Exploring Lightweight S-boxes Using Cellular Automata and Reinforcement Learning

Tarun Ayyagari, Anirudh Saji(✉), Anita John, and Jimmy Jose

National Institute of Technology Calicut, Kozhikode, India
{tarun_b180682cs,anirudh_b180387cs,anita_p170007cs,jimmy}@nitc.ac.in

Abstract. The most important elements of block ciphers are nonlinear functions known as substitution boxes (S-boxes). S-boxes with weak cryptographic properties are vulnerable to attacks by various cryptanalysis techniques. Cellular Automata has been used to design S-boxes with good cryptographic properties such as nonlinearity, differential uniformity, balancedness, etc. Cellular Automata based S-boxes also have low implementation cost due to their highly parallel nature. In this work, we explore an approach of using Cellular Automata based semi-bent Boolean functions to generate S-boxes. Genetic algorithms have been used extensively to generate CA based S-boxes. Here we explore the use of Reinforcement Learning algorithms that uses relatively well understood and mathematically grounded framework of Markov Decision Processes as an alternative to genetic programming.

Keywords: Lightweight S-boxes · Semi-bent Boolean functions · Cellular Automata · Reinforcement Learning · Block ciphers

1 Introduction

Cellular Automata (CA) have been proved to be quite useful in the field of cryptography, widely being used as keystream generators for stream ciphers due to their good pseudo-randomness. CAs have also been proved to be very useful in creating semi-bent functions with good cryptographic properties as proven in [1]. In this work, we expand on the work done by Mariot et al. in [1] and consider the different permutations of Boolean functions. Further, we implement these Boolean functions as the coordinate functions of an S-box. For finding a suitable subset of Boolean functions to use as coordinate functions, we will use Reinforcement Learning. By the end, we evolve a good set of functions which would result in an S-box with good cryptographic properties such as nonlinearity (NL) and differential uniformity (DU). In previous works [2], we have seen designs using genetic programming to create the S-boxes. Genetic programming is largely based on heuristics. Genetic programming's end goal is to evolve an unfit population of elements using various genetics inspired functions. Genetic programming has adapted concepts of crossover and mutation from genetics and

B. Chopard et al. (Eds.): ACRI 2022, LNCS 13402, pp. 17–28, 2022.
https://doi.org/10.1007/978-3-031-14926-9_2

implemented them in code. The crossover operation involves swapping random parts of selected pairs of parents (elements from the previous population) to produce new and different offspring that become part of the new generation. Mutation involves substitution of some random part of an element with some other random part of an element. Both these crossover and mutation functions are used on populations in the hope that the next population would create stronger off-springs (elements). This process will continue until the desired level of fitness is observed in a population. Our design involves using Reinforcement Learning (RL) instead of genetic programming to select the set of semi-bent Boolean functions that will be used to generate the S-boxes. Both Genetic Programming and Reinforcement Learning aim to maximize a defined reward signal. We aim to experiment with RL as it is based on the mathematically grounded framework of Markov Decision Processes (MDPs) and on initial analysis, can be seen to even speed up the convergence process of finding the set of functions that produce the strongest S-boxes. Reinforcement Learning is used to systematically explore the solution space to find the permutation of semi-bent Boolean functions which has the strongest cryptographic properties.

2 Cellular Automata

CA has been widely researched because of their low implementation cost and parallel computing nature. These properties of CA make them excellent high bandwidth cryptographic application solutions. A CA is a system of finite state automata (cells) which are arranged in a grid. The state of a cell at a timestep depends on the cell's state as well as the state of the cell's neighbours. Each cell has a local update rule which determines the state of the cell at the next timestep. The states of a cell in a CA are generally from the set {0,1}. In context of CAs, we consider both time and space to be discrete. In each timestep, each cell in the CA is updated according to the local rule of the cell. A local rule can be represented as a function

$$f : S^m \rightarrow S$$

where m represents the neighbourhood size considered in the update rule. For a 1D CA (where the cells are arranged in a 1-dimensional array), a neighbourhood size of m indicates that we consider $2m + 1$ states in the local update rule (m on each side and the state itself). For a 2D CA (where the cells are arranged as a 2-dimensional array), a neighbourhood of m considers $4m + 1$ states for the update rule (m on each side and the state itself). At the ends of the CA, the neighbourhood wraps around to the other end of the CA. Theoretically, a CA can be represented in any number of dimensions [3] but increasing the dimensionality also increases the computational cost for the CA. CA are mathematically complete and lightweight in most of their implementations, thus making them suitable for a wide range of applications. In our application, we will only be considering a 1D CA with a neighbourhood size of 1. The output for each combination can be succinctly represented by a single number called the CA's Wolfram Number [5]. An important aspect of the CA rule is that each CA rule

can be represented in their *algebraic normal form (ANF)* in terms of the states of the cells considered. The ANF of a Boolean function f can be represented as

$$f(x_1, \ldots, x_n) = \bigoplus_{I \in 2^{[n]}} a_I x^I \tag{1}$$

where, $x^I = \prod_{i \in I} x_i$, $2^{[n]}$ is the power set of $[n] = \{1, 2, ..., n\}$ and $a_I = 0$ or 1. This is how the rules will be represented in code. The algebraic degree of the Boolean function f is the cardinality of the largest subset $I \in 2^{[n]}$ in its ANF, such that its coefficient $a_I \neq 0$.

3 Substitution Boxes (S-boxes)

S-boxes are an integral part of many encryption systems. An S-box can be represented as

$$F : \mathbb{F}_2^m \to \mathbb{F}_2^n$$

where n,m are two positive integers and $\mathbb{F}_2$ is the Galois Field of two elements. S-boxes are also referred to as (n, m) functions, where n and m correspond to number of inputs and outputs of the S-box respectively. The function F is also called a vectorial Boolean function. Function F can be decomposed into the vector $F = (f_1, f_2, ..., f_m)$ where each function f_i is a Boolean function $f_i : \mathbb{F}_2^n \to \mathbb{F}_2 \forall i$. The functions $f_i \; \forall \; i \in \{1, 2, ..., m\}$ are called the coordinate functions of S-boxes function F. Any non-trivial linear combination of the coordinate functions is called a component function of F. For an S-box to be cryptographically strong, there are a number of properties it must satisfy [7].

3.1 Nonlinearity

The *Walsh-Hadamard Transform* W_F of (n, m) function F is given by

$$W_F(u, v) = \sum_{x \in \mathbb{F}_2^n} (-1)^{v \cdot F(x) \oplus u \cdot x}, v \in \mathbb{F}_2^m, u \in \mathbb{F}_2^n \tag{2}$$

The nonlinearity N_F of function F is given by the equation:

$$N_F = 2^{n-1} - \frac{1}{2} \max_{u \in \mathbb{F}_2^n, v \in (\mathbb{F}_2^m)^*} |W_F(u, v)| \tag{3}$$

where $(\mathbb{F}_2^m)^* = \mathbb{F}_2^m \backslash \{0\}$. We aim to achieve maximum nonlinearity, i.e., reduce the linearity between the function F and its component functions. A high value for nonlinearity will make it harder to perform linear cryptanalysis on the S-boxes.

3.2 Differential Uniformity

For a given (n, m) function F, we can define a *difference distribution table* D_F as

$$D_F(a, b) = \{x \in \mathbb{F}_2^n : F(x) \oplus F(x \oplus a) = b\},\ a \in \mathbb{F}_2^n,\ b \in \mathbb{F}_2^m$$

The value at (a,b) represents the cardinality of $D_F(a, b)$ denoted by $\delta(a, b)$. The differential uniformity of the function F, δ_F is given by

$$\delta_F = \max_{a \neq 0, b} \delta(a, b) \tag{4}$$

We aim to minimize the differential uniformity of an S-box. A low value of δ_F implies that the S-box can withstand differential cryptanalysis. The minimum attainable value for differential uniformity is 2 and S-boxes which achieve this value are called *almost perfect nonlinear (APN) functions.*

4 Semi-bent Boolean Functions

Consider a Boolean function *f*. From Eq. (3), it is evident that the maximum value of N_f is achieved when the max term in the equation evaluates to $2^{\frac{n}{2}}$, resulting in the bound: $N_f \leq 2^{n-1} - 2^{\frac{n}{2}-1}$. Functions that satisfy this bound are known as *bent functions*, but these functions only exist for even values of n [8]. Unfortunately, bent functions are not balanced, so they cannot be considered for use in cryptographic systems. A Boolean function is balanced if its truth table has equal number of 0's and 1's in its output, i.e., for an arbitrary input, it is equally likely to get a 0 or 1 as the output. The truth table of a Boolean function is the mapping of the input bits to the output bits for that Boolean function. Every Boolean function can be represented as a truth table. The quadratic bound for when n is odd is $N_F \leq 2^{n-1} - 2^{\frac{n+1}{2}-1}$. Any function of algebraic degree 2 can achieve this bound. In general, this bound is not tight when n is odd and $n > 7$. It is still an open problem to determine the true upper bound on the nonlinearity for that case. The Walsh transform for a Boolean function $f : \mathbb{F}_2^m \to \mathbb{F}_2$ is given by the equation

$$W_f(u) = \sum_{x \in \mathbb{F}_2^m} (-1)^{f(x) \oplus u \cdot x},\ \forall u \in \mathbb{F}_2^m \tag{5}$$

The Walsh transform of a Boolean function measures the correlation between the function f and the linear function $u \cdot x$. It is therefore, used to calculate the nonlinearity of a Boolean function f. Semi-bent Boolean functions are Boolean functions whose Walsh transform can be defined as:

$$W_f(u) = \begin{cases} 2^{\frac{n+1}{2}} & if\ n\ is\ odd, \\ 2^{\frac{n+2}{2}} & if\ n\ is\ even. \end{cases} \tag{6}$$

These functions reach the bound on nonlinearity when n is odd. It is possible for these functions to be balanced, so we shall be considering these to use as

coordinate functions in our construction of S-boxes. A Boolean function is balanced if its truth table has equal number of 0's and 1's in its output, i.e., for an arbitrary input, it is equally likely to get a 0 or 1 as the output.

5 Reinforcement Learning

Reinforcement Learning is the area of machine learning that deals with how intelligent agents interact within an environment to maximize a cumulative reward. RL is considered to be the 3rd machine learning paradigm alongside supervised learning and unsupervised learning and is sometimes semi-supervised in nature. The learner and decision maker is called the agent. The thing it interacts with, comprising everything outside the agent, is called the environment. RL agents interact with the environment, which can be classified as a set of states that can be both continuous or discrete, using a set of pre-defined actions. Each action in each state, also known as a state-action pair is associated with a reward signal. The goal of the agent is to maximize the cumulative sum of the reward signals. It does so by exploring the actions it has never taken before and exploiting the actions that have been taken and the agent has prior knowledge about. In almost all RL problems, there exists an exploration-exploitation dilemma. The dilemma is that the agent has to exploit what it already knows to obtain reward but the agent also has to explore in order to make better selections in the future. Generally, on knowing what actions are optimal in a state, the agent still chooses sub-optimal actions once in a while, in the hopes to achieve a greater cumulative sum of rewards by choosing a different sequence of actions and states. Apart from the actions and states, an RL problem has 4 more sub-elements: a policy, a reward signal, a value function and optionally a model of the environment [4]. The policy defines how the agent behaves in a state and what actions it chooses. In their book, Richard Sutton and Andrew Barto define a policy as a mapping from states to probabilities of selecting each possible action. A reward signal defines the goal of a reinforcement learning problem. The value function or the value of a state is the expectation of total reward it will accumulate over time. There also exists state action values, which is, the value of taking a particular action from a particular state. The reward signal, takes into account what is good as the immediate next step, whereas the value function is far sighted and looks into the total reward accumulated in the future. The last element is the model. This is something that mimics the behavior of the environment, or more generally, that allows inferences to be made about how the environment will behave. RL problems are usually formulated as a Markov Decision Processes. In [4], MDPs are defined as a mathematically idealized form of the reinforcement learning problem for which precise theoretical statements can be made. For a finite MDP, the states, actions and rewards have a well defined discrete probability. That is,

$$\sum_{s' \in S} \sum_{r \in \mathcal{R}} p\left(s', r \mid s, a\right) = 1, \forall s \in \mathcal{S}, a \in \mathcal{A}(s) \ , \tag{7}$$

where $\mathcal{S}$ is the set of all states, $\mathcal{A}(s)$ is the set of all actions available at the state s and r is the reward received when after transitioning to state s' from the state s on taking action a. In an MDP, the probabilities given by p completely characterize the environment's dynamics. That is, the probability of each possible value for S_t and R_t depends only on the immediately preceding state and action, S_{t-1} and A_{t-1}, and, given them, not at all on earlier states and actions [4]. The RL agent together with its policy and state action pairs make decisions to explore the environment, learning and exploiting data learnt through positive and negative reinforcements to maximize the reward signal.

6 Our Design

Our goal is to build an S-box with excellent cryptographic properties, i.e., high nonlinearity and low differential uniformity. To implement our goal, it has been formulated as a 3-part problem. The three parts include Boolean Functions, Substitution Box and Reinforcement Learning (Fig. 1 and 2).

6.1 Boolean Functions

Our design allows us to use a set of 2 or more semi-bent Boolean functions to generate the output array. We consider the CA as the input bits to the S-boxes. The CA length is 8 cells long, the state of each cell is given by the corresponding input bits. We consider the CA to have a periodic boundary, i.e., the neighbourhoods for the edge cells wrap around to the other end of the CA. In this work, we consider set sizes of 2 and 3 semi-bent Boolean functions to generate the output array from the set of input bits. Each Boolean function involves a set of 2 operations. The first operation is the application of a CA rule on the set of 8 input bits. After first step, an intermediary array is created whose bits are then XORed to get one bit. This is the second step. The size of this intermediary array depends on the size of the neighbourhood of the CA rule (Boolean function). In our design, the size of the intermediary array is always 6, as we are only work with CA rules of neighbourhood size 3 for both the set sizes of rules. Each intermediary array produces only one of the output bits. Hence, the process of creating the intermediary array is iterated 8 times in order to get the 8 output bits. During each of these iterations, the neighbourhood cells do not overlap with the initial cell. Hence, during each iteration to produce one output bit, it can be said that the boundary is fixed at 8 cells starting from the cell indexed at *start* and ending at the cell indexed at $(start + nbr\_size)\%8$. Here, it is to be remarked that for periodic boundary conditions, the constructions from [1] does not work as it is not possible to prove that the resulting Boolean function has the same degree as the local rule [9]. The pseudocode for this has been given below in Algorithm 1. *rule* is the CA rule which we will be applying on the input. *nbr_size* is the neighbourhood size on which the CA rule is applied. *len* is the size of the input CA (here 8). *start* indicates the neighbourhood offset. 2 steps together make the semi-bent function. Semi-bent functions are known for

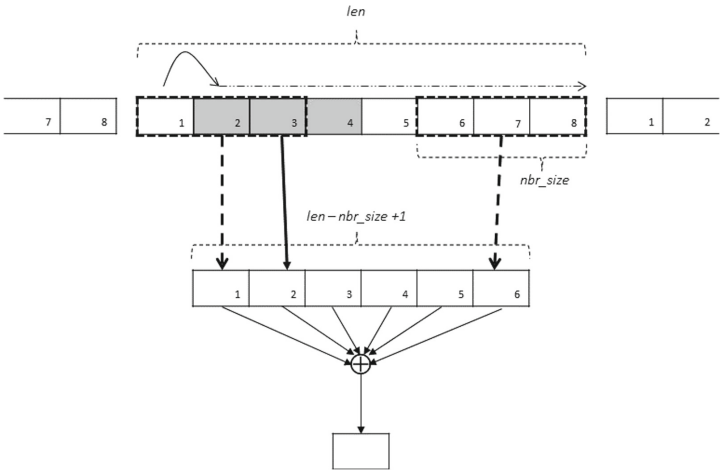

Fig. 1. First iteration of Algorithm 1 applying a CA rule of neighbourhood size nbr_size to a set of bits of length len generating $len - nbr\_size + 1$ output bits which are further XORed to get a single bit. Here, $start = 1$, and in our design $nbr\_size = 3$ bits and $len = 8$ bits.

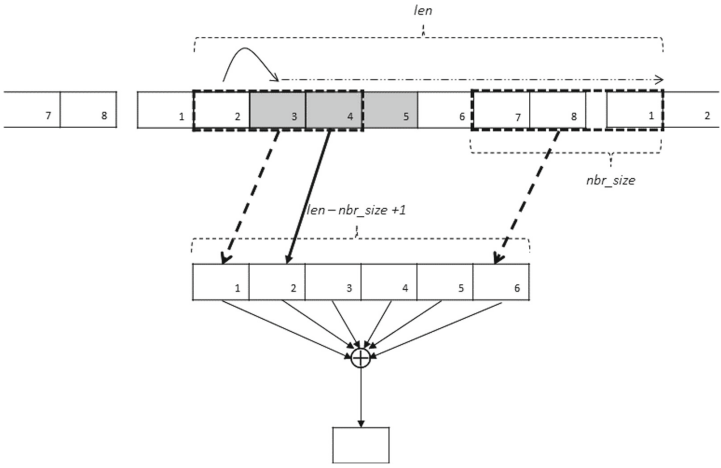

Fig. 2. Second iteration of Algorithm 1 applying a CA rule of neighbourhood size nbr_size to a set of bits of length len generating $len - nbr\_size + 1$ output bits which are further XORed to get a single bit. Here, $start = 2$, and in our design $nbr\_size = 3$ bits and $len = 8$ bits.

their high cryptographic standard, exhibiting properties of high nonlinearity and low differential uniformity. Mariot et al. [1] discovered several CA based semi-bent functions using varying neighbourhood sizes. For our use, we narrowed down to the 56 CA rules of neighbourhood size 3. We will use these rules in the construction of the S-boxes.

Algorithm 1

```
function RULE_OP(rule,nbr_size,len,start)
    outputs ← ∅
    i ← start
    max_iter ← start + len − nbr_size + 1
    while i < max_iter do
        nbr ← ∅
        j ← 0
        while j < nbr_size do
            nbr.Append(CA[(i + j)%len])
        end while
        outputs.Append(rule(nbr))
        i ← i + 1
    end while
    return ⊕_{bit∈outputs} bit
end function
```

6.2 Substitution Box

Our design is implemented as a CA consisting of 8 cells. The CA rules are semi-bent functions as discussed in the previous section. In this work, we discuss 2 designs of S-boxes. In the first design, we use 2 CA rules, each generating 4 bits of the final output. In the second design, we use 3 CA rules, 2 of which generate 3 bits towards the final output and the last one generating only 2. The S-box pseudocode has been described in Algorithm 2. The S-box takes in the set of CA rules as parameter. *input_size* is the length of the CA (8 in our case) and *num_rules* represents the number of rules we are using in the design.

6.3 Reinforcement Learning

In this work, we will be using Reinforcement Learning to find suitable sets of semi-bent functions to use in our S-box design. To solve a problem using RL, it first has to be formulated as a Markov Decision Process. So we start by identifying our states, rewards and actions with respect to our design. We define the state space as all possible k-permutations of all semi-bent functions, where $k \in \{2, 3\}$. The number of permutations of k items from n objects is given by

$$^{n}P_{k} = \frac{n!}{(n-k)!} \tag{8}$$

where k is the number of rules selected and n is the total number of rules. The state space is discrete. Each permutation of the set of rules can be considered as a different state and hence, each state varies by at least one semi-bent function. When considering 2 semi-bent functions, the total space consists of $^{56}P_2 = 3080$ different states and in the 3 semi-bent functions design, we have $^{56}P_3 = 166320$ states. The set of actions are also discrete and can be considered as the swapping

Algorithm 2

```
function SBOX(rules)
    outputs ← ∅
    for each possible input ip do
        output ← ∅
        start ← 0
        k ← ceil(input_size/num_rules)
        for all rule in rules do
            i ← 0
            while i < k do
                op ← RULE_OP(rule, 3, input_size, ip, start)
                output.Append(op)
                start ← start + 1
                i ← i + 1
            end while
        end for
        outputs.Append(output[0 : 7]) ▷ Only the first 8 bits of the output are taken
    end for
    return outputs
end function
```

of a rule for another rule. The reward for transitioning from one state to another is the cryptographic strength of the latter state given as

$$strength = (scaled\_NL + (100 - scaled\_DU))/2 \tag{9}$$

where *strength* is the cryptographic strength of the current set of rules in the S-box. The scaled nonlinearity of the S-box denoted by *scaled_NL* is

$$scaled\_NL = (NL/112) * 100 \tag{10}$$

scaled_DU is the scaled differential uniformity of the S-box.

$$scaled\_DU = ((DU - 4)/(128 - 4)) * 100 \tag{11}$$

The scaling was done with respect to the values attained by the AES S-box [6]. The policy is chosen to be epsilon greedy, that is, with epsilon probability, the agent chooses a non greedy action. We talk about greedy/non-greedy with respect to the calculated state value or the state action value. We use the on-policy Sarsa algorithm [4] to calculate the state action value pairs from the information gathered during exploration. We use the concept of average reward in our Sarsa algorithm [4]. This is used in our problem as our problem deals with a continuous task. We chose to use the value-function approximation method [4], given the large state spaces. An Artificial Neural Network (ANN) was chosen to be the function approximator given the nonlinear relationship between the rules used and the strength of the state. The function approximator is used to approximate the value of a given state, given the input parameters from a given state. In our design, we give the rules used in that particular state as the

parameters so as to calculate the value of the state using the rules used in that state. For each rule, the input array to the ANN is flattened, so as to achieve a distinct array for each distinct permutation of rules. In the 2 rule design, there are 2 positions for the rules and 56 rules can be inserted in each space. Hence the size of the input array to the ANN will be a binary array of length $2 * 56 = 112$.

We define the reward for transitioning from state s_1 to state s_2 as

$$reward = strength(s_2) \tag{12}$$

In other words, the immediate reward we get for transitioning to state s_2 from s_1 is the cryptographic strength of the state s_2. This indicates how good that particular transition is for us.

7 Results

We ran the experiment with set sizes of both 2 and 3 semi-bent Boolean functions. We ran the RL algorithm with each configuration 10 times, each time the algorithm traversed fifty unique states. The results obtained are summarized in Table 1. *Set Size* is the number of CA rules used in the S-box. The columns *DU, NL* and *Strength* represent the best values obtained for Differential Uniformity, Nonlinearity and Strength (computed using 9) by the particular design. $\overline{Strength}$ is the average strength of the S-boxes that were explored during the ten runs. $\sigma_{Strength}$ gives the standard deviation of the average strengths obtained by the S-boxes in the ten runs. The best possible properties obtained of the S-boxes created from the design consisting of 2 rules had a differential uniformity of 32 and nonlinearity of 96. There were multiple states that gave these properties. With the design consisting of 3 rules, the best S-box obtained had a nonlinearity of 96 and differential uniformity of 16. Again, there were multiple states that gave this result. The values obtained are on par with the values obtained using Genetic Programming. Furthermore, our design using 3 semi-bent functions was able to outperform the S-box obtained using genetic programming in both differential uniformity as well as nonlinearity. The genetic programming based S-box had a maximum differential uniformity and nonlinearity of 20 and 82 respectively [2].

Table 1. Summary of the results

Set Size	DU	NL	Strength	$\overline{Strength}$	$\sigma_{Strength}$
2	32	96	81.57	58.42	2.51
3	16	96	88.02	59.72	3.04

8 Conclusion and Future Work

It can be successfully concluded that not only genetic programming but reinforcement learning can also be used to generate S-boxes with strong cryptographic properties. The semi-bent Boolean function based S-boxes are relatively lightweight in their implementation as well.

Our work is heavily constrained by the computational resources and time. As Reinforcement Learning is a computationally intensive task, it requires very high performance computing machines. Certain computations such as the calculation of cryptographic properties is also a very compute intensive task. In our work, we only explored S-boxes created by 2 and 3 semi-bent functions. This work can be further expanded by increasing the number of rules used to 4, 5, etc. The manner of usage of rules can also be changed. In our work, the rules were used in an in-order manner. This can be changed to alternating rules, or randomly selecting rules from a subset of the rules. In the reinforcement learning part, a lot can be expanded and built on. Other control algorithms such as the off-policy Q-learning, or Expected Sarsa, can be used instead of the Sarsa algorithm used in our design. Other policies can also be implemented such as policy gradient methods instead of the epsilon greedy policy used in our design. The parameters to the function approximators (ANN) can be changed and experimented with. This work only focused on 8×8 S-boxes, we can also try to modify the design to work on different input and output sizes.

A Appendix

The source code for the S-box design and RL implementation is available at https://github.com/tarunaygr/RL-based-S-boxes.

References

1. Mariot, L., Saletta, M., Leporati, A., et al.: Exploring semi-bent Boolean functions arising from cellular automata. In: Gwizdałła, T.M., Manzoni, L., Sirakoulis, G.C., Bandini, S., Podlaski, K. (eds.) Cellular Automata: 14th International Conference on Cellular Automata for Research and Industry, ACRI 2020, Lodz, Poland, December 2–4, 2020, Proceedings, vol. 12599, pp. 56–66. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-69480-7_7
2. Picek, S., Mariot, L., Leporati, A., Jakobovic, D.: Evolving S-boxes based on cellular automata with genetic programming. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO 2017, pp. 251–252. Association for Computing Machinery, New York (2017). https://doi.org/10.1145/3067695.3076084
3. Kari, J.J.: Basic concepts of cellular automata. In: Rozenberg, G., Bäck, T., Kok, J.N. (eds.) Handbook of Natural Computing, pp. 3–24. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-540-92910-9_1
4. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction, 2nd edn. The MIT Press, Cambridge, Massachusetts, London, England (2018)

5. Wolfram, S.: Statistical mechanics of cellular automata. Rev. Mod. Phys. **55**(3), 601 (1983)
6. afify, E., Khalil, A.T., El sobky, W.I., Alez, R.A.: Performance analysis of advanced encryption standard (AES) S-boxes. Int. J. Recent Technol. Eng. (IJRTE) **9**(1), 2214–2218 (2020). https://doi.org/10.35940/ijrte.F9712.059120
7. Mariot, L., Picek, S., Leporati, A., Jakobovic, D.: Cellular automata based S-boxes. Cryptogr. Commun. **11**(1), 41–62 (2018). https://doi.org/10.1007/s12095-018-0311-8
8. Carlet, C.: Boolean Functions for Cryptography and Coding Theory. Cambridge University Press, Cambridge (2021). https://doi.org/10.1017/9781108606806
9. Mariot, L., Saletta, M., Leporati, A., et al.: Heuristic search of (semi-)bent functions based on cellular automata. Nat. Comput. (2022). https://doi.org/10.1007/s11047-022-09885-3