions using LGP. The new representation of the program brings several advantages over other approaches. Unlike GP, LGP has the ability to reuse calculated results, and thus avoid many unnecessary calculations and create shorter phenotypes. Over CGP, the new approach limits the number of possible inputs and outputs, making the evolution more effective. In CGP nodes pick their inputs from any of the previously calculated values, making it less likely that the input they selected will contain important previously calculated value, or are strictly limited in what inputs they can choose. In LGP the number of registers is traditionally significantly smaller than the number of instructions, making it more likely that a calculated

Table 1: Frequency of bent Boolean functions.

variables	number of outputs	Boolean functions	bent functions	relative frequency
2	2^2	2^4	2^3	2^{-1}
4	2^4	2^{16}	$\approx 2^{9.8}$	$\approx 2^{-6.2}$
6	2^6	2^{64}	$\approx 2^{32.3}$	$\approx 2^{-31.7}$
8	2^8	2^{256}	$\approx 2^{106.3}$	$\approx 2^{-149.7}$

value will be reused as an input for some further calculation, and that the selected inputs will contain a meaningful value.

For the LGP approach, a parallelization using an island based model of migration is proposed in this paper. A series of experiments detailing the influence of individual evolutionary parameters has been explored, the efficiency of parallelization was tested, and new bent Boolean functions with up to 24 inputs were created, significantly surpassing other approaches.

The paper is organized in the following way. Section 2 provides the necessary definitions for bent Boolean functions and their possible representations. Related works, concerned with design of bent Boolean function using EAs are presented in Section 3. Section 4 proposes a method of parallelization of the LGP approach to designing bent Boolean functions, and describes how the proposed approach was implemented. Following Section 5 describes the setting and results of conducted experiments. The work is concluded in Section 6, where the results are summarized and possibilities for further research are suggested.

2 BENT BOOLEAN FUNCTIONS

A Boolean function f of n variables is a map from the n-dimensional vector space $\mathbb{V}_2 = \mathbb{F}_2^n$ to the two-element field $\mathbb{F}_2$. For a function f, let $f_0 = f(0, 0, ..., 0)$, $f_1 = f(0, 0, ..., 1)$, and $f_{n^{n-1}} = f(1, 1, ..., 1)$. $TT = (f_0, f_1, ..., f_{n^{n-1}})$ is the truth table representation of a function f [8].

Linear Boolean function is either constant 0 function or the exclusive OR (XOR) of any of its variables. Affine function is either a linear function or a complement of a linear function (thus including both constant 0 and constant 1 functions). The nonlinearity N_f of a function is the minimum number of truth table entries that must be changed to convert function f to an affine function [4], a value also known as the Hamming distance. Boolean function is a function with maximum possible nonlinearity for the given number of variables, which is equal to [9]:

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

The high nonlinearity of bent Boolean functions makes them extremely difficult to approximate with linear functions, providing resistance to linear cryptanalysis [4]. The difficulty of searching for bent Boolean functions resides in their scarcity. The relative frequency of bent Boolean functions with up to 8 inputs is shown in Table 1 [4]. Because functions with odd number of inputs cannot satisfy the limit for maximum nonlinearity, all bent functions have even number of variables.

2.1 Boolean function representations

Boolean functions can be represented in multiple ways. While these forms are equivalent from the mathematical standpoint, they each posses their own strengths an weaknesses in terms of their use [10].

Truth table representation stores the function in the form of a binary vector of length 2^n where n is the number of variables. The values in the vector then define the function outputs for all possible combinations of inputs, ordered lexicographically. The weakness of this type of a representation is its exponentially growing size, making the function hard to store, and even harder to transmit.

Walsh transform representation (Walsh spectrum) uses Walsh transform to measure the correlation between the function $f(\vec{x})$ and all linear functions. For $\vec{a} \in \mathbb{F}_2^n$ the linear function $\vec{a}.\vec{x}$ denotes the dot product of $\vec{a}$ and $\vec{x}$, which is defined as [11]:

$$\vec{a}.\vec{x} = x_0a_0 \oplus x_1a_1 \oplus ... \oplus x_na_n \tag{2}$$

where $\oplus$ represents addition modulo two (XOR). Walsh transform of a Boolean function $f(\vec{x})$ is then defined as [11]:

$$W_f(\vec{a}) = \sum_{\vec{x} \in \mathbb{F}_2^n} (-1)^{f(\vec{x}) \oplus (\vec{x}.\vec{a})} \tag{3}$$

This means the Walsh transform calculates the Hamming distance between the function $f(\vec{x})$ and a single linear function. Walsh spectrum is then obtained by applying Walsh Transform on the function $f(\vec{x})$ and all linear functions. This allows for an easy computation of the function's nonlinearity, as a maximum of absolute values in the Walsh spectrum. The transform itself can be calculated in $n \log n$ steps using the Fast Walsh Transform algorithm [12]. The disadvantage of this representation is that, similarly to the truth table representation, its size grows exponentially with the number of variables.

Algebraic normal form (ANF) represents a Boolean function f on $\mathbb{F}_2^n$, as a polynomial in $\mathbb{F}_2[x_0, ..., x_{n-1}]/(x_0^2 - x_0, ..., x_{n-1}^2 - x_{n-1})$. The algebraic normal form is the multivariate polynomial P defined as [10]:

$$P(x) = \oplus_{w \in \mathbb{F}_2^n} h(w).x^w \tag{4}$$

where $h(w)$ is defined as the Möbius inversion principle [13]:

$$h(w) = \oplus_{x \preceq w} f(x), \text{ for any } w \in \mathbb{F}_2^n \tag{5}$$

which means the function is represented as XOR of ANDs of its variables. The benefit of this representation is its size, which does not necessarily grow with the number of variables. The negative property is that outputs of the equation need to be evaluated every time an output for given inputs is needed.

All of these methods are relevant for this paper. The truth table representation is used during the execution of LGP, as vectors of Boolean inputs are transformed into vector Boolean output. The Walsh Transform is used to determine the nonlinearity property of candidate solutions. ANF serves as a basis of our own Boolean function representation, used to store the candidate solutions in population and their transmission between computation cores.

2.2 Boolean functions in cryptography

Boolean functions are most notably used in stream ciphers. These ciphers use symmetrical cryptography to encrypt (decrypt) messages by taking their plain (ciphered) text and adding it modulo two (XOR) with a pseudo-random sequence of equal length. This sequence is called keystream, and is usually created by a linear feedback shift register (LFSR). This register is initialized by a relatively short secret key, and implements a polynomial of degree n to calculate its output. If the polynomial is primitive, the generator repeats its sequence with a period of $2^n - 1$, and has good statistical properties [2].

However, the output of LFSR is not cryptographically secure. Because LFSR is linear, it is vulnerable to linear attacks, and if the attacker knows the output sequence, they can use Berlekamp-Massey algorithm [14] to reconstruct both the generating polynomial and its initial state, thus uncovering the secret key. To create a cryptographically secure keystream, a nonlinear Boolean function takes outputs of multiple LFSRs (combiner generator) or multiple outputs from a single larger LFSR (filter generator), and creates a single output that is difficult to approximate. Errors caused by inaccurate approximation are equivalent to transmission noise [15]. The goal of the Boolean function is to make the keystream so hard to approximate, that even the best approximation will still contain irreparable number of errors, thus preventing the linear attack.

There also exist other cryptographic attacks exploiting different weaknesses, and nonlinearity is not the only property the Boolean function should posses [16]. These properties are balancedness, algebraic degree, algebraic immunity, fast algebraic immunity, and in the case of combiner generators, also resiliency [9]. Aside from their use in stream ciphers, a version of Boolean functions called vectorial Boolean functions (S-boxes) are also used in block cyphers, where high nonlinearity remains one of the desired properties [2].

3 RELATED WORK

EAs have already been used for designing bent functions in the past [3, 17–22]. However, neither of these approaches have used LGP. The search for highly nonlinear Boolean functions has first been performed using genetic algorithms (GA) [17], which have later been used for this purpose again, combined with hill climbing [18]. GA and GP have further been used to find Boolean functions possessing high nonlinearity and other cryptographically important properties [19].

Evolution strategy, GA, GP and CGP have been used to design bent Boolean functions of 8 inputs, and other highly nonlinear functions with other cryptographic properties [20]. Aside from generating bent functions directly, there has also been research into constructions that would allow to construct bent functions based on smaller bent Boolean functions, resulting in creation of bent functions with up to 20 inputs [3].

The most advanced work came in the form of using multiple variations of parallel CGP to design bent Boolean functions of up to 18 inputs [21, 22]. These works are related not only in their goal, but also in using similar manner of parallelization. They also try to create bent functions of multiple sizes, allowing comparison in terms of the ability to deal with increasing demand on the size of functions designed.

4 PROPOSED PARALLELIZATION OF LGP

LGP allows multiple types of parallelization. A trivial way to parallelize calculation is to replace operations over Boolean variables with integers, which can be treated as fixed-length arrays of Boolean variables, making it possible to evaluate multiple independent inputs simultaneously.

Another layer of parallelism can be introduced by using multiple calculation paths. If the number of registers is sufficient, multiple instructions can be calculated in parallel. The degree of parallelization is then limited by the number of instructions that do not overwrite the registers used as inputs or outputs of the other concurrently executed instructions [6]. This method of parallelization shortens the time required to evaluate the fitness function of individuals.

One of the ways to ensure that there will be no conflict between the executed instructions is to split the entire chromosome into multiple sub-chromosomes, each using its own set of registers [23]. Aside from allowing parallel execution, this approach also manages to suppress the negative properties of evolutionary operators like code disruption. When used on complex problems, each sub-chromosome can potentially focus on solving different part of the overall problem, promoting specialization, and allowing further increases in efficiency.

The problem of the non-trivial approaches used for parallelization of fitness function calculation is their limited scalability. They are limited by physical technology, and the effective number of registers, or length of the overall chromosome respectively. A type of parallelization, which can be implemented using a large number of computation cores, is the island based model of evolution [24]. In this approach the population is split into multiple separate sub-populations, each assigned to a single core. In this coarse-grained model, each of these populations is evolved independently, and the best found individuals are occasionally transmitted along a given topology, and integrated into the other sub-populations [25]. If this migration happens asynchronously, each core can spend the maximum of its runtime by performing computation, without having to wait for outside inputs. This is useful as the evaluation of various individuals can require different amounts of time to evaluate based on the length of their phenotype. The lack of need for a fine-grained synchronization makes this approach highly scalable, in some cases reaching super-linear speedup [26].

4.1 Implementation of LGP

The implementation has been done in standard C++ and parallelized using Message Passing Interface (MPI). The communication happens over a simple one-way ring topology, because preliminary experiments have shown that excessive communication has detrimental effect on the quality of provided results.

Individuals are stored in a steady-state population, and selected via tournament. Initial population is generated randomly, and new individuals are created by a single point-crossover of the tourney's best individuals, or if the crossover rate is not supposed to happen, as a plain copy of the best individuals. In both cases the newly created individual also undergoes universal mutation, where each of its instruction has a given chance of being replaced by a new randomly generated instruction. The worst individual in the tourney is then replaced by the new one, and its fitness is evaluated. Bent functions

always factor all of their primary inputs, and favoring functions that use greater number inputs speeds up the evolutionary process. For this reason, the fitness is defined as:

$$fitness = N_f * 100 + u \tag{6}$$

where N_f is the nonlinearity, and u represents the number of primary inputs used in the individuals phenotype. Because the goal is to create Boolean functions in shortest possible time, other possible parameters, like the length of the generated phenotypes, are not factored.

During fitness evaluation the chromosome is searched for active instructions, which create the individual's phenotype. These instructions are then applied on all possible inputs, 64 bits at a time, and stored in a truth table. It is then translated into Walsh spectrum using the Fast Walsh-Hadamard transform. Nonlinearity is calculated as a maximum of absolute values in the Walsh spectrum.

Each instruction takes two inputs, which can be either one of the primary inputs, or one of the registers, applies an operation, and provides one output, which is stored into one of the registers. Operation to be applied is selected from a set of only two instructions {AND, XOR}, based on ANF in which all bent functions can be represented. This however does not guarantee that the generated functions will be in ANF, as the other of operations can be different. Following is the example of a phenotype representing a 14-bit bent function:

```
r0 = X11 AND X2; r5 = X3 XOR X12; r5 = X9 AND r5;
r7 = X4 AND X8; r3 = X14 AND X13; r6 = X6 AND X3;
r8 = r7 XOR r3; r9 = r6 XOR r8; r6 = X1 AND X5;
r5 = r5 XOR r0; r5 = X6 XOR r5; r8 = X10 AND X7;
r0 = r5 XOR r8; r0 = r9 XOR r0; r0 = r6 XOR r0;
```

where X1-X14 are the primary inputs, r0-r9 are the registers. Before the fitness calculation all registers are initialized to zero, and r0 is used to store the output.

After the fitness of a new individual is calculated, it is compared to the fitness of the best known individual from the given computation core, and if new best fitness was found, the value is updated, and the individual is sent to the core's neighbors, along with information about its fitness. The receiving core integrates the individual into its own population during the evolutionary process, by skipping creation and fitness evaluation of a new individual, and replacing the tournament's loser, by the received individual, and possibly forwarding it to the next core in topology, according to the received value of fitness.

5 RESULTS

The implementation of LGP has been used to conduct three experiments. The first experiment focused on finding the optimized evolutionary parameters. The second examined the effectiveness of used parallelization model. The third experiment designed bent functions of largest size possible. To obtain statistically meaningful results, 100 runs have been performed for every experiment. Due to limitations on available resources, all runs were limited to 1 hour of maximum runtime.

All experiments were performed on a computing cluster using up to four nodes with the following hardware configuration: 2 x Intel Xeon E5-2680v3 processor, 2.5 GHz, 12 cores; 128GB RAM, 5.3 GB per core, DDR4@2133 MHz; InfiniBand FDR56 network connection.

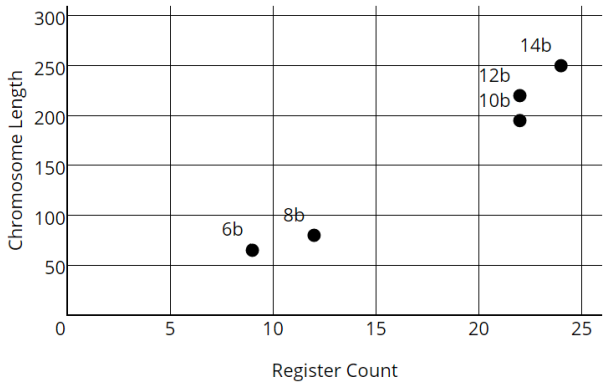

Figure 1: Combination of chromosome length and register count parameters, providing the best mean runtime for bent functions of various sizes.

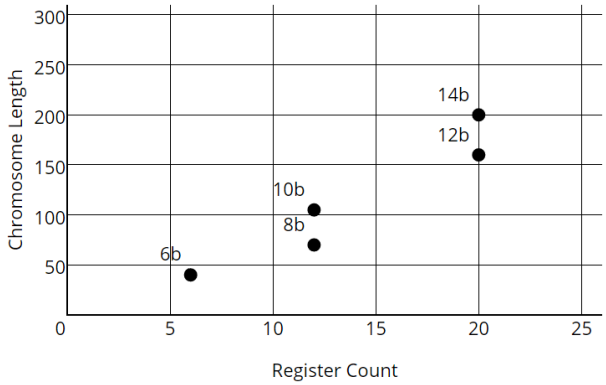

Figure 2: Combination of chromosome length and register count parameters, providing the best median runtime for bent functions of various sizes.

5.1 Search for LGP parameters

Because, to our best knowledge, LGP has never been used for designing bent Boolean functions before, the first experiment focuses on finding the optimal evolutionary parameters. To provide information whether the values are dependent on the desired function size, the search is conducted on bent functions of 6 to 14 inputs. Six parameters are evaluated, two at a time, in order to limit the size of the search space. Namely they are {register count, chromosome length}, {crossover rate, mutation rate}, and {tournament size, population size}. Initially the values for the other four parameters are selected based on previous experience. When the experiment is done, the values considered best for the largest tested bent function are taken as the new initial values, and the entire experiment is repeated. These values are selected, because they are most relevant to design of other even larger bent functions in the other experiments. The experiment has been performed on a computation cluster using

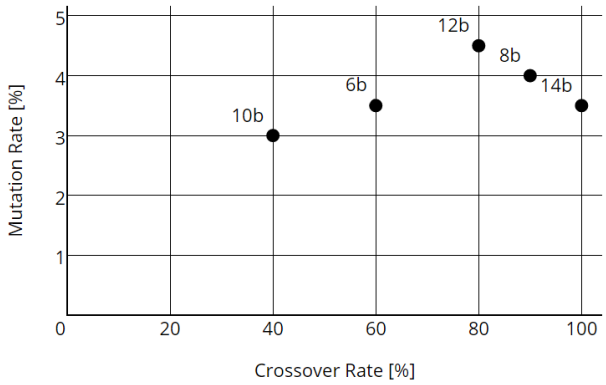

Figure 3: Combination of crossover and mutation rate parameters, providing the best mean runtime for bent functions of various sizes. Combinations providing the best median runtime were similar.

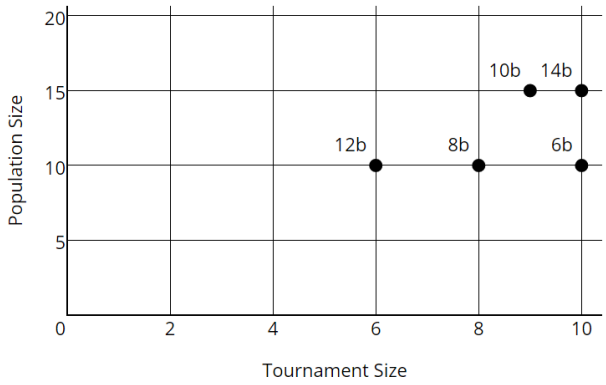

Figure 4: Combination of population and tournament size parameters, providing the best mean runtime for bent functions of various sizes. Combinations providing the best median runtime were similar.

24 cores. The measured variable was the runtime required to find a bent function with the desired number of inputs.

The first examined pair of parameters were the chromosome length and the number of registers added to the single mandatory output register. In the experiment's first iteration, the tested values ranged from 30 to 300 (step size 30) for chromosome length, and 3 to 30 (step size 3) for register count respectively. In the second iteration, the experiments were performed with decreased granularity, focusing on the area around the expected value. Ordered in ascending order based on the function size, the granularity of chromosome length was {5,10,15,15,15} instructions and {1,2,2,2,2} registers.

The combination of these parameters that resulted in the best mean and median runtime are shown in Figures 1 and 2, respectively. Larger bent functions require longer chromosomes and more registers available. The two values remain proportional, with the ideal rate staying around 10 instructions for every register. The parameters to obtain the best median runtime are lower than those

Table 2: Evolutionary parameters used for designing large bent functions.

Parameter	Value
Chromosome length	250
Crossover prob.	100%
Mutation prob.	3.5%
Register count	24
Tournament size	6
Population size	10

providing best mean runtime. This shows, that while shorter chromosomes with fewer registers often find the solution faster, they also lead to a significant number of extremely long runs. For the purpose of other tests, the setting leading to best mean runtime have been selected.

The second examined pair of arguments was the crossover and mutation rates. In both iterations and for all function sizes, the examined mutation rate ranged from 0.5 to 5% (step size 0.5%) and the crossover rate ranged from 0 to 100% (step size 10%). The results providing the best mean runtime value is shown in Figure 3, with the results for best median runtime being similar. LGP performs best with high probability of crossover, in the range of 40–100%, and small mutation rate around 3.5%, regardless of the number of function inputs.

The third examined pair of parameters were the population and tournament sizes. In both iterations and for all function sizes the tournament size ranged from 2 to 10 (step size 1). In the first iteration, the examined population size ranged from 10 to 100 (step size 10). Because smaller populations performed better for all function sizes, the range was changed to 10 to 50 (step size 5) in the second iteration.

Figure 4 shows the results providing the best mean values with the best median values being extremely similar. Regardless of the function size the best results are obtained with small population and a large tournaments, creating very high evolutionary pressure. In some cases, the tournament and population sizes were the same, thus effectively leaving the individuals with intermediate fitness out of the evolutionary process. This creates doubt over the suitability of steady-state population scheme for the purposes of this LGP approach, and suggests a venue for further study.

Based on these results, the parameters shown in Table 2 were selected as the best for design of large bent Boolean functions using the LGP approach.

5.2 Evaluation of parallelization

To evaluate the effectiveness of the island model using one-way ring topology, an experiment was conducted, using the same evolutionary settings with different number of computation cores. Bent functions with 20 inputs are designed using the best setting from the previous experiment. The number of inputs was selected as the largest function size the implementation is able to reliably create under one hour using a single core. The examined values are the total runtime, runtime per core, and total number of fitness function evaluations.

Table 3: Results obtained by design of bent functions with 20 inputs using different number of cores.

cores	runtime [s]	core run [s]	evaluations	eval run [s]
1	564.2840	564.2840	25007.63	0.0226
3	165.8795	497.6385	28051.02	0.0177
6	118.7637	712.5821	40105.78	0.0178
12	61.2441	734.9292	41348.42	0.0178
24	61.0679	1465.6299	53444.42	0.0274
48	32.0727	1539.4909	56817.47	0.0271
96	21.5422	2068.0509	75815.44	0.0273

In this experiment the parameters shown in Table 2 were used to create bent Boolean functions with 20 inputs, using 1 to 96 computation cores. The results of the experiment presented in Table 3 show that parallelization is successful and adding multiple computation cores significantly improves the application runtime. However the speedup is not linear. Column *core run* shows the increase in the overall runtime for all computation cores combined, with the maximum number of cores requiring nearly four times as much total runtime to perform the same task as a single independently performing core.

Part of this increase can be explained by the increase in total number of fitness function evaluations shown in the column *evaluations*. The gradual increase in this value implies that the communication topology used to parallelize LGP approach is not ideal. While the sub-populations are still being evolved at their own rate, the lag in distribution of the globally best known individual limits their effectiveness. The analysis of the influence of different topologies could motivate further study. In the last column marked as *eval run*, the average runtime per core needed to perform a single fitness function evaluation is shown.

Evaluations using 12 of fewer cores required significantly less time to perform a single fitness function evaluation, than when a larger number of cores was used. This can be explained by examining the underlying hardware configuration. Because the computation nodes are equipped with two 12-core processors, experiments using this number of cores are localized to a single physical processor using a single physical memory, allowing extremely fast communication.

The single core implementation used during the experiments was pruned from all code pertaining the communication and synchronization with the intent to potentially increase the speed of evaluation, and were run 24 at a time on a single node. While these runs should in theory be entirely separate, physically the HW configuration likely caused competition during memory access. Therefore, while removing the need for communication sped up the evaluation in comparison to 24 cooperating processes, the results were worse, than when the node was only partially loaded.

The obtained results can be compared to similar experiments performed using CGP [22], where parallelization using 4 cores reduced the average runtime by factor of 3.35, and 40 cores by a factor of 9.78. The information about the number of fitness function evaluations is not available, but the effectiveness is similar to LGP using a comparable core count.

5.3 Design of bent Boolean functions

The purpose of the last experiment is to create the largest bent functions possible, and compare the results provided by LGP to other approaches. Functions of 6 to 24 inputs were designed using the settings found in first the experiment presented in Table 2. While the evolutionary parameters ideal for larger functions may be different than for 14-bit bent functions, the amount of runtime required for their evaluation makes it impossible to perform enough runs to obtain statistically meaningful results.

The results, obtained using 96 cores, are shown in Table 4. All runs performed with 6 to 22 inputs have ended successfully by creating a bent Boolean function. For 24 inputs, 2 out of the 100 runs have failed to create a bent function, and they were accounted into the dataset with the values they possessed at the end of their one hour run somewhat skewing the average values.

For functions with 8 to 12 inputs the runtime stayed nearly the same due to the time required for initiation of communication infrastructure, and evaluation of initial population. Lasting about 20 milliseconds, its influence on the runtime of larger functions is negligible, and the runtime they require grows at increasing rate.

Other characteristics of the evolution have been considered as well. For all function sizes, the average length of phenotype is around 50 instructions, even for smaller bent functions. This is likely caused by the length of chromosome and the lack of optimization for this parameter. The number of fitness function evaluations shows one of the main reasons for the rapid increase in runtime. Larger bent functions appear with lower frequency and the evolutionary process needs to perform more generations to reach its goal. This, combined with the growing size of truth tables, shows why larger bent functions are so hard to find. The last measured parameter shows the total number of messages sent, which averages at one message per 15 to 40 fitness function evaluations, representing the relative frequency of migrations between islands.

The results of this test are compared to results provided by CGP in other works, as shown in Table 5. The comparison shows that the LGP approach is significantly faster than both CGP implementations. This could be at least partially accounted to the use of more modern hardware. However, LGP also performs better in regard of the relative increase in runtime required for design of larder functions. This makes LGP more suitable for creation of very large bent functions, and allowed the creation of bent functions with up to 24 inputs in manageable time.

The rate at which runtime increases suggests that, given more time, LGP should be able to create even larger functions. The expected growth rate is a roughly tenfold increase in runtime for every two additional inputs. Because the experiment used the setting ideal for designing 14-bit bent functions, it should also be possible to shorten the runtime by applying evolutionary parameters suitable for larger functions. The values of chromosome length and number of registers in particular have been shown to increase with increasing number of inputs. Analysis of these parameters on runtime required for design of large bent functions presents an opportunity for further study.

Table 4: Results of bent function design.

input size	time [s] mean	median	deviation	mean values phenotype	evaluations	messages
6	0.0197	0.0193	0.0026	63.90	6853.52	423.29
8	0.0204	0.0199	0.0022	53.66	10436.91	651.67
10	0.0225	0.0219	0.0036	51.76	13323.56	914.55
12	0.0311	0.0303	0.0041	47.87	17178.94	1184.89
14	0.0824	0.0766	0.0233	48.32	23918.58	1472.75
16	0.3459	0.3112	0.1391	46.24	30941.52	1733.66
18	2.2523	1.8654	1.2530	45.98	49270.83	2089.75
20	19.4152	14.2275	16.1421	48.03	69168.67	2375.32
22	230.9818	150.1619	181.5669	47.28	116466.26	2774.53
24	1095.7768	770.8292	845.4280	50.09	127075.23	3115.01

Table 5: Comparison of LGP with other state of the art approaches.

input size	LGP time [s]	inc.	CGP1 [22] time [s]	inc.	CGP2 [21] time [s]	inc.
12	0.0311	–	0.3859	–	0.84	–
14	0.0824	2.65	15.806	40.96	3.62	4.31
16	0.3459	4.2	636.13	40.25	40.88	11.29
18	2.2523	6.51	–	–	814.02	19.91

6 CONCLUSION

In this paper, a new approach to designing bent Boolean functions has been proposed using LGP. While other forms of GP have been used for design of bent and other interesting types of Boolean functions in the past, they have all struggled to maintain their efficiency when used to design functions with large number of inputs. While some increase in runtime is inevitable due to the increasing scarcity of larger bent functions, it is crucial for these increases to be minimized.

To make use of modern hardware, the approach is parallelized using island model with one-way ring topology to achieve significant decrease in runtime, and design bent Boolean functions that could not be otherwise created within an acceptable time frame. The scalability of the approach was worse than linear, and presents a potential for further study.

Influence of multiple evolutionary parameters on the performance of LGP were explored, and values suitable for designing bent Boolean functions of various sizes were found. The ideal length of the chromosome and number of registers scales with the number of inputs, and the ratio between the two values remains the same. LGP woks best with high rate of crossover (40–100%), and moderate rate of mutation (around 3.5%), regardless of the number of inputs. LGP also works best with high evolutionary pressure, created by small population combined with large size of tournament selection, suggesting a possibility of further experimentation using various population schemes.

The approach has been used to design bent Boolean functions of up to 24 inputs, which is significantly more than with other methods. Comparison with other similar works also shows that LGP copes better with increasing demand on the size of functions it designs, and shows promise for further use.

Aside from the possibilities already mentioned, future works will focus on using this new approach to create other types of Boolean functions with other cryptographically interesting properties, like balancedness, algebraic degree, algebraic immunity, fast algebraic immunity and resilience.

ACKNOWLEDGMENTS

This work was supported by the Czech science foundation under the project GP16-08565S.

REFERENCES

[1] Stjepan Picek and Marin Golub. On evolutionary computation methods in cryptography. In *MIPRO, 2011 Proceedings of the 34th International Convention*, pages 1496–1501. IEEE, 2011.
[2] Ann Braeken. *Cryptographic properties of Boolean functions and S-boxes*. PhD thesis, phd thesis-2006, 2006.
[3] Stjepan Picek and Domagoj Jakobovic. Evolving algebraic constructions for designing bent boolean functions. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, pages 781–788, New York, NY, USA, 2016. ACM.
[4] Jon T Butler and Tsutomu Sasao. Logic functions for cryptography-a tutorial. 2009.
[5] Markus F. Brameier and Wolfgang Banzhaf. *Linear Genetic Programming*. Springer Publishing Company, Incorporated, 1st edition, 2010.
[6] Markus Brameier. *On linear genetic programming*. PhD thesis, Universität Dortmund, 2005.
[7] Julian F. Miller and Peter Thomson. *Cartesian Genetic Programming*, pages 121–132. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.
[8] Jennifer L Shafer, SW Schneider, Jon T Butler, and Pantelimon Stanica. Enumeration of bent boolean functions by reconfigurable computer. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 265–272. IEEE, 2010.
[9] Claude Carlet. Boolean functions for cryptography and error correcting codes. *Boolean models and methods in mathematics, computer science, and engineering*, 2:257–397, 2010.
[10] Stjepan Picek, Elena Marchiori, Lejla Batina, and Domagoj Jakobovic. Combining evolutionary computation and algebraic constructions to find cryptography-relevant boolean functions. In *International Conference on Parallel Problem Solving from Nature*, pages 822–831. Springer, 2014.
[11] Réjane Forrié. *The Strict Avalanche Criterion: Spectral Properties of Boolean Functions and an Extended Definition*, pages 450–468. Springer New York, New York, NY, 1990.
[12] B. J. Fino and V. R. Algazi. Unified matrix treatment of the fast walsh-hadamard transform. *IEEE Transactions on Computers*, C-25(11):1142–1146, Nov 1976.
[13] Willi Meier, Enes Pasalic, and Claude Carlet. *Algebraic Attacks and Decomposition of Boolean Functions*, pages 474–491. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[14] James Massey. Shift-register synthesis and bch decoding. *IEEE transactions on Information Theory*, 15(1):122–127, 1969.
[15] Claude E Shannon. Communication theory of secrecy systems. *Bell Labs Technical Journal*, 28(4):656–715, 1949.
[16] Hongjun Wu. Cryptanalysis and design of stream ciphers. *A PhD thesis of Katholieke Universiteit Leuven, Belgium*, 2008.
[17] William Millan, Andrew Clark, and Ed Dawson. An effective genetic algorithm for finding highly nonlinear boolean functions. In *International Conference on Information and Communications Security*, pages 149–158. Springer, 1997.
[18] William Millan, Andrew Clark, and Ed Dawson. Heuristic design of cryptographically strong balanced boolean functions. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 489–499. Springer, 1998.
[19] Stjepan Picek, Domagoj Jakobovic, and Marin Golub. Evolving cryptographically sound boolean functions. In *Proceedings of the 15th annual conference companion on Genetic and evolutionary computation*, pages 191–192. ACM, 2013.
[20] Stjepan Picek, Domagoj Jakobovic, Julian F. Miller, Lejla Batina, and Marko Cupic. Cryptographic boolean functions: One output, many design criteria. *Applied Soft Computing*, 40:635 – 653, 2016.
[21] Radek Hrbacek. *Bent Functions Synthesis on Intel Xeon Phi Coprocessor*, pages 88–99. Springer International Publishing, Cham, 2014.
[22] Radek Hrbacek and Vaclav Dvorak. *Bent Function Synthesis by Means of Cartesian Genetic Programming*, pages 414–423. Springer International Publishing, Cham, 2014.
[23] Carlton Downey and Mengjie Zhang. Parallel linear genetic programming. In *Proceedings of the 14th European Conference on Genetic Programming*, EuroGP'11, pages 178–189, Berlin, Heidelberg, 2011. Springer-Verlag.
[24] Theodore C. Belding. The distributed genetic algorithm revisited. In *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 114–121, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.
[25] M. Ruciński, D. Izzo, and F. Biscani. On the impact of the migration topology on the island model. *Parallel Comput.*, 36(10-11):555–571, October 2010.
[26] David Andre and John R. Koza. A parallel implementation of genetic programming that achieves super-linear performance. *Inf. Sci.*, 106(3-4):201–218, May 1998.