Evolving Cryptographic Boolean Functions with Minimal Multiplicative Complexity

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

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

***Abstract*—The multiplicative complexity (MC) is a cryptographic criterion that describes the vulnerability of a Boolean function to certain algebraic attacks, and in many important cryptographic applications also determines the computational cost. In this paper, we use Cartesian genetic programming to find various types of cryptographic Boolean functions, improve their implementation to achieve the minimal MC, and examine how difficult these optimized functions are to find in comparison to functions than only need to satisfy some base cryptographic criteria. To provide a comparison with other state-of-the-art optimization approaches, we also use our method to improve the implementation of several generic benchmark circuits. Our results provide new upper limits on MC of certain functions, show that our approach is competitive, and also that finding functions with an implementation that has better MC is not mutually exclusive with improving other performance criteria.**

***Index Terms*—Genetic programming, Cartesian genetic programming, cryptography, multiplicative complexity, optimization.**

I. Introduction

Boolean functions are one of the basic cryptographic primitives used in implementation of many modern encryption algorithms, sometimes being the only nonlinear element of the cipher [1]. Their utilization ranges from being used as a filtering function of a pseudo-random number generator in stream ciphers [2], to being used for construction of safe substitution boxes in block ciphers, or masking the implementation of other cryptographic algorithms to guard them against side-channel attacks [3], and many more.

The cryptographic uses of a Boolean function are governed by its cryptographic properties, which determine how resistant it is against specific types of attacks. Because many of the desirable properties are in conflict with one another, functions that possess good values of multiple properties simultaneously are extremely rare, and how to find them has been a subject of extensive research [2]–[7].

Evolutionary algorithms (EA) are a set of heuristic search algorithms inspired by the natural evolutionary process [8]. They work by maintaining a population of individuals, each of which contains a *genotype* that encodes a candidate solution to the presented problem. The quality of each individual is appraised by a *fitness function* that evaluates how close the individual is to the desired solution and assigns it a *fitness value*. The algorithm then uses this fitness to select the individuals from which to spawn a new population. There are two main ways how new individuals are created. *Mutation* takes a single individual and performs a small random change in its genotype, following the idea that better solutions can be found by making small incremental changes to an already good solution. *Crossover* takes two individuals and mixes their genotypes together, following the idea that better solutions can be found by combining the good parts of already good solutions.

A Boolean function can be represented in several ways, for example, a truth table, graph, Walsh spectrum, or a logical expression. One notable subset of EAs is *Genetic programming* (GP), which uses the logical expression representation that allows it to efficiently store, mutate and recombine even very large solutions using a comparatively short genotype [8]. For example, in the context of cryptographic Boolean functions, a *bent* function with a truth table of length 2^n (with n being the number of its inputs) can be encoded using as few as $n - 1$ operators.

In this paper, we use a specific type of GP called Cartesian genetic programming (CGP) to evolve four types of cryptographic Boolean functions, namely: *bent*, *balanced*, *resilient*, and *masking*. Each of these functions needs to satisfy a different set of cryptographic properties, including *nonlinearity*, *balancedness*, *correlation immunity*, and *Hamming weight*. On top of the cryptographic properties, we also consider the *multiplicative complexity of implementation* (MCI) of these functions, which provides an upper bound for their MC [9]. To the best of our knowledge, this is the first work that has considered this criterion when evolving these types of cryptographic Boolean functions.

Our first objective is to find Boolean functions with given cryptographic properties and reduce their MCI as much as possible. Our second objective is to examine how difficult functions with minimal MCI are to find in comparison to regular cryptographic Boolean functions. Our third objective is to reduce the MCI of several benchmark circuits from the EPFL Combinational Benchmark Suite [10] (CAVLC encoder,

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

978-1-7281-6929-3/20/$31.00 ©2020 IEEE

ALU control unit, 8-bit decoder, and int-to-float converter), to compare our method with other state-of-the-art optimization approaches.

The rest of this paper is organized as follows. Section II provides the necessary theoretical background, defines the cryptographic properties of the Boolean functions we create, and details the combinational circuits we use as our benchmark. Section III offers an overview of related works done on our subject. Section IV describes the implementation of our method and the fitness function used for each task. Section V shows our experiment setup, how we optimized the evolutionary parameters, what values of cryptographic properties we reached, what experiments we performed, and provides a discussion of the results. Finally, the paper concludes in Section VI, which summarizes the results and considers the possibilities for future research.

II. Preliminaries

Boolean function is a function of the type $B^n \rightarrow B$, where $B \in \{0,1\}$ and $n \in \mathbb{N}$. In other words, it is a function that takes multiple binary inputs and provides a single binary output. Boolean functions are usually represented as an expression consisting of primary inputs and logical operators. By evaluating this expression, we obtain a *truth table* that specifies what output corresponds to each possible combination of input values [11], [12].

The *nonlinearity* (NL) of a Boolean function is defined as the *Hamming distance* between its truth table, and the truth table of the nearest affine function (the number of bits that would need to be flipped to turn one truth table into the other). An *affine function* is a linear function that may have its output inverted. And finally, a *linear function* is defined as a function created by logical XOR of any number of its primary inputs.

Functions that reach the theoretical upper limit of nonlinearity given by equation.

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

are called *bent* functions. They are rare and exist only for an even number of inputs.

The number of ones in the truth table of a function is called *Hamming weight* (HW). If the truth table contains the same number of ones and zeroes, we call the function *balanced*. If a function is not balanced, its inputs and outputs are not statistically independent, making it unsuitable for most cryptographic applications. Bent functions are never balanced, but they have other uses, for example, as a non-linear component of other larger Boolean functions [11].

A function is said to possess *correlation immunity* (CI) if its output is statistically independent of any one of its primary inputs. In other words, this means that if we were to take a function and split its truth table in half based on the value of any one of its primary inputs, the two halves would contain the same number of ones. If the same were to be true even if we split the truth table in quarters based on the value of any two of the primary inputs, the function would possess correlation immunity of a second order. Similarly, we can define the correlation immunity of up to order *n* (the number of inputs), which is only reached by functions of constant one or constant zero [12].

TABLE I
Circuits selected for optimization from the EPFL benchmark suite, and their properties.

Circuit	inputs	outputs	MCI
Cavlc	10	11	704
Controller	7	26	199
Decoder	8	256	304
Int2float	11	7	267

Functions that are both balanced and immune to first-order correlation are called *resilient* and are suitable for use as a filtering function in pseudo-random number generator of a stream cipher [2]. Functions with a high order of correlation immunity and low (but non-zero) HW are suitable for protecting the implementation of various other cryptographic algorithms against side-channel attacks, by masking the processed intermediary values. We refer to them as *masking* functions [3], [7].

The *multiplicative complexity* (MC) is defined as the *minimum* number of two-input ANDs necessary to implement a function using only logical AND, XOR, and NOT operators [13]. The *multiplicative complexity of implementation* (MCI) is defined as the *actual* number of two-input ANDs used to implement the function using the same set of operators [9]. As a result, known MC determines the target value when minimizing the MCI of a given function, while known MCI provides an upper bound for functions whose MC is yet to be determined.

For 5 or fewer inputs, the MC of any function is at most $n-1$ [14]. For 6 to 8 inputs, there exist functions with MC of up to n. For 9 and more inputs, the situation is largely unexplored, because MC of any function with more than 6 inputs is extremely difficult to calculate. To the best of our knowledge, there has never been found any Boolean function with MC greater than n [15].

The importance of MC stems from the fact that functions with low MC are vulnerable to certain algebraic attacks [14]. MCI, on the other hand, is important in functions used to implement certain high-level cryptographic protocols (fully homomorphic encryption, zero-knowledge protocols, multiparty communication, and non-interactive proofs of knowledge) [16]–[18]. In these applications, the linear operations (XOR and NOT) can be computed locally, while the nonlinear operations (AND) require cooperation between multiple parties, resulting in a significant performance bottleneck, which makes the need to optimize these functions for a minimal number of ANDs (rather than the overall number of operations) readily apparent [16], [19].

A. Benchmark Circuits

To provide a comparison with other optimization methods, we have selected four circuits from the EPFL Combinational

Benchmark Suite [10] and used our algorithm to minimize their MCI. The individual circuits were selected based on the number of their inputs, outputs, and initial MCI, which can be seen in Table I.

Coding-cavlc is a part of Context-adaptive variable-length coding (CAVLC) video encoder for the H.264/MPEG-4 AVC format, containing look-up tables for coefficients, total zeros, trailing ones and other signals.

ALU control unit is a combinational circuit from simple Arithmetic Logic Unit Controller containing various signals controlling ALU operations, register destinations, memory operations, and jumps.

Decoder is a traditional 8-bit decoder, which converts a number from a binary representation into one of 256 separate signals.

Int to float converter is a circuit which coverts 10-bit integer numbers, into a floating point representation using an exponent of length 3 and mantissa of length 4.

III. Related Works

The earliest application of EAs to create cryptographic Boolean functions was the use of a *genetic algorithm* to find functions with high NL [20]. The works that followed expanded the list of considered cryptographic criteria to create functions that are balanced [5], [6], resilient [3], [5], or suitable for masking [3], [4], [7], as well as functions possessing additional cryptographic properties, like high *algebraic degree* and *algebraic immunity* [5]–[7], [21], and others. Importantly, the use of EAs in these works resulted in the discovery of new Boolean functions with superior cryptographic properties.

GP has first been used to search for functions with 8 inputs and optimal values of multiple cryptographic properties [22], while CGP has been used to create bent functions with up to 16 inputs [23]. Another variant of GP, the *linear genetic programming* (LGP), has been used to create bent functions with up to 24 inputs [1].

Multiple studies have shown that GP and its variants significantly outperform other EAs, as well as other heuristic search algorithms like *hill climbing*, *simulated annealing*, and *particle swarm optimization* [2], [5], [6], [21]. The individual variants of GP are mutually competitive, and each is suitable for evolving Boolean functions with a different combination of cryptographic properties [2], [4].

Current state-of-the-art techniques for MCI optimization do not use an evolutionary approach. They rely heavily on manual decomposition and optimization, and while they can guarantee an optimal result, they are effective only for very small circuits [24], [25].

Recently, a new automated *logic synthesis* approach has been proposed. Starting with a working circuit, it implements an *enumeration cut algorithm* to search for all possible subcircuits of up to 6 inputs (for whom the optimal MCI is known) and rewrites them with a functionally-equivalent but MCI-optimal replacement. Unlike the manual approaches, this technique can be used even on very large circuits with over a hundred thousand of nodes. On average, it managed to improve the MCI of benchmark circuits by 34%, but does not guarantee optimal results and fails to optimize circuits that can't be improved by local changes [9].

This approach has then been combined with two additional operators, *resubstitution* and *refactoring*, to create a *logic synthesis toolbox* (LST). Resubstitution tries to express the function of each node by using other nodes already present in the circuit, while Refactoring helps to overcome local optima by re-synthesizing large parts of the circuit from scratch without using any of the existing nodes. On average, LST improved the MCI of benchmark circuits by an additional 15% over the original ECA technique [26].

IV. Implementation

Cartesian genetic programming (CGP) is a type of GP originally developed for the automated design of combinational circuits [27]. Each individual in the population represents a circuit with several primary inputs (n_i) and outputs (n_o) and consists of several rows (n_r) and columns (n_i) of nodes forming a Cartesian grid. The size of each individual in CGP is therefore constant, unlike the tree-based representation used in regular GP. Each node in the grid represents its own trivial function, has a number of inputs defined by its arity, and provides a single output. When evolving Boolean functions or combinational circuits, the arity is two, the trivial function is a logical operator, and the output is a binary value. The inputs of each node can connect either to a node in one of the preceding columns, up to *L-back* columns away, or the primary inputs of the overall circuit. Meanwhile, the primary outputs of the circuit can connect to any node in its grid.

Topologically, each individual forms a directed acyclic graph, and the limitations put on its size and structure by CGP make it suitable for implementation in hardware. If the circuit is to be implemented only in software, the structural restrictions are usually loosened and the circuit is represented using a grid with a single row and unlimited *L-back* [27].

The advantage CGP has over regular GP, is that the output of each node can be used in multiple following calculations, making it easier to find an implementation with fewer operators. Its disadvantage, on the other hand, is that this property also makes it extremely difficult to perform a meaningful crossover between two individuals [28]. As a result, CGP usually produces new individuals only by mutation and does not benefit from having a large population, which usually consists of a single parent and a small number (λ) of its immediate offspring [27], [29].

Our implementation of CGP is written in C++ and parallelized via a *Message Passing Interface* (MPI). We use an *island model* of parallelization where each core maintains its own separate population of candidate solutions and shares its individuals with other neighboring cores over a cyclic, 2-dimensional, uni-directional topology.

Our function set consists of XOR and AND, each of which can have its first, second, both, or neither input inverted (giving us a total of 8 trivial functions). When evolving cryptographic Boolean functions, we create the initial population randomly

using a standard *Mersenne Twister 19937* generator seeded by a stochastic process. When evolving benchmark circuits, we initialize the individuals with a known working solution and randomize any remaining genes in its chromosome.

Our algorithm begins by evaluating the initial population and then selecting the best individual, following one of three possible scenarios. In the first scenario, the individual is selected based purely on its cryptographic fitness. The second scenario is similar to the first, but if there are multiple individuals with optimal fitness, the algorithm picks between them based on their MCI. In the third scenario, the algorithm picks between the individuals primarily based on their fitness and uses their MCI as a secondary criterion throughout the whole evolutionary process. In all three scenarios, if there are multiple individuals with the same fitness and MCI, the newer individuals are picked over the old as a tertiary criterion.

If the selected individual is strictly better (not just newer) than the previous best individual, the algorithm sends its copy from the current core to all of its neighbors. Then it clears the rest of the old population and inserts any individuals received from the neighboring cores. Finally, the rest of the new population is filled by offspring created by applying the mutation operator on the selected parent. The algorithm uses a node-based, one-point mutation, meaning it selects a given number of nodes and randomizes both of their inputs and the trivial function at the same time.

We have examined other types of mutation as well, namely, a gene-based mutation that mutates either the trivial function of a node or one of its inputs, and universal mutation, that doesn't select a given number of nodes, but rather individually mutates each node with a small probability. However, neither of these mutation operators performed better than our one-point, node-based mutation.

If the mutation modifies any of the individual's active (output-affecting) nodes, it's truth table, relevant cryptographic criteria, fitness, and MCI are re-evaluated. The selection, communication, mutation, and evaluation then repeat until the algorithm either finds an individual with the desired fitness and MCI or after reaching a pre-defined number of fitness function evaluations.

We choose the island model of parallelization because without the use of a crossover operator there is little use for closer cooperation between the individual cores. On the contrary, having several smaller local populations actually increases the diversity of the global population, and may benefit the evolutionary process by helping it to overcome local optima. This is also the reason why we prefer new individuals over the old during selection. The infrequent (compared to the employer-worker model of parallelization) and asynchronous communication also allows the algorithm to maximize the time spent on computation.

The weakness of this model is that once a new best individual is found, it takes several generations for it to migrate to all of the other cores. However, because the number of generations required to find a strictly better individual is several magnitudes greater than the number of individuals being sent, and the maximum number of inefficient generations (i.e. $2\lceil\sqrt{n}-1\rceil$) is limited by our topology, this does not present a significant issue.

A. Fitness function

We define the fitness function separately for each of the Boolean functions and commonly for the benchmark circuits. For bent functions, the fitness score is defined by equation:

$$F_{bent} = NL * INP \tag{2}$$

where higher values mean better fitness. *INP* is a real value from 0 to 1 defined as the percentage of active (output-affecting) inputs. Cryptographic Boolean functions must consider all of their inputs to reach the optimal fitness. Penalizing those that do not use all of them helps to guide the evolutionary process and speeds up its convergence [1].

For balanced functions, we defined the fitness score by equation:

$$F_{bal} = NL * INP * BAL \tag{3}$$

where higher values mean better fitness. *BAL* is a real value from 0 to 1 defined as 1 if the function is balanced, and linearly decreasing from 0.5 to 0, the greater is the difference between the number of ones and zeroes in the function's truth table. This coefficient prevents nearly-balanced, highly-nonlinear functions (like bent functions) from dominating the evolutionary process.

Similarly, resilient functions aim to maximize the value of equation:

$$F_{res} = NL * INP * BAL * FineCI \tag{4}$$

where *FineCI* is a real value from 0 to 1 defined as the percentage of primary inputs for which the correlation immunity criterion is satisfied, and helps to guide the evolutionary process better than an integer value would.

For masking functions, the goal is to minimize the value of fitness defined by equation:

$$F_{mask} = HW + (3 - CoarseCI)^n \tag{5}$$

where *n* is the number of primary inputs, and *CoarseCI* is an integer value from 0 to 3. In this case we do not use a real value because computing the CI for each triplet of primary inputs individually is too computationally costly. To further speed up the process and avoid deceptive local optima, we penalize constant functions because while they have perfect CI, they are cryptographically useless. Moreover, we also penalize balanced functions, because while their CI can be extremely high (for example, the CI of a parity function is $n-1$), their HW is very poor, and their CI is therefore not worth evaluating.

For benchmark circuits, we use a fitness function defined simply as:

$$F_{bench} = DIF \tag{6}$$

where *DIF* is the Hamming distance between the individual's truth table and the truth table of the initial template. Because we seed the initial population of benchmark circuits with a working solution, this fitness ensures that the selected individual will always represent a working solution, and the evolutionary process will focus solely on optimizing its MCI.

V. Experiments

We conduct our experiments in four stages. First, we infer the MC of our Boolean functions, and experimentally determine what levels of cryptographic properties and MCI is our algorithm able to achieve for a varying number of function inputs. Second, we select a specific number of inputs, level of cryptographic properties, and MCI we can reliably achieve, and individually optimize the evolutionary parameters for each of these tasks. Third, we use the optimized parameters to examine how difficult it is to accomplish each of these tasks using the three different selection scenarios. Fourth, we use our algorithm to reduce the MCI of the selected benchmark circuits.

We perform all experiments on a computing cluster with two Intel Xeon E5-2680v3, 2.5 GHz, 12 core processors, 128GB DDR4@2133 MHz RAM, and InfiniBand FDR56 connection network. Topologically, we connect the cores into a 4-by-6 mesh.

A. Setup

We start with bent functions. Because each of the primary inputs of a Bent function must be processed in a nonlinear way, the MC of a bent function is a least $n/2$. We examine the bent functions of 6 to 16 inputs, and for each of them we manage to find a function with optimal NL, and MCI.

For balanced functions, the situation is more complicated because the optimal NL for functions with 8 or more inputs is unknown. We examine balanced functions with 6 to 16 inputs, and for functions of 6, 8, 10, and 14 inputs, we manage to find functions with NL equivalent to the best value known, and for 12 inputs with NL second to the best value known [30]. Because balanced functions are non-trivial, their MC for 8 or more inputs is also unknown. Because, to the best of our knowledge, this is the first work that examines the MCI of balanced Boolean functions, our results provide a novel upper bound for their MC.

For resilient functions, the situation is similar, and the optimal NL and MC for functions with 8 or more inputs is unknown. We examine resilient functions with 6 to 16 inputs. For all sizes, the best resilient functions we can find have MCI one lower than the bent functions with an equivalent number of inputs and two times higher linearity. This means that while our resilient functions generally do not achieve the best known NL [30], their MCI is optimal for the NL that they achieve.

With masking functions, we aim to find functions with a minimal (non-zero) HW and a CI of the third order, for which the optimal HW values are known for up to 13 inputs [31]. Because each AND operator can reduce the HW of a function at most by half, there is a known lower bound for their MC.

TABLE II
The best values of nonlinearity, Hamming weight, and multiplicative complexity of implementation, achieved by CGP during our preliminary experiments with the four types of cryptographic Boolean functions.

Inputs	Bent		Balanced		Resilient		Masking	
	NL	MCI	NL	MCI	NL	MCI	HW	MCI
6	28	3	26	5	24	2	16	1
8	120	4	116	6	112	3	16	3
10	496	5	492	10	480	4	32	4
12	2 016	6	2 008	11	1 984	5	32	6
14	8 128	7	8 120	16	8 064	6	–	–
16	32 640	8	32 512	7	32 512	7	–	–

For masking functions with 6 and 8 inputs, we find functions with the known optimal HW of 16. For masking functions with 10 and 12 inputs, we find functions with HW of 32, which is equivalent to the best HW ever found by evolutionary approaches [3], [7]. In all four cases, we find functions with MCI equal to the lower bound for their MC, proving that the lower and upper bound are the same, thus determining their MC.

Lastly, for the four benchmark circuits, we do not aim to achieve any cryptographic properties and focus solely on reducing their MCI while maintaining identical output as the original template. The MC of these non-trivial circuits is unknown.

The results of our first set of experiments are summed up in Table II. Based on these results, we decide to focus our second set of experiments on bent functions with 16 inputs and NL of 32640, balanced and resilient functions with 16 inputs and NL of 32512, and masking functions with 10 inputs and HW of 32.

B. Parameter Optimization

During the optimization, we focus on analyzing the impact of three main evolutionary parameters, the population size (λ), the number of columns (n_c), and the number of nodes selected during mutation (m_a). For each function, we perform a series of increasingly focused parameter sweeps, with 50 independent runs for each combination of population size, mutation rate, and the number of columns. After each sweep, we select the best values and repeat the experiment on a finer scale until the values stabilize. Because we do not have the space to show our results in completion, we illustrate them with Table III, which shows the median number of evaluations and runtime for each of the cryptographic Boolean functions, with $\lambda = 5$, $m_a = 10$, and variable n_c.

The results of this stage and the best performing parameters we found are summarized in Table IV. In all cases, small populations performed better than the large. Because our implementation of CGP does not perform any type of crossover which would benefit from a larger population, this result is not surprising. Also, because we're using the island model of parallelization with 24 islands and the population size is set per-island, the effective global population is not 5 but 120 individuals.

TABLE III
EXAMPLE OF RESULTS OF A PARAMETER SWEEP TO FIND THE BEST-PERFORMING NUMBER OF COLUMNS FOR EACH OF THE FOUR CRYPTOGRAPHIC FUNCTIONS ($\lambda = 5, m_a = 10$).

Function	Property	Number of columns (n_c)					
		50	100	200	400	800	1600
Bent	runtime [s]	47.22	5.33	4.367	4.08	10.96	39.96
	evaluations	454 848	34 572	17 064	8 940	12 744	24 360
Balanced	runtime [s]	10.48	2.66	1.19	1.65	2.65	6.33
	evaluations	98 664	16 752	4 512	3 480	3 000	3 768
Resilient	runtime [s]	25.16	28.01	8.84	7.10	9.74	12.98
	evaluations	225 252	23 760	6 816	4 680	5 124	4 788
Masking	runtime [s]	100.37	45.58	28.118	27.73	50.104	129.22
	evaluations	2 065 200	1 448 352	70 992	522 132	606 888	773 904

TABLE IV
BEST PERFORMING CGP PARAMETERS OPTIMIZED FOR EACH CRYPTOGRAPHIC FUNCTION AND BENCHMARK CIRCUIT.

Function type	Population size (λ)	Mutation rate (m_a)	Number of columns (n_c)
Bent	5	10	400
Balanced	5	10	450
Resilient	5	10	500
Masking	5	10	200
Benchmark	5	1	min
Examined range	5-100	1-50	50-2000

Changes in mutation rate had a negligible effect, with any value from 5 to 30 performing approximately equally well for all cryptographic functions, with a slight bias towards a higher number of mutations if longer chromosomes were utilized. We assumed this is because most nodes inside the individual usually remain inactive, meaning that a low mutation rate can fail to make meaningful changes, while an extremely high mutation rate can become destructive for the functioning part of the chromosome and lead towards a random search.

For benchmark circuits, minimal mutation rates performed the best, with mutation rates higher than three becoming completely ineffective. We assumed this is because the benchmark circuits started with a fully working solution that takes up most (if not all) of the nodes of the individual. Mutations therefore always happen on active nodes, resulting in a new and distinct circuit.

For cryptographic Boolean functions the best results were obtained with a reasonable number of nodes in the low hundreds, with the exact value slightly dependent on the complexity of the function and strongly dependent on the number of primary inputs. If the node count is too low the evolution stalls, requiring an increasing number of evaluations to create an improvement. If the node count is too high, there is no additional reduction in the number of evaluations, but each individual takes an increased amount of time to evaluate.

For benchmark circuits, the minimal number of nodes is limited by the size of the initial working solution. In all cases, this minimal value always performed the best, with higher node counts performing slightly worse both in terms of evaluation count and processing time.

C. Cryptographic functions

For each Boolean function, we perform experiments using the three selection scenarios. In the first scenario, we consider only the cryptographic properties of each function while ignoring its MCI, and the evolution ends as soon as we find a function with the desired fitness. We denote this scenario as *(Fit, 0)* and use it as our baseline. In the second scenario, we first search for a function with the desired fitness, then optimize its MCI, and the evolution ends only when both the desired fitness and MCI have been reached. We denote it as *(Fit, MC)*.

In our third scenario, we optimize both Fitness and MCI at the same time by using fitness as the primary, and MCI as the secondary selection criterion throughout the entire evolutionary process. The evolution also ends only when both the desired fitness and MCI have been reached, and we denote it as *(Fit + MC)*.

For each of these scenarios, we perform 100 independent runs and measure the number of fitness function evaluations necessary to find the desired function, its MCI, the number of active nodes in the chromosome, and runtime.

Because the outputs of EAs generally do not follow a normal distribution and a single stalled run can wildly influence the average, we measure the median (Q2) which, in our opinion, is a better indicator of the expected performance of the algorithm. To provide information about the spread of obtained values we also provide the first (Q1) and third (Q3) quartile of each experiment, all presented in Table V.

Our results show that for bent, balanced, and resilient functions it is significantly easier to optimize both fitness and MCI at the same time than it is to first find a function with good fitness and then try to improve its MCI. For bent function, co-optimizing both parameters at the same time is so easy that the MCI-optimal functions do not take significantly longer to find than functions optimized purely for their fitness. Furthermore, decreasing the MCI of Boolean functions does not lead to an increase in the overall number of active nodes, and in some cases even helps to decrease it.

For masking functions, however, we observe the opposite effect. Including the MCI in the evolutionary process more than doubles the evolution time, while finding a function with optimal fitness and then optimizing its MCI does not take

TABLE V
RESULTS OF EVOLVING THE FOUR TYPES OF CRYPTOGRAPHIC BOOLEAN FUNCTIONS WITH CGP, USING THE BEST PERFORMING PARAMETERS AND ALL THREE SCENARIOS.

	Scenario	Evaluations			MCI			Columns (n_c)			Runtime [s]		
		Q1	Q2	Q3	Q1	Q2	Q3	Q1	Q2	Q3	Q1	Q2	Q3
Bent	(Fit, 0)	5 322	8 280	19 230	11	13	18	35	40	45	2.47	3.74	8.71
	(Fit, MC)	7 410	12 588	21 648	8	8	8	33	36	40	3.38	5.70	9.75
	(Fit + MC)	5 418	8 640	15 054	8	8	8	32	36	40	2.51	3.93	6.76
Balan.	(Fit, 0)	2 514	3 216	5 364	11	14	17	32	40	44	1.32	1.69	2.73
	(Fit, MC)	4 134	5 928	11 670	7	7	7	30	34	38	2.12	3.04	5.91
	(Fit + MC)	3 492	4 548	8 364	7	7	7	29	33	39	1.80	2.35	4.24
Resil.	(Fit, 0)	2 778	4 392	10 806	10	12	14	31	36	41	4.54	7.13	17.20
	(Fit, MC)	4 590	7 452	15 930	7	7	7	29	33	38	7.35	11.87	25.08
	(Fit + MC)	3 936	6 168	11 184	7	7	7	27	32	36	6.29	9.82	17.64
Mask.	(Fit, 0)	322 920	623 400	860 487	4	5	7	19	21	24	13.51	23.55	34.16
	(Fit, MC)	401 364	610 920	1 023 462	4	4	4	19	20	22	16.72	24.04	40.16
	(Fit + MC)	804 078	1 514 760	3 624 264	4	4	4	18	20	21	30.89	56.28	131.38

TABLE VI
IMPROVEMENTS IN MCI OF THE BENCHMARK CIRCUITS AFTER 1 MILLION AND 10 MILLION EVALUATIONS OF THE CGP ALGORITHM.

Benchmark circuit	Initial MC	1M evaluations			10M evaluations		
		Q1	Q2	Q3	Q1	Q2	Q3
Cavlc	704	586	582	577	498	492	489
Controller	199	85	82	79	56	54	52
Decoder	304	299	298	298	295	294	294
Int2float	267	190	186	181	139	137	131

TABLE VII
COMPARISON OF THE BEST RESULTS OF MCI OPTIMIZATION PROVIDED BY THE ECA, LST, AND OUR CGP ALGORITHM, AND THE NUMBER OF EVALUATIONS NEEDED TO FIND THIS VALUE.

Benchmark circuit	Logic synthesis [9], [26]			CGP		
	initial	ECA	LST	initial	best	evaluations
Cavlc	536	494	394	704	387	131 568 624
Controller	86	85	45	199	39	661 846 896
Decoder	341	341	328	304	292	46 166 832
Int2float	133	100	85	267	92	175 548 528

significantly longer than finding the optimal fitness alone. We assume that this is partially caused by the fact that, even with their MCI not optimized, the functions were often already optimal in this criterion as well, as Table V also shows.

D. Benchmark circuits

When experimenting with the benchmark circuits, we take the initial function as provided by the benchmark suite [10] and convert its representation from VHDL into our CGP implementation. In all cases, the initial circuit consisted almost entirely out of AND gates, and we did not perform any kind of manual optimization.

Because the MC of these non-trivial functions is unknown, we decide not to set a specific goal for our MCI optimization, but rather to observe how its value is reduced after a certain number of evaluations. For each circuit, we perform 100 independent runs limited to 1 million and 10 million evaluations. For each experiment, we provide the values of the first, second and third quartile as seen in Table VI.

To truly test the limits of our algorithm we perform five extra runs for each circuit, limited not by a given number of evaluations but only by our available resources, which are capped at 1 hour of cluster runtime for each run. For these experiments, we provide the best MCI reached and the number of evaluations necessary to find it. We also compare our results to those provided by the *enumeration cut algorithm* and *logic synthesis toolbox* presented in literature [9], [26]. Because our CGP uses a different representation, the initial MCI is different, but functionally, these are indeed the same circuits and their (unknown) MC is the same. These results are presented in Table VII.

Our results show that our approach is capable of steadily improving the MCI of a given circuit even past the 10 million evaluations mark, and, if given enough time, outperforms the competing approaches on most examined circuits. Note that the results provided by both the ECA and LST were gained by repeating their method until convergence was reached, and therefore would not benefit from any further increase in computational resources. Meanwhile, our method was still finding better solutions and given enough time it could likely optimize the circuits even further.

On the other hand, our method relies on the heuristic search and repeated evaluation of the evolved functions, making it unsuitable for improving circuits with more than 20 inputs, while ECA and LST work directly on the graph representation of the circuit, allowing one to improve even very large circuits.

VI. CONCLUSION AND FUTURE WORK

In this paper, we have explained the importance of the multiplicative complexity of functions used in high-level cryptographic protocols. We have selected four types of cryptographic Boolean functions: bent, balanced, resilient, and masking. We detailed their cryptographic properties, possible uses, and tried to improve the multiplicative complexity of their implementation.

To do this, we have implemented a parallel CGP algorithm, optimized its evolutionary parameters, and performed several sets of experiments on functions with up to 16 inputs. All examined functions preferred a very small population size,

node count in the low hundreds, and a moderate mutation rate. We have shown that for bent, balanced, and resilient functions, it is preferable to search for the best cryptographic properties and MCI at the same time, while for masking functions, it is better to first find a function with the desired properties, and only then try to improve its MCI.

We have shown that for bent and masking functions with 16 inputs it is easy to find functions with optimal MCI, provided an upper bound for MC of some of the best balanced functions known to exist. And to the best of our knowledge, this is the first work that has examined these types of Boolean functions under the MC or MCI criteria. To allow a comparison between our method and other state-of-the-art optimization approaches, we have also optimized MCI of several combinational circuits from the EPFL benchmark suite. We have shown that our method is competitive, and for circuits with a low number of inputs, it outperforms the other approaches.

There are several ways our work could be expanded upon with further research. For example, by broadening its scope and including Boolean functions with other cryptographic properties like algebraic degree and algebraic immunity, or further exploring the relationship and necessary trade-off between the nonlinearity and MCI for other sizes of input. Another interesting venue would be to compare the performance of CGP with regular, tree-based GP or LGP, which have already been shown to provide great results when designing many types of cryptographic Boolean functions. Or compare it with other state-of-the-art approaches for designing Boolean functions, like *learning classifier systems*.

REFERENCES

[1] J. Husa and R. Dobai, "Designing bent boolean functions with parallelized linear genetic programming," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. ACM, 2017, pp. 1825–1832.

[2] J. Husa, "Comparison of genetic programming methods on design of cryptographic boolean functions," in *European Conference on Genetic Programming*. Springer, 2019, pp. 228–244.

[3] S. Picek, S. Guilley, C. Carlet, D. Jakobovic, and J. F. Miller, "Evolutionary approach for finding correlation immune boolean functions of order t with minimal hamming weight," in *International Conference on Theory and Practice of Natural Computing*. Springer, 2015, pp. 71–82.

[4] J. Husa, "Designing correlation immune boolean functions with minimal hamming weight using various genetic programming methods," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. ACM, 2019, pp. 342–343.

[5] J. McLaughlin and J. A. Clark, "Evolving balanced boolean functions with optimal resistance to algebraic and fast algebraic attacks, maximal algebraic degree, and very high nonlinearity." *IACR Cryptology ePrint Archive*, vol. 2013, p. 11, 2013.

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

[7] S. Picek, C. Carlet, S. Guilley, J. F. Miller, and D. Jakobovic, "Evolutionary algorithms for boolean functions in diverse domains of cryptography," *Evolutionary computation*, vol. 24, no. 4, pp. 667–694, 2016.

[8] J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*. MIT press, 1992, vol. 1.

[9] E. Testa, M. Soeken, L. Amarù, and G. De Micheli, "Reducing the multiplicative complexity in logic networks for cryptography and security applications," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6.

[10] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "The EPFL combinational benchmark suite," in *Proceedings of the 24th International Workshop on Logic & Synthesis (IWLS)*, 2015.

[11] C. Carlet, Y. Crama, and P. L. Hammer, "Boolean functions for cryptography and error correcting codes," *Boolean models and methods in mathematics, computer science, and engineering*, vol. 2, pp. 257–397, 2010.

[12] A. Braeken, "Cryptographic properties of boolean functions and s-boxes," Ph.D. dissertation, phd thesis-2006, 2006.

[13] J. Boyar, R. Peralta, and D. Pochuev, "On the multiplicative complexity of boolean functions over the basis (*,+,1)," *Theoretical Computer Science*, vol. 235, no. 1, pp. 43–57, 2000.

[14] M. T. Sönmez and R. Peralta, "The multiplicative complexity of boolean functions on four and five variables," in *International Workshop on Lightweight Cryptography for Security and Privacy*. Springer, 2014, pp. 21–33.

[15] Ç. Çalık, M. S. Turan, and R. Peralta, "The multiplicative complexity of 6-variable boolean functions," *Cryptography and Communications*, vol. 11, no. 1, pp. 93–107, 2019.

[16] M. R. Albrecht, C. Rechberger, T. Schneider, T. Tiessen, and M. Zohner, "Ciphers for mpc and fhe," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2015, pp. 430–454.

[17] M. Chase, D. Derler, S. Goldfeder, C. Orlandi, S. Ramacher, C. Rechberger, D. Slamanig, and G. Zaverucha, "Post-quantum zero-knowledge and signatures from symmetric-key primitives," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 1825–1842.

[18] I. Giacomelli, J. Madsen, and C. Orlandi, "Zkboo: Faster zero-knowledge for boolean circuits," in *25th {usenix} security symposium ({usenix} security 16)*, 2016, pp. 1069–1083.

[19] V. Kolesnikov and T. Schneider, "Improved garbled circuit: Free xor gates and applications," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2008, pp. 486–498.

[20] W. Millan, A. Clark, and E. Dawson, "An effective genetic algorithm for finding highly nonlinear boolean functions," in *International Conference on Information and Communications Security*. Springer, 1997, pp. 149–158.

[21] S. Picek, C. Carlet, D. Jakobovic, J. F. Miller, and L. Batina, "Correlation immunity of boolean functions: an evolutionary algorithms perspective," in *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. ACM, 2015, pp. 1095–1102.

[22] S. Picek, D. Jakobovic, and M. Golub, "Evolving cryptographically sound boolean functions," in *Proceedings of the 15th annual conference companion on Genetic and evolutionary computation*. ACM, 2013, pp. 191–192.

[23] R. Hrbacek and V. Dvorak, "Bent function synthesis by means of cartesian genetic programming," in *International Conference on Parallel Problem Solving from Nature*. Springer, 2014, pp. 414–423.

[24] J. Boyar, P. Matthews, and R. Peralta, "Logic minimization techniques with applications to cryptology," *Journal of Cryptology*, vol. 26, no. 2, pp. 280–312, 2013.

[25] N. Courtois, T. Mourouzis, and D. Hulme, "Exact logic minimization and multiplicative complexity of concrete algebraic and cryptographic circuits," *Int. J. Adv. Intell. Syst*, vol. 6, no. 3, pp. 165–176, 2013.

[26] E. Testa, M. Soeken, H. Riener, L. Amaru, and G. De Micheli, "A logic synthesis toolbox for reducing the multiplicative complexity in logic networks."

[27] J. F. Miller and P. Thomson, "Cartesian genetic programming," in *European Conference on Genetic Programming*. Springer, 2000, pp. 121–132.

[28] R. Kalkreuth, G. Rudolph, and A. Droschinsky, "A new subgraph crossover for cartesian genetic programming," in *European Conference on Genetic Programming*. Springer, 2017, pp. 294–310.

[29] J. Husa and R. Kalkreuth, "A comparative study on crossover in cartesian genetic programming," in *European Conference on Genetic Programming*. Springer, 2018, pp. 203–219.

[30] X. Tang, D. Tang, X. Zeng, and L. Hu, "Balanced boolean functions with (almost) optimal algebraic immunity and very high nonlinearity." *IACR Cryptology ePrint Archive*, vol. 2010, p. 443, 2010.

[31] C. Carlet and X. Chen, "Constructing low-weight d th-order correlation-immune boolean functions through the fourier-hadamard transform," *IEEE Transactions on Information Theory*, vol. 64, no. 4, pp. 2969–2978, 2017.