A 10-bit S-box generated by Feistel construction from cellular automata*

Thomas Prévost[1 [0009−0000−2224−8574]] and Bruno Martin[1 [0000−0002−0048−5197]]

Université Côte d'Azur, CNRS, I3S, France
{thomas.prevost,bruno.martin}@univ-cotedazur.fr

Abstract. In this paper, we propose a new 10-bit S-box generated from a Feistel construction. The subpermutations are generated by a 5-cell cellular automaton based on a unique well-chosen rule and bijective affine transformations. In particular, the cellular automaton rule is chosen based on empirical tests of its ability to generate good pseudorandom output on a ring cellular automaton. Similarly, Feistel's network layout is based on empirical data regarding the quality of the output S-box.
We perform cryptanalysis of the generated 10-bit S-box: we test the properties of algebraic degree, algebraic complexity, nonlinearity, strict avalanche criterion, bit independence criterion, linear approximation probability, differential approximation probability, differential uniformity and boomerang uniformity of our S-box, and relate them to those of the AES S-box. We find security properties comparable to or sometimes even better than those of the standard AES S-box. We believe that our S-box could be used to replace the 5-bit substitution of ciphers like ASCON.

Keywords: S-box · Block cipher · Cellular automata · Feistel permutation · Boolean functions.

Acknowledgements We would like to thank Jean-Charles Regin (`http://www.constraint-programming.com/people/regin/`) for kindly lending us his computing machines.

Introduction

Cryptography today plays a leading role in the development of telecommunications. Symmetric encryption is an important part of modern cryptography. It must allow two parties who share a common secret key to exchange enciphered data. The encrypted data should look like random bits from the perspective of an external attacker who does not have the key.

There are two main families of symmetric ciphers: stream ciphers and block ciphers. Stream ciphers encrypt data on the fly, bit by bit, while block ciphers

* This work has been supported by a government grant managed by the Agence Nationale de la Recherche under the Investissement d'avenir program, reference ANR-17-EURE-004

treat data as a series of blocks of a specific size. It is the latter which is most used today. AES (*Advanced Encryption Standard*) or Blowfish are the best known block cipher algorithms.

Substitution boxes (abbreviated S-boxes) are the most important nonlinear component of many block ciphers. They play the role of input bits mixer and are essential for the security of the cipher. This is the part that must be designed with the greatest attention, since it is on its weakness that most attacks focus.

The designer of substitution boxes must in particular ensure that his S-box is resistant against linear [5], differential [4] or boomerang [44] attacks, which are today the main threats to the security of S-boxes.

However, it is impossible to study all possible n-bit S-boxes, starting from a certain n. Indeed, an S-box can be seen as a permutation on the discrete set $[\![0, 2^n - 1]\!]$. So there is a total of $2^n!$ possible S-boxes. For 8-bit S-boxes like AES, this represents approximately $8.58 \cdot 10^{506}$ possible S-boxes. For 10-bit S-boxes like the one we propose, there are $5.42 \cdot 10^{2639}$ possible permutations. It is therefore absolutely unthinkable to study them all exhaustively.

Current constructions of S-boxes rely on algebraic approaches by using properties of finite fields, like AES. We propose here a combinatorial construction with a S-box based on a Feistel construction of depth 11. This construction consists of three layers of bijective affine transformation, and eight layers of permutations by a uniform binary cellular automaton (CA) of dimension 1, with a well-chosen local function considered as a Boolean function. This function is used as a pseudo-random permutation.

First, we discuss the related work in this area. In section 2 we recall the definitions of Boolean functions and uniform cellular automata. We give some important properties, which will be useful in the rest of this paper. Next, we recall Feistel constructions, and how they can be used to generate cryptographically secure random permutations in section 3 (in particular thanks to the Luby-Rackoff theorem). In section 4, we explain how we generate a 10-bit S-box from a Feistel construction based on uniform cellular automata permutations. Subsequently, we carry out the cryptanalysis of the S-box thus obtained, according to different criteria (section 5). Finally, we conclude on the possible use of this type of construction, and suggestions to generate larger S-boxes.

1 Related work

There are many research papers that propose new methods for generating S-boxes. Most focus on 8-bit S-boxes, although some offer smaller S-boxes.

The generation of $2n$-bit S-boxes from n-bit subpermutations has already been considered in the literature, either by Feistel or MISTY constructions [27,8].

Many other methods have also been considered. Burnett et al. proposed a heuristic method to generate MARS-like S-boxes [14]. Methods based on genetic programming were used to successively select the S-boxes with the best cryptographic properties [39,38]. Many stochastic methods have been proposed to generate S-boxes with the best possible properties [31]. Others have already thought about using chaotic functions to generate S-boxes [15].

The AES S-box [12] nevertheless remains to this day the security reference for 8-bit S-boxes. This S-box is a finite field polynomial construction (and the security standard). Many other scientific papers also propose S-boxes based on polynomial constructions [49,50].

Another property currently sought in the design of S-boxes is their energy efficiency, that is to say that carrying out the permutations uses the minimum of resources. This involves designing the S-box according to the internal functioning of the processor's logic gates. Such S-boxes have been proposed on 8 bits [20].

Other 8-bit S-boxes have been proposed to be easily adaptable to FPGA (Field Programmable Gate Array) [29].

Other ciphers, on the contrary, rely on smaller S-boxes to further reduce energy consumption [6,16]. However, such designs cannot be done without a reduction in the cryptographic quality of the S-box, and therefore in the security of the cipher [35].

To our knowledge no paper has been published on the generation of 10-bit S-boxes.

As indicated in section 4.2, the construction of S-boxes from functions validating the NIST FIPS 140-2 test has already been explored by [51], but it was not cellular automata that were then tested.

The search for Boolean functions with chaotic behavior was first studied by Wolfram in 1983 [47]. He discovered that the cellular automaton rule 30 presents the best chaotic evolution. However, the Siegenthaler bound tells that functions with 3 variables are not suitable for cryptography [32].

The classification of Boolean functions has already been done according to multiple criteria [1,26], we do not bring much new in this area except perhaps their selection based on a random test.

The use of cellular automata for cryptography is not recent [11]. Gutowitz proposed in 1993 the use of cellular automata for the block cipher [19]. Several papers have already been proposed to construct S-boxes or hash-functions from such automata [30,25]. However, to our knowledge, no one has yet designed an S-box from a cellular automaton based Luby-Rackoff construction.

2 Definitions and notation

2.1 Uniform cellular automata

Cellular Automata (CA) form a model of discrete parallel computation, composed of cells. Each cell is a finite state machine. At each time step, all the cells of the cellular automaton update their state synchronously according the states of their neighbors and their current state, following a local rule. The best known cellular automaton is Conway's Game of Life, which is two-dimensional.

We can define formally 1-dimensional cellular automata as triples (Q, δ, N) where:

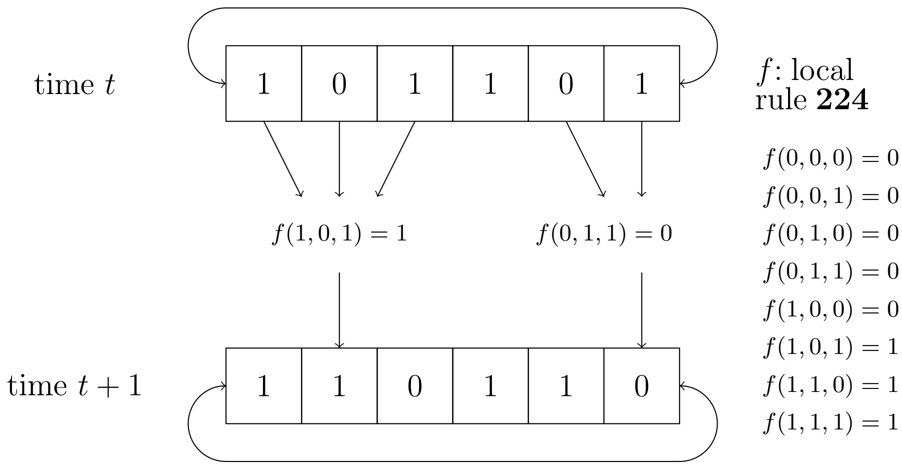

Fig. 1. Example of a 1-dimensional uniform cellular automaton with the single 3-bit local rule 224. To calculate the next value of a cell on the border, we consider the cell on the other edge as neighboring this one.

- Q is a finite set of states, here the set of states is $\{0, 1\}$, the Boolean values.
- δ is the local transition function $Q^n \longrightarrow Q$, called *rule*. n is the *arity* of the rule. Here we use Boolean functions as local transition rules.
- $N \subseteq \mathbb{Z}$ is the finite neighborhood, $card(N) = a$ being the size of the CA.

Here we are interested in 1-dimensional cellular automata. It is a finite ring of cells, each containing a Boolean value. At each time step, each cell is updated according to itself and its neighbors. It is possible to have several local rules within the same cellular automaton, this CA then called *non uniform.*

A cellular automaton is said to be **uniform** if it applies the same local transition rule for all its cells.

Here we consider uniform cellular automata, with an n-variable Boolean local transition function (or rule) δ. At each time step, each cell is modified according to the result of δ on itself and its $n-1$ neighbors. In the case of cells at the edge of the automaton, we arbitrarily choose to count the cells on the other edge as neighbors, thus forming a ring, as shown in Fig. 1.

By choosing a good local rule, it is possible to create a pseudo-random bit generator, as shown in [18]. Generally speaking, certain local rules are capable of producing a chaotic effect on the states taken by designated cells at successive time steps. The best known is the 3-bit rule 30, as shown by Wolfram in 1983 [47]. Most pseudo-random bit generators, however, use Linear-Feedback Shit Registers (LFSRs) [43].

However, uniform automata can have short and therefore non-chaotic cycles depending on the inputs. For example, a uniform automaton filled only with ones will only be able to take successive ones or zeros as the value at the next time step. It is therefore advisable to be careful with these particular inputs when using a uniform cellular automaton for cryptographic applications.

2.2 Boolean functions

A Boolean function is a function that takes n Boolean values as input and returns a single Boolean value as output, n being the number of variables in the function. The local transition functions of cellular automata can be viewed as Boolean functions.

Truth table According to Wolfram's numbering, Boolean functions are characterized by their truth table, which lists the outputs corresponding to the inputs, unique in the set of functions with a given number of variables.

Example 1. In the set of 3-bit functions, rule 30 is expressed 00011110 in binary. Starting with the least significant bit, this means that $f(0,0,0) = 0$, $f(0,0,1) = 1$, $f(0,1,0) = 1$ etc.

There are 2^{2^n} n-variable Boolean functions. A convenient way to represent them is given by the Algebraic Normal Form, that we present in the next subsection.

Algebraic Normal Form

Definition 1. *Any n-variable Boolean function f can be expressed by a unique binary polynomial, called Algebraic Normal Form (ANF):*
$f(x) = \bigoplus_{u \in \mathbb{F}_2^n} a_u(\prod_{i=1}^{n} x_i^{u_i}), a_u \in \mathbb{F}_2, u_i$ *i-th projection of u, x_i being the i-th bit of input x.*

Example 2. The ANF of rule 30 is $x_1 \oplus x_2 \oplus x_3 \oplus x_2 \cdot x_3$.

Example 3. The ANF of the 3-variable function $\chi : \mathbb{F}_2^3 \longrightarrow \mathbb{F}_2$ used by Keccak [3] is $\chi(x_1, x_2, x_3) = x_1 \oplus x_2 \cdot x_3 \oplus x_3$. Its rule number is 210.

Definition 2. *The algebraic degree of a function f counts the number of variable in the largest monomial $x_1^{u_1}...x_n^{u_n}$ of its ANF.*

Example 4. The largest monomial of rule 30 is $x_2.x_3$, its degree is 2, as well as rule 210.

A function f is said to be *nonlinear* if and only if its degree is at least 2.

Hamming weight

Definition 3. *The Hamming weight of a Boolean function f, written $w_h(f)$, is the number of $x \in \mathbb{F}_2^n$ such that $f(x) = 1$.*

Definition 4. *A n-variable Boolean function f is balanced if and only if $w_h(f) = 2^{n-1}$ (it returns as many ones as zeroes).*

Correlation-immunity

Definition 5. *An n-variable Boolean function f is k-correlation immune, $1 \leq k \leq n$, if and only if for any binary random input $x = x_1, ..., x_n$, $f(x)$ is statistically independent from any subset of size k of x.*

The Walsh-Hadamard transform [9] is an essential tool for analyzing the statistical properties of a Boolean function. The Walsh-Hadamard transform of a Boolean function f is defined by:

$$\hat{f}(\omega) = \sum_{x=0}^{2^n-1} (-1)^{f(x) \oplus x \cdot \omega} \tag{1}$$

where $x \cdot \omega = \sum_{i=0}^{n-1} x_i \cdot \omega_i$ denotes the dot product of the two binary vectors.

Theorem 1. *A n-variable Boolean function f is k-order correlation immune, $1 \leq k \leq n$ if and only if for every $\omega \in \mathbb{F}_2^n$ such that $1 \leq w_h(\omega) \leq k$, $\hat{f}(\omega) = 0$.*

Xiao and Massey proved theorem 1 in [48]. A Boolean function that is both balanced and correlation immune at order k is said to be *resilient at order k*.

Strict avalanche criterion

Definition 6. *A n-variable Boolean function f satisfies the Strict Avalanche Criterion (SAC) if and only if $\forall i \in [\![1, n]\!]$, flipping the i-th bit of the input x results in the output $f(x)$ being changed for exactly half of the inputs x.*

The strict avalanche criterion is particularly interesting in the cryptographic context since it makes it difficult to infer input from output. It makes the Boolean function "chaotic".

3 Feistel constructions

The Feistel construction [17] is a method for constructing secure pseudo-random bijective permutations from pseudo-random functions. The Feistel network, from a certain depth, guarantees the computational indistinguishability of its pseudo-random permutation from a random permutation.

Definition 7. *A function $f : \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^n$ is said to be* pseudo-random *(PRF) if its output is computationally difficult to distinguish from a random output.*

Definition 8. *A pseudo-random function $f : \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^n$ is called* pseudo-random permutation *(PRP) if and only if it is bijective.*

As shown in Fig. 2, the Feistel construction creates a block permutation function of size n. It is made up of a stack of layers, each composed of PRP f_i of input and output size $\frac{n}{2}$. We call *depth* the number of sub-permutations f_i.

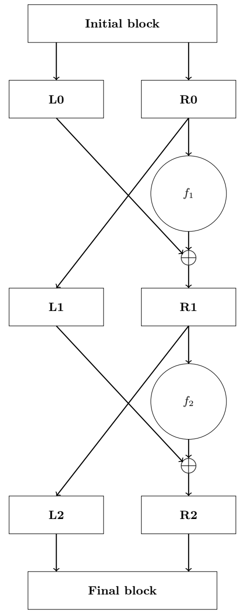

Fig. 2. Example of Feistel construction of depth 2. f_1 and f_2 are pseudo-random permutations

Luby and Rackoff proved in [28] that the output of the LR function is computationally indistinguishable from a random output as long as the depth of the network is at least 4, even for an adversary who knows the input (Known-Plaintext-Attack, KPA).

As shown by [37], a Feistel construction with a depth of at least 7 returns an output that is computationally indistinguishable from a random output for an adversary able to choose the input value (Chosen-Plaintext-Attack, CPA), that is to say, there is no probabilistic algorithm that is capable of making the distinction in polynomial time.

4 Our 10-bit S-box from a Cellular Automata based Feistel construction

4.1 Architecture of the Feistel construction

The permutation function generated by the Feistel construction allows to construct an S-box. We pass the $2^{10} = 1024$ possible inputs to the function, the output of which gives us the S-box. The latter must validate several security requirements, explained in section 5. Another important property to respect is bijectivity, which makes it possible to invert the S-box and therefore to proceed with decryption. In short, it is necessary that for the S-box $S : \mathbb{F}_2^{10} \longrightarrow \mathbb{F}_2^{10}$:

$$\forall x, y \in \mathbb{F}_2^{10}, S(x) = S(y) \implies x = y \tag{2}$$

In our network, we use a 5-cell cellular automaton as a pseudo-random permutation f_i, its output is evaluated after a single time step on the input. The automaton has only one local transition function with 5 variables, which we will detail in section 4.2.

However, as explained in section 2.1, a uniform cellular automaton will fail to return chaotic output for some particular inputs that are regular. If we used only this type of cellular automata for intermediate permutation functions f_i, some inputs would still return a predictable result, for example $S(0)$ or $S(1023)$ which would return 0 or 1023 (0b1111111111).

Fortunately, [34] tells us that it is possible to replace certain pseudo-random permutations by pair-wise independent permutations, i.e. permutations whose output is "almost" uniformly distributed for any two given inputs. An affine function $f_{a,b}(x) = a \cdot x + b$ satisfies these requirements. So that the permutation is bijective, we chose a and b prime.

The Luby-Rackoff construction we chose to generate our 10 bits S-box consists of the following eleven layers:

1. A first layer uses the affine function $f_{5,3}(x) = 5.x + 3 \mod 2^{10}$.
2. Next comes 4 layers using the 5-bit-cellular automaton which will be defined in section 4.2 as pseudo-random permutation.
3. The next layer uses the affine function $f_{7,11}(x) = 7.x + 11 \mod 2^{10}$.
4. Next we have 3 layers of cellular automaton.
5. We have another affine layer, $f_{13,17}(x) = 13.x + 17 \mod 2^{10}$.
6. Finally a last layer reuses the 5-bit-cellular automaton.

All affine functions are expressed modulo $2^{10} = 1024$. Fig. 3 schematizes our LR construction.

4.2 Construction of the local pseudo-random permutation with a cellular automaton

Construction of the cellular automaton We are looking for a cellular automaton which takes as input the value to be permuted, and which returns the result of the permutation. For this, we build a cellular automaton in a ring of 5 cells. To perform the permutation, we assign to the cells of the ring the value to be permuted, then we return the value of the cellular automaton after a single time step. We chose this construction because there are no 4-variable Boolean functions which have the right cryptographic properties and which allow us to create a bijective cellular automaton.

Basic properties of local transition rule There are a total of $2^{2^5} = 2^{32} =$ 4.294.967.296 5-bit local transition Boolean functions. The papers [47,32,18] fortunately give us some ideas for selecting the Boolean functions most likely to introduce "chaos" into the output of the cellular automaton.

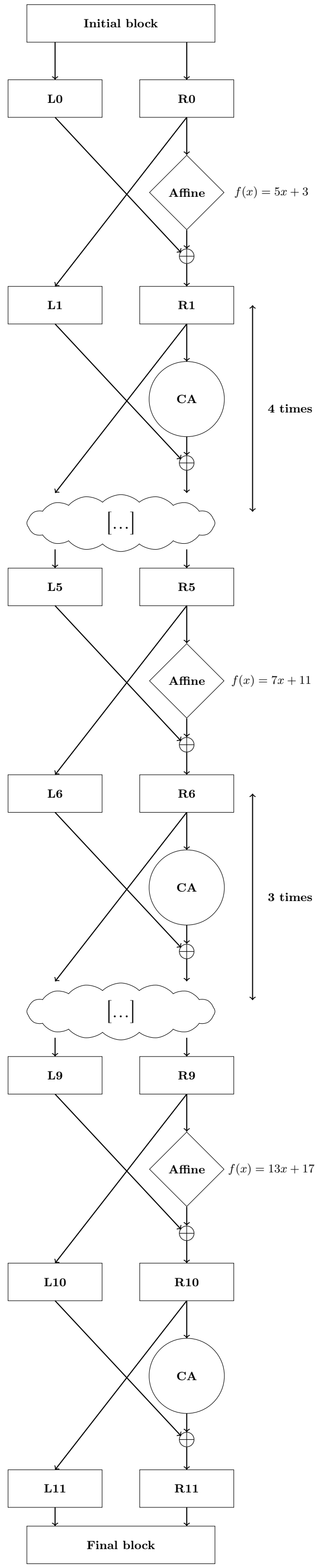

Fig. 3. Selected Feistel construction, with eleven layers. The "CA" functions correspond to the output the 5-cell cellular automaton after one time step. Affine functions are expressed modulo $2^{10} = 1024$.

Let us start by keeping only balanced Boolean functions, as explained by definition 4. There are then $\binom{32}{16} = 601.080.390$ functions left.

We then eliminate the functions which are not first-order correlation immune, as explained in definition 5, to keep only 807.980 rules.

We also eliminate linear functions, to keep 807.928 functions. We are satisfied with the nonlinearity property here: [40] proves that there cannot exist bent functions (maximally non-linear) with an odd number of variables. As we will show later, it was unnecessary for our functions to belong to the "almost-bent" class [7].

Finally we eliminate all functions that do not respect the Strict Avalanche Criterion (SAC), explained in definition 6, to keep 7.080 local rules.

Selection over NIST FIPS 140-2 randomness test We were inspired by [51] which gives us an original method to search for the most "chaotic" local rules. We try to create a pseudo-random generator from a uniform cellular automaton, as explained by [18], then we only keep the rules which allow us to create "good" pseudo-random bit generators.

We start by creating a ring cellular automata, as explained in section 2.1, of size 1024 bits. Indeed, [41] informs us that the ring must have > 1000 cells to produce a secure pseudo-random generator. Having a size equal to a power of 2 simply speeds up the calculations.

For the seed value, we fill the ring with 1024 truly random bits, downloaded from the website `https://www.random.org/`. In order to have a reproducible experience, you can find the seed at the following address: `https://github.com/thomasarmel/cellular_automata_prng` (although the seed value should not influence the results). You will also find that the pseudo random generator by 1024 cells cellular automaton.

Next, we test each of the 7.080 local rules as follows: at each time step, we update all cells in the ring with the rule under test. We then extract the 512th bit from the ring. We repeat the operation for as many bits as we wish to extract.

For each of the 7.080 rules, we evaluate the pseudo-random bit generator with the NIST FIPS 140-2 test [42]. The generator must pass all tests ("Monobit", "Poker", "Runs", "Long run" and "Continuous run") out of 100.000 bits generated, which is equivalent to passing each test 39 times. This threshold of 100.000 bits is arbitrary, but more than sufficient to eliminate any statistical bias.

There then remain 53 rules which allow the ring cellular automaton to validate the NIST FIPS 140-2 pseudo-random generation test.

Cellular automaton bijectivity Finally, we must ensure that the 5-bit ring cellular automaton of the Luby-Rackoff construction is bijective. To do this, we eliminate all the local rules which do not allow us to create a bijective cellular automaton.

Finally, only one rule remains, whose truth table numbering is 1.438.886.595 (in decimal), or $x_0 \cdot x_3 + x_1 \cdot x_3 + x_2 \cdot x_3 + x_3 \cdot x_4 + x_1 + x_2 + x_3 + 1$ in its ANF form.

The generated S-box can be found in Appendix A. The source code to generate the S-box can be found at the following address: `https://github.com/thomasarmel/luby_rackoff_sbox_finder`.

5 Cryptanalysis

Here we propose the cryptanalysis of the specific S-box that we generated from the construction described above. In order to have a more systemic analysis of the security of S-boxes generated from Feistel networks, the reader can refer to [8].

5.1 S-box analysis criteria

[45] gives us the main properties expected on an S-box. The latter being the only non-linear component of a block cipher algorithm, it must be able to resist linear [5] and differential [4] cryptanalysis. We will also discuss resistance to the boomerang attack [44].

An S-box must also be bijective, but this was already addressed in section 4.

To quantify the security of our S-box, we will compare it to the AES S-box [12], which is the industry standard for security today. The latter has a dimension of 8 bits, but we have not found any 10-bit S-box that is used in practice. When necessary, we will therefore adapt our metrics to compare them to an S-box of a different size. We will also compare certain metrics to other S-boxes presented in the literature. We particularly used the SageMath language to calculate certain metrics, the latter offering a library dedicated to the cryptographic properties of S-boxes (`https://doc.sagemath.org/html/en/reference/cryptography/sage/crypto/sbox.html`). Sage notably allowed us to calculate algebraic complexity, nonlinearity, linear and differential approximation probabilities, differential uniformity and boomerang uniformity.

5.2 Algebraic degree

It is possible to represent an S-box by its component functions. For an S-box of size n, $S : \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^n$, then it is possible to write $S(x_1, x_2, ..., x_n) = (y_1, y_2, ..., y_n)$.

The component functions are then the n Boolean functions $f_i : \mathbb{F}_2^n \longrightarrow \mathbb{F}_2$, such that $f_i(x_1, x_2, ..., x_n) = y_i$, for $i \in [\![1, n]\!]$.

As explained by definition 1, any Boolean function can be expressed in its ANF form. A possible attack against an S-box consists of trying to approximate its value by component functions of low degree, this is a low order approximation attack [33]. A high minimum algebraic degree, as explained in definition 9, makes it possible to protect against this type of attack.

Definition 9. *The minimal algebraic degree of an S-box is the minimal degree of its component functions. The maximal algebraic degree is the maximal degree of the component functions.*

Our S-box has a minimum algebraic degree of **8** and a maximum degree of 9. In comparison, the minimum and maximum degree of the AES S-box is 7 (having a larger S-box gives us an advantage).

5.3 Algebraic complexity

The algebraic complexity of an S-box defines its ability to resist interpolation attacks [23].

Definition 10. *The* algebraic complexity *AC of a n-bit S-box S is the number of monomials in the univariate polynomial representation of S such that*

$$S(x) = a_0 + a_1.x + ... + a_{2^n-1}.x^{2^n-1}$$

The algebraic complexity of our S-box is **1023**, which is the maximum possible. The algebraic complexity of the AES S-box is 255, which is also the highest possible value.

5.4 Nonlinearity

Strong nonlinearity [10] allows the S-box to resist linear cryptanalysis. It is defined as the minimum nonlinearity of each of the component functions.

Definition 11. *For a n-variable Boolean function f_i, the nonlinearity N_{f_i} is given by the equation*

$$N_{f_i} = 2^{n-1} - \frac{1}{2} \max_{\omega \in \mathbb{F}_2^n} |\hat{f}_i(\omega)| \tag{3}$$

with $\hat{f}_i$ the Walsh-Hadamard transform of f_i, as defined in 2.2.

The nonlinearity of our S-box is **434**. It is difficult to compare with the nonlinearity of the AES S-box (112), because the two S-boxes do not have the same size. If we divide the constructed S-box nonlinearity by $2^{10-8} = 4$, we obtain a nonlinearity of 108.5, which is a little less than AES but better than many S-boxes presented in the literature. For both AES S-box and ours however, it is not possible to express the value of one of the output bits as a function of a linear combination of the input bits with a probability $\geq 60\%$ (56.25% for AES S-box and 57.62% for our S-box). We used the following GitHub repository to obtain these values: `https://github.com/PoustouFlan/SUnbox`.

5.5 Strict avalanche criterion (SAC)

The notion of *Strict avalanche criterion* (SAC) for the design of S-boxes was first introduced in 1985 by [46]. To satisfy the SAC, half of the output bits must be modified when a single input bit is modified. For an S-box, the bits of the SAC dependency matrix must be close to the ideal value of 0.5.

Table 1. SAC dependency matrix of the constructed S-box. Each row represents the modified input bit, and each column the impact on the output bit. For example, flipping the first input bit will change the first output bit 51% of the time.

0.51	0.44	0.48	0.48	0.48	0.45	0.47	0.48	0.48	0.50
0.54	0.52	0.50	0.51	0.53	0.53	0.48	0.48	0.53	0.50
0.52	0.48	0.54	0.48	0.53	0.52	0.48	0.50	0.54	0.49
0.51	0.54	0.50	0.50	0.50	0.53	0.46	0.51	0.50	0.51
0.51	0.52	0.51	0.46	0.48	0.52	0.52	0.54	0.54	0.54
0.46	0.48	0.48	0.50	0.52	0.51	0.48	0.48	0.50	0.47
0.51	0.49	0.49	0.54	0.50	0.49	0.50	0.52	0.51	0.57
0.46	0.50	0.47	0.50	0.51	0.48	0.48	0.50	0.54	0.50
0.49	0.50	0.48	0.50	0.47	0.49	0.55	0.48	0.48	0.52
0.48	0.49	0.50	0.47	0.50	0.52	0.52	0.55	0.53	0.49

Table 1 gives the SAC dependency matrix of the proposed S-box. We used the following GitHub repository to calculate these values: `https://github.com/abrari/block-cipher-testing/`.

Table 2 compares the average value as well as the extreme values of our dependency table to those of the AES S-box.

Table 2. SAC dependency matrix comparison between AES and proposed S-box

	Average	**Minimum**	**Maximum**
AES	0.50	0.45	0.56
Proposed	0.50	0.44	0.57

Our average value is good, and the extreme values are almost as good as those of the AES S-box.

5.6 Bit Independence Criterion (BIC) parameter

The concept of Bit Independence Criterion (BIC) was first introduced by [46].

Definition 12. *We say that a n-bit S-box S satisfies the BIC if $\forall i, j, k \in [\![1, n]\!]$, with $i \neq j$, inverting the k^{th} bit of the input changes the i^{th} and the j^{th} output bits independently.*

The metric we use for S-boxes is called the *Bit Independence Criterion Parameter*, which measures how far an S-box is from validating the BIC. This distance is between 0 and 1, the closer to 0 being the better.

To calculate this parameter, we need to know the BIC parameter between two output bits i and j. The latter is defined as the maximum correlation coefficient between output bits i and j after inversion of input bit k, for all k.

The BIC parameter of an S-box is the maximum value of the BIC parameter of output bits i and j, for all combinations of i and j such that $i \neq j$.

The BIC parameter of our S-box is **0.124**. For comparison, the one of the AES S-box is 0.134. Our BIC parameter is therefore better than the one of the AES S-box, and also better than other S-boxes proposed in the scientific literature. For example, the S-box of the block cipher PRESENT [6] has a BIC parameter of 1.

We thank the author of the GitHub repository `https://github.com/abrari/block-cipher-testing/` for the code which allowed us to calculate the BIC of our S-box.

5.7 Linear Approximation Probability (LAP)

The *Linear Approximation Probability* (LAP) gives us an indication of how resistant our S-box is to linear cryptanalysis [5]. It is calculated, for a n-bit S-box S, by the maximum correlation between $x.\alpha$ and $S(x).\beta$, $\forall \alpha, \beta \in [\![0, 2^n - 1]\!]$ (except when $\alpha = \beta = 0$).

The paper [21] gives us the following equation for calculating LAP:

$$LAP = \max_{\alpha,\beta \in [\![0,2^n]\!], \alpha+\beta \neq 0} \left| \frac{card\{x \in \mathbb{F}_2^n | \alpha \cdot x = \beta \cdot S(x)\}}{2^n} - \frac{1}{2} \right| \tag{4}$$

The LAP can be calculated from the Linear Approximation Table, as explained by [13]. To do this, we take the maximum correlation value from the table, except at the coordinate 0, 0 (for which the correlation is logically 1), we then obtain the "correlation potential" ϵ. We then calculate LAP using:

$$LAP = (2 \cdot \epsilon)^2 \tag{5}$$

The Linear Approximation Probability of our S-box is **9.28%**, which is comparable or even better than other S-boxes proposed in the scientific literature [2]. However, the LAP is a little bit worse than that of the AES S-box, which is 6.25%.

5.8 Differential Approximation Probability

The *Differential Approximation Probability* is determined by the XOR distribution between the input and output of an S-box. The lowest possible value guarantees the security of the S-box against differential cryptanalysis [4].

The DAP is given by the maximum value of the differential probability table.

Let us denote by $\Delta x \in \mathbb{F}_2^n$ an input of the S-box, and by $\Delta y \in \mathbb{F}_2^m$ an output, with m the output size in bits of the S-box. For each Δx, Δy of a n-bit S-box differential probability table, the probability is calculated by:

$$DP(\Delta x \to \Delta y) = \frac{card\{x \in \mathbb{F}_2^n | S(x) \oplus S(x \oplus \Delta x) = \Delta y\}}{2^n} \tag{6}$$

And so

$$DAP = \max_{\Delta x, \Delta y} DP(\Delta x \to \Delta y) \tag{7}$$

The DAP of our S-box is **1.37%**, which is better than the AES S-box, which has a DAP of 1.56%. This DAP is also better than many S-boxes presented in the literature [22].

5.9 Differential Uniformity

The *Differential Uniformity* of an S-box defines its proximity to perfect non-linearity [36]. For an n-bit S-box S, its Differential Uniformity δ_S is defined by the equation

$$\delta_S = \max_{a,b \in \mathbb{F}_2^n, a \neq 0} \delta(a,b) = \max_{a,b \in \mathbb{F}_2^n, a \neq 0} card\{x \in \mathbb{F}_2^n | S(x \oplus a) \oplus S(x) = b\} \quad (8)$$

The Differential Uniformity of our 10-bit S-box is **14**. Let us divide this value by $\frac{2^{10}}{2^8} = 4$ in order to compare with 8-bit S-box, we obtain 3.5. This is a better value than other S-boxes presented in the scientific literature [51,24], and even better than the Differential Uniformity of the AES S-box which is 4.

5.10 Boomerang Uniformity

The *Boomerang Uniformity* $\mathcal{BU}$ defines the resistance of an S-box to the boomerang attack [44], which is an improvement of differential cryptanalysis. A small $\mathcal{BU}$ value provides better resistance to the boomerang attack.

To calculate the $\mathcal{BU}$ of an n-bit S-box, we start by calculating the Boomerang Connectivity Table (BCT), an $n \times n$ matrix whose entry in the $\Delta_i \in \mathbb{F}_2^n$ row and in the $\Delta_o \in \mathbb{F}_2^n$ column is given by:

$$BCT(\Delta_i, \Delta_o) = card\{x \in \mathbb{F}_2^n | S^{-1}(S(x) \oplus \Delta_o) \oplus S^{-1}(S(x \oplus \Delta_i) \oplus \Delta_o) = \Delta_i\} \quad (9)$$

The Boomerang Uniformity $\mathcal{BU}$ is given by the maximum entry of the Boomerang Connectivity Table, ignoring the first row and first column.

Our 10-bit S-box has a $\mathcal{BU}$ of **24**. Let's divide this value by 4 to compare it with 8-bit S-boxes. We then find a value of 6, which is better than to the values found in the literature [35,51], and equal to the $\mathcal{BU}$ of the AES S-box, which is also 6.

Discussion

This method of constructing 10-bit S-box showed security results comparable to the AES standard, now widely used in the industry. However, we believe that it is still possible to improve the quality of our S-box by modifying parameters of the Feistel network. In particular, the quality of the S-box can be improved by increasing the network depth. Changing the parameters of the affine functions would result in different S-boxes, but we expect the security parameters to be roughly equivalent.

It can also be considered to build even larger S-boxes, provided that the number of variables n is even and $\frac{n}{2}$ is odd. For example we could consider building S-boxes with 14 or even 18 variables. However, this would require selecting "good" Boolean functions with 7 or 9 variables, which represents a significant computational challenge. Indeed, the number of possible n-bit Boolean functions is 2^{2^n}. So, for example, there exist $1.34 \cdot 10^{154}$ 9-bit Boolean functions.

Our 10-bit S-box could for example be used in an ASCON-type sponge network [16]. This cipher performs permutations on 320-bit blocks, and for this purpose uses a 5-bit S-box on 64 sub-blocks. We believe that the quality and therefore the security of the permutation would be improved, but the algorithmic complexity would be increased. We give an example of a possible ASCON implementation with our S-box (in Rust) on the following URL: `https://github.com/thomasarmel/sponges/blob/sbox_10/ascon/src/lib.rs#L100`. On our 13^{th} Gen Intel Core i7-13700H 5 GHz CPU, the modified **round()** function is however 10 to 15 times slower than the original one using ASCON's original S-box, but has superior cryptographic quality. It would also be more complex to propose a constant-time implementation of the modified cipher.

Conclusion

In this paper, we propose a new 10-bit S-box from a Feistel construction based on uniform cellular automata permutations, and carried out the cryptanalysis. In particular, we evaluated its robustness against linear, differential or boomerang attacks. We shown that our S-box has comparable, or even better security than other S-boxes presented in the scientific literature. In particular, the security evaluations were comparable, and sometimes even better, than those of the AES S-box, which is today the widely used standard.

To our knowledge, no method for constructing 10-bit S-boxes has ever been proposed in the scientific literature. Our method can be extended for the construction of n-bit S-box, given that n is even and $\frac{n}{2}$ is odd.

Data availability

All the data needed to replicate our results is freely available in open source, from the links mentioned in this paper.

Statements and Declarations

No funds, grants, or other support was received for conducting this study. The authors have no competing interests to declare that are relevant to the content of this article. The authors have no financial or proprietary interests in any material discussed in this article.

Appendix A Generated 10-bit S-box

Table 3. Generated S-box input/output table

input	output	**input**	output	**input**	output	**input**	output	**input**	output	**input**	output	**input**	output	**input**	output
0x000	0x090	**0x080**	0x05a	**0x100**	0x2d8	**0x180**	0x13a	**0x200**	0x238	**0x280**	0x2b7	**0x300**	0x05e	**0x380**	0x2d6
0x001	0x15c	**0x081**	0x123	**0x101**	0x354	**0x181**	0x27c	**0x201**	0x137	**0x281**	0x36a	**0x301**	0x12a	**0x381**	0x244
0x002	0x1c0	**0x082**	0x38d	**0x102**	0x2fa	**0x182**	0x0bf	**0x202**	0x005	**0x282**	0x176	**0x302**	0x04a	**0x382**	0x159
0x003	0x292	**0x083**	0x14d	**0x103**	0x36b	**0x183**	0x1da	**0x203**	0x147	**0x283**	0x1eb	**0x303**	0x15a	**0x383**	0x043
0x004	0x101	**0x084**	0x2c2	**0x104**	0x0c6	**0x184**	0x1bd	**0x204**	0x0a4	**0x284**	0x036	**0x304**	0x1e7	**0x384**	0x162
0x005	0x309	**0x085**	0x089	**0x105**	0x2c7	**0x185**	0x02c	**0x205**	0x3cc	**0x285**	0x010	**0x305**	0x18f	**0x385**	0x36c
0x006	0x228	**0x086**	0x39d	**0x106**	0x3b9	**0x186**	0x306	**0x206**	0x384	**0x286**	0x34d	**0x306**	0x293	**0x386**	0x1e6
0x007	0x36f	**0x087**	0x1bc	**0x107**	0x10a	**0x187**	0x0a2	**0x207**	0x3c1	**0x287**	0x1b1	**0x307**	0x180	**0x387**	0x231
0x008	0x20f	**0x088**	0x157	**0x108**	0x35d	**0x188**	0x32f	**0x208**	0x071	**0x288**	0x340	**0x308**	0x0b9	**0x388**	0x193
0x009	0x28e	**0x089**	0x096	**0x109**	0x042	**0x189**	0x166	**0x209**	0x023	**0x289**	0x205	**0x309**	0x070	**0x389**	0x330
0x00a	0x2fc	**0x08a**	0x1ae	**0x10a**	0x240	**0x18a**	0x132	**0x20a**	0x303	**0x28a**	0x24f	**0x30a**	0x094	**0x38a**	0x329
0x00b	0x379	**0x08b**	0x268	**0x10b**	0x1b9	**0x18b**	0x160	**0x20b**	0x118	**0x28b**	0x0ad	**0x30b**	0x188	**0x38b**	0x242
0x00c	0x214	**0x08c**	0x064	**0x10c**	0x2da	**0x18c**	0x203	**0x20c**	0x366	**0x28c**	0x01f	**0x30c**	0x251	**0x38c**	0x1cf
0x00d	0x085	**0x08d**	0x059	**0x10d**	0x0d6	**0x18d**	0x307	**0x20d**	0x208	**0x28d**	0x2e2	**0x30d**	0x167	**0x38d**	0x241
0x00e	0x2be	**0x08e**	0x0d9	**0x10e**	0x0bc	**0x18e**	0x2bd	**0x20e**	0x215	**0x28e**	0x3e8	**0x30e**	0x37f	**0x38e**	0x336
0x00f	0x269	**0x08f**	0x3dc	**0x10f**	0x06e	**0x18f**	0x05f	**0x20f**	0x177	**0x28f**	0x108	**0x30f**	0x3c9	**0x38f**	0x2e0
0x010	0x392	**0x090**	0x28b	**0x110**	0x192	**0x190**	0x0f0	**0x210**	0x172	**0x290**	0x16f	**0x310**	0x09e	**0x390**	0x11b
0x011	0x358	**0x091**	0x2ac	**0x111**	0x356	**0x191**	0x158	**0x211**	0x38e	**0x291**	0x2d7	**0x311**	0x00e	**0x391**	0x0fa
0x012	0x361	**0x092**	0x15f	**0x112**	0x237	**0x192**	0x0d4	**0x212**	0x01a	**0x292**	0x298	**0x312**	0x0e0	**0x392**	0x110
0x013	0x2c9	**0x093**	0x088	**0x113**	0x0f6	**0x193**	0x22d	**0x213**	0x07c	**0x293**	0x3dd	**0x313**	0x2e6	**0x393**	0x178
0x014	0x23b	**0x094**	0x33f	**0x114**	0x020	**0x194**	0x168	**0x214**	0x372	**0x294**	0x1a9	**0x314**	0x009	**0x394**	0x16c
0x015	0x2ec	**0x095**	0x17a	**0x115**	0x069	**0x195**	0x052	**0x215**	0x2fd	**0x295**	0x1ba	**0x315**	0x02f	**0x395**	0x1ec
0x016	0x1b8	**0x096**	0x38b	**0x116**	0x0d7	**0x196**	0x264	**0x216**	0x0aa	**0x296**	0x1b5	**0x316**	0x38c	**0x396**	0x3b6
0x017	0x055	**0x097**	0x08d	**0x117**	0x24a	**0x197**	0x266	**0x217**	0x1c4	**0x297**	0x376	**0x317**	0x011	**0x397**	0x245
0x018	0x091	**0x098**	0x259	**0x118**	0x225	**0x198**	0x142	**0x218**	0x3a6	**0x298**	0x114	**0x318**	0x33e	**0x398**	0x260
0x019	0x1a8	**0x099**	0x363	**0x119**	0x322	**0x199**	0x236	**0x219**	0x1f5	**0x299**	0x345	**0x319**	0x170	**0x399**	0x2e1
0x01a	0x08a	**0x09a**	0x153	**0x11a**	0x07f	**0x19a**	0x3a2	**0x21a**	0x181	**0x29a**	0x049	**0x31a**	0x015	**0x39a**	0x1ab
0x01b	0x282	**0x09b**	0x2db	**0x11b**	0x220	**0x19b**	0x35a	**0x21b**	0x1b0	**0x29b**	0x0e8	**0x31b**	0x342	**0x39b**	0x2aa
0x01c	0x34f	**0x09c**	0x367	**0x11c**	0x0ed	**0x19c**	0x03c	**0x21c**	0x035	**0x29c**	0x311	**0x31c**	0x030	**0x39c**	0x0a5
0x01d	0x36e	**0x09d**	0x077	**0x11d**	0x216	**0x19d**	0x28a	**0x21d**	0x35c	**0x29d**	0x3bf	**0x31d**	0x09c	**0x39d**	0x1d0
0x01e	0x32e	**0x09e**	0x078	**0x11e**	0x16e	**0x19e**	0x328	**0x21e**	0x186	**0x29e**	0x00c	**0x31e**	0x000	**0x39e**	0x006
0x01f	0x0af	**0x09f**	0x315	**0x11f**	0x2e5	**0x19f**	0x1bf	**0x21f**	0x247	**0x29f**	0x0c1	**0x31f**	0x2bb	**0x39f**	0x06b
0x020	0x212	**0x0a0**	0x060	**0x120**	0x1fc	**0x1a0**	0x3a9	**0x220**	0x0b1	**0x2a0**	0x0a1	**0x320**	0x0c4	**0x3a0**	0x390
0x021	0x37c	**0x0a1**	0x3b8	**0x121**	0x380	**0x1a1**	0x299	**0x221**	0x140	**0x2a1**	0x278	**0x321**	0x202	**0x3a1**	0x152
0x022	0x0f8	**0x0a2**	0x3f5	**0x122**	0x22f	**0x1a2**	0x200	**0x222**	0x2a2	**0x2a2**	0x02e	**0x322**	0x2a6	**0x3a2**	0x0cf
0x023	0x113	**0x0a3**	0x3af	**0x123**	0x32d	**0x1a3**	0x3eb	**0x223**	0x3e7	**0x2a3**	0x368	**0x323**	0x2ad	**0x3a3**	0x001
0x024	0x2b0	**0x0a4**	0x194	**0x124**	0x145	**0x1a4**	0x254	**0x224**	0x3f4	**0x2a4**	0x040	**0x324**	0x3cd	**0x3a4**	0x300
0x025	0x1d9	**0x0a5**	0x15b	**0x125**	0x0e1	**0x1a5**	0x294	**0x225**	0x211	**0x2a5**	0x1c1	**0x325**	0x3e9	**0x3a5**	0x1c9
0x026	0x362	**0x0a6**	0x044	**0x126**	0x0de	**0x1a6**	0x3b3	**0x226**	0x3d2	**0x2a6**	0x3e0	**0x326**	0x14e	**0x3a6**	0x120
0x027	0x2cb	**0x0a7**	0x3aa	**0x127**	0x03b	**0x1a7**	0x16d	**0x227**	0x320	**0x2a7**	0x37e	**0x327**	0x3c2	**0x3a7**	0x232
0x028	0x099	**0x0a8**	0x301	**0x128**	0x06d	**0x1a8**	0x2c3	**0x228**	0x025	**0x2a8**	0x1a2	**0x328**	0x364	**0x3a8**	0x131
0x029	0x1e8	**0x0a9**	0x11e	**0x129**	0x04d	**0x1a9**	0x127	**0x229**	0x27a	**0x2a9**	0x2c5	**0x329**	0x179	**0x3a9**	0x183
0x02a	0x29b	**0x0aa**	0x056	**0x12a**	0x067	**0x1aa**	0x1e5	**0x22a**	0x270	**0x2aa**	0x17e	**0x32a**	0x00b	**0x3aa**	0x0ff
0x02b	0x26f	**0x0ab**	0x2bc	**0x12b**	0x1ee	**0x1ab**	0x1d4	**0x22b**	0x08b	**0x2ab**	0x148	**0x32b**	0x2a0	**0x3ab**	0x21b
0x02c	0x11c	**0x0ac**	0x11f	**0x12c**	0x1e3	**0x1ac**	0x31d	**0x22c**	0x1f9	**0x2ac**	0x20a	**0x32c**	0x319	**0x3ac**	0x0f2
0x02d	0x075	**0x0ad**	0x271	**0x12d**	0x0ef	**0x1ad**	0x10f	**0x22d**	0x331	**0x2ad**	0x33b	**0x32d**	0x00d	**0x3ad**	0x1f2
0x02e	0x26c	**0x0ae**	0x333	**0x12e**	0x0b7	**0x1ae**	0x2f7	**0x22e**	0x0ec	**0x2ae**	0x0f7	**0x32e**	0x11d	**0x3ae**	0x03a
0x02f	0x1bb	**0x0af**	0x230	**0x12f**	0x285	**0x1af**	0x1db	**0x22f**	0x3fb	**0x2af**	0x0fd	**0x32f**	0x3b4	**0x3af**	0x2f5

input	output	**input**	output	**input**	output	**input**	output	**input**	output	**input**	output	**input**	output	**input**	output
0x030	0x3ec	**0x0b0**	0x291	**0x130**	0x2a3	**0x1b0**	0x30b	**0x230**	0x218	**0x2b0**	0x196	**0x330**	0x2b2	**0x3b0**	0x187
0x031	0x027	**0x0b1**	0x12d	**0x131**	0x1b7	**0x1b1**	0x13f	**0x231**	0x1d5	**0x2b1**	0x05d	**0x331**	0x24b	**0x3b1**	0x27e
0x032	0x393	**0x0b2**	0x045	**0x132**	0x084	**0x1b2**	0x314	**0x232**	0x033	**0x2b2**	0x3c6	**0x332**	0x3e1	**0x3b2**	0x3da
0x033	0x2e8	**0x0b3**	0x355	**0x133**	0x2c0	**0x1b3**	0x04c	**0x233**	0x29a	**0x2b3**	0x01e	**0x333**	0x3ef	**0x3b3**	0x079
0x034	0x3e4	**0x0b4**	0x374	**0x134**	0x391	**0x1b4**	0x23f	**0x234**	0x3c7	**0x2b4**	0x219	**0x334**	0x22e	**0x3b4**	0x360
0x035	0x24e	**0x0b5**	0x038	**0x135**	0x274	**0x1b5**	0x243	**0x235**	0x04f	**0x2b5**	0x34c	**0x335**	0x25f	**0x3b5**	0x0be
0x036	0x29e	**0x0b6**	0x092	**0x136**	0x258	**0x1b6**	0x204	**0x236**	0x1a0	**0x2b6**	0x20e	**0x336**	0x115	**0x3b6**	0x144
0x037	0x353	**0x0b7**	0x03f	**0x137**	0x3b1	**0x1b7**	0x37a	**0x237**	0x0ae	**0x2b7**	0x34a	**0x337**	0x3a8	**0x3b7**	0x3d4
0x038	0x175	**0x0b8**	0x3a7	**0x138**	0x318	**0x1b8**	0x004	**0x238**	0x265	**0x2b8**	0x057	**0x338**	0x126	**0x3b8**	0x33a
0x039	0x34e	**0x0b9**	0x0b6	**0x139**	0x30a	**0x1b9**	0x23d	**0x239**	0x3bc	**0x2b9**	0x0ca	**0x339**	0x2ed	**0x3b9**	0x2a4
0x03a	0x19a	**0x0ba**	0x262	**0x13a**	0x223	**0x1ba**	0x396	**0x23a**	0x111	**0x2ba**	0x19f	**0x33a**	0x3fd	**0x3ba**	0x3c5
0x03b	0x03e	**0x0bb**	0x0d1	**0x13b**	0x002	**0x1bb**	0x102	**0x23b**	0x185	**0x2bb**	0x0b3	**0x33b**	0x15e	**0x3bb**	0x2ce
0x03c	0x0d3	**0x0bc**	0x058	**0x13c**	0x357	**0x1bc**	0x3df	**0x23c**	0x3be	**0x2bc**	0x222	**0x33c**	0x25d	**0x3bc**	0x05b
0x03d	0x326	**0x0bd**	0x382	**0x13d**	0x0d2	**0x1bd**	0x1cd	**0x23d**	0x3d6	**0x2bd**	0x0cb	**0x33d**	0x1b3	**0x3bd**	0x0ba
0x03e	0x2b6	**0x0be**	0x1fa	**0x13e**	0x1cb	**0x1be**	0x276	**0x23e**	0x026	**0x2be**	0x13c	**0x33e**	0x365	**0x3be**	0x399
0x03f	0x226	**0x0bf**	0x0dc	**0x13f**	0x1dc	**0x1bf**	0x256	**0x23f**	0x0c9	**0x2bf**	0x3a0	**0x33f**	0x0e4	**0x3bf**	0x39b
0x040	0x351	**0x0c0**	0x323	**0x140**	0x16b	**0x1c0**	0x08c	**0x240**	0x2a5	**0x2c0**	0x095	**0x340**	0x191	**0x3c0**	0x051
0x041	0x2d4	**0x0c1**	0x0a8	**0x141**	0x1d2	**0x1c1**	0x06a	**0x241**	0x38a	**0x2c1**	0x339	**0x341**	0x3ca	**0x3c1**	0x0d8
0x042	0x3d8	**0x0c2**	0x154	**0x142**	0x2d5	**0x1c2**	0x21d	**0x242**	0x20b	**0x2c2**	0x17b	**0x342**	0x003	**0x3c2**	0x313
0x043	0x35f	**0x0c3**	0x0c2	**0x143**	0x281	**0x1c3**	0x150	**0x243**	0x112	**0x2c3**	0x272	**0x343**	0x25a	**0x3c3**	0x09d
0x044	0x161	**0x0c4**	0x195	**0x144**	0x2f3	**0x1c4**	0x346	**0x244**	0x098	**0x2c4**	0x1be	**0x344**	0x1b4	**0x3c4**	0x310
0x045	0x1f3	**0x0c5**	0x1ed	**0x145**	0x1fb	**0x1c5**	0x255	**0x245**	0x007	**0x2c5**	0x10c	**0x345**	0x14f	**0x3c5**	0x32c
0x046	0x3cf	**0x0c6**	0x00f	**0x146**	0x3d5	**0x1c6**	0x207	**0x246**	0x18a	**0x2c6**	0x3ba	**0x346**	0x04b	**0x3c6**	0x121
0x047	0x02d	**0x0c7**	0x31c	**0x147**	0x0ac	**0x1c7**	0x3e6	**0x247**	0x23a	**0x2c7**	0x2d1	**0x347**	0x2a9	**0x3c7**	0x283
0x048	0x3f8	**0x0c8**	0x3ae	**0x148**	0x1d6	**0x1c8**	0x26d	**0x248**	0x0df	**0x2c8**	0x08f	**0x348**	0x296	**0x3c8**	0x275
0x049	0x117	**0x0c9**	0x252	**0x149**	0x3ee	**0x1c9**	0x041	**0x249**	0x29d	**0x2c9**	0x182	**0x349**	0x1aa	**0x3c9**	0x0eb
0x04a	0x1a4	**0x0ca**	0x3e2	**0x14a**	0x0ea	**0x1ca**	0x3d0	**0x24a**	0x029	**0x2ca**	0x217	**0x34a**	0x14b	**0x3ca**	0x24c
0x04b	0x385	**0x0cb**	0x15d	**0x14b**	0x2fe	**0x1cb**	0x3d3	**0x24b**	0x065	**0x2cb**	0x1af	**0x34b**	0x2e3	**0x3cb**	0x128
0x04c	0x1d3	**0x0cc**	0x2f4	**0x14c**	0x066	**0x1cc**	0x1ff	**0x24c**	0x133	**0x2cc**	0x107	**0x34c**	0x1c8	**0x3cc**	0x312
0x04d	0x3b2	**0x0cd**	0x290	**0x14d**	0x13e	**0x1cd**	0x072	**0x24d**	0x338	**0x2cd**	0x2ae	**0x34d**	0x1e9	**0x3cd**	0x347
0x04e	0x20d	**0x0ce**	0x138	**0x14e**	0x21c	**0x1ce**	0x1c5	**0x24e**	0x1b6	**0x2ce**	0x0f3	**0x34e**	0x3f0	**0x3ce**	0x087
0x04f	0x0b2	**0x0cf**	0x27b	**0x14f**	0x1c2	**0x1cf**	0x395	**0x24f**	0x019	**0x2cf**	0x31b	**0x34f**	0x164	**0x3cf**	0x1f7
0x050	0x0c5	**0x0d0**	0x32a	**0x150**	0x1df	**0x1d0**	0x33d	**0x250**	0x17f	**0x2d0**	0x1ef	**0x350**	0x08e	**0x3d0**	0x227
0x051	0x3cb	**0x0d1**	0x2e4	**0x151**	0x081	**0x1d1**	0x3e5	**0x251**	0x1c7	**0x2d1**	0x1a6	**0x351**	0x26b	**0x3d1**	0x2f6
0x052	0x3e3	**0x0d2**	0x2c8	**0x152**	0x174	**0x1d2**	0x3fc	**0x252**	0x12e	**0x2d2**	0x086	**0x352**	0x2c4	**0x3d2**	0x173
0x053	0x29f	**0x0d3**	0x3bd	**0x153**	0x350	**0x1d3**	0x304	**0x253**	0x171	**0x2d3**	0x0f5	**0x353**	0x2b3	**0x3d3**	0x2d2
0x054	0x2e7	**0x0d4**	0x0b4	**0x154**	0x25c	**0x1d4**	0x097	**0x254**	0x151	**0x2d4**	0x1cc	**0x354**	0x2f8	**0x3d4**	0x2de
0x055	0x0a6	**0x0d5**	0x06f	**0x155**	0x235	**0x1d5**	0x109	**0x255**	0x106	**0x2d5**	0x2bf	**0x355**	0x3bb	**0x3d5**	0x39a
0x056	0x273	**0x0d6**	0x3a3	**0x156**	0x07a	**0x1d6**	0x155	**0x256**	0x341	**0x2d6**	0x022	**0x356**	0x1a7	**0x3d6**	0x3b0
0x057	0x29c	**0x0d7**	0x253	**0x157**	0x2ea	**0x1d7**	0x046	**0x257**	0x21a	**0x2d7**	0x18b	**0x357**	0x3de	**0x3d7**	0x327
0x058	0x163	**0x0d8**	0x0bd	**0x158**	0x13d	**0x1d8**	0x0dd	**0x258**	0x048	**0x2d8**	0x3f1	**0x358**	0x3a4	**0x3d8**	0x119
0x059	0x210	**0x0d9**	0x1c3	**0x159**	0x09b	**0x1d9**	0x289	**0x259**	0x381	**0x2d9**	0x0fc	**0x359**	0x080	**0x3d9**	0x23c
0x05a	0x01d	**0x0da**	0x12b	**0x15a**	0x30c	**0x1da**	0x14c	**0x25a**	0x286	**0x2da**	0x3a5	**0x35a**	0x3db	**0x3da**	0x0e9
0x05b	0x083	**0x0db**	0x16a	**0x15b**	0x2cf	**0x1db**	0x0cd	**0x25b**	0x28f	**0x2db**	0x2cc	**0x35b**	0x250	**0x3db**	0x373
0x05c	0x19e	**0x0dc**	0x349	**0x15c**	0x2c6	**0x1dc**	0x1ac	**0x25c**	0x0a7	**0x2dc**	0x13b	**0x35c**	0x141	**0x3dc**	0x063
0x05d	0x197	**0x0dd**	0x14a	**0x15d**	0x00a	**0x1dd**	0x1ce	**0x25d**	0x10b	**0x2dd**	0x104	**0x35d**	0x37b	**0x3dd**	0x26a
0x05e	0x279	**0x0de**	0x3f6	**0x15e**	0x0fb	**0x1de**	0x261	**0x25e**	0x39f	**0x2de**	0x0f1	**0x35e**	0x1c6	**0x3de**	0x1f6
0x05f	0x332	**0x0df**	0x17c	**0x15f**	0x047	**0x1df**	0x224	**0x25f**	0x317	**0x2df**	0x3f7	**0x35f**	0x1e2	**0x3df**	0x325

input	output	**input**	output	**input**	output	**input**	output	**input**	output	**input**	output	**input**	output	**input**	output
0x060	0x2d0	**0x0e0**	0x0a9	**0x160**	0x297	**0x1e0**	0x184	**0x260**	0x246	**0x2e0**	0x139	**0x360**	0x22c	**0x3e0**	0x008
0x061	0x389	**0x0e1**	0x031	**0x161**	0x1f8	**0x1e1**	0x19b	**0x261**	0x308	**0x2e1**	0x124	**0x361**	0x134	**0x3e1**	0x23e
0x062	0x3b7	**0x0e2**	0x18e	**0x162**	0x0da	**0x1e2**	0x12f	**0x262**	0x34b	**0x2e2**	0x190	**0x362**	0x1f1	**0x3e2**	0x25e
0x063	0x388	**0x0e3**	0x2fb	**0x163**	0x3c4	**0x1e3**	0x305	**0x263**	0x135	**0x2e3**	0x2f2	**0x363**	0x343	**0x3e3**	0x287
0x064	0x2dd	**0x0e4**	0x3c3	**0x164**	0x3f3	**0x1e4**	0x2ff	**0x264**	0x02b	**0x2e4**	0x07e	**0x364**	0x054	**0x3e4**	0x1de
0x065	0x18d	**0x0e5**	0x021	**0x165**	0x2a7	**0x1e5**	0x05c	**0x265**	0x288	**0x2e5**	0x103	**0x365**	0x1e1	**0x3e5**	0x2e9
0x066	0x053	**0x0e6**	0x037	**0x166**	0x229	**0x1e6**	0x017	**0x266**	0x01c	**0x2e6**	0x0f4	**0x366**	0x093	**0x3e6**	0x337
0x067	0x37d	**0x0e7**	0x07b	**0x167**	0x3ab	**0x1e7**	0x24d	**0x267**	0x039	**0x2e7**	0x394	**0x367**	0x344	**0x3e7**	0x016
0x068	0x352	**0x0e8**	0x302	**0x168**	0x1a5	**0x1e8**	0x3ad	**0x268**	0x0b0	**0x2e8**	0x2d3	**0x368**	0x3c8	**0x3e8**	0x35e
0x069	0x146	**0x0e9**	0x28c	**0x169**	0x2b4	**0x1e9**	0x334	**0x269**	0x12c	**0x2e9**	0x1ea	**0x369**	0x3d1	**0x3e9**	0x03d
0x06a	0x1b2	**0x0ea**	0x377	**0x16a**	0x295	**0x1ea**	0x22b	**0x26a**	0x061	**0x2ea**	0x0cc	**0x36a**	0x2b1	**0x3ea**	0x369
0x06b	0x3d9	**0x0eb**	0x0e5	**0x16b**	0x31e	**0x1eb**	0x2ee	**0x26b**	0x156	**0x2eb**	0x1d8	**0x36b**	0x2b9	**0x3eb**	0x1ad
0x06c	0x249	**0x0ec**	0x076	**0x16c**	0x26e	**0x1ec**	0x0f9	**0x26c**	0x2b8	**0x2ec**	0x136	**0x36c**	0x0e2	**0x3ec**	0x0fe
0x06d	0x0db	**0x0ed**	0x3fe	**0x16d**	0x02a	**0x1ed**	0x1f4	**0x26d**	0x30f	**0x2ed**	0x284	**0x36d**	0x1dd	**0x3ed**	0x3ce
0x06e	0x2f0	**0x0ee**	0x2ca	**0x16e**	0x130	**0x1ee**	0x2df	**0x26e**	0x014	**0x2ee**	0x27d	**0x36e**	0x387	**0x3ee**	0x024
0x06f	0x3fa	**0x0ef**	0x2dc	**0x16f**	0x3ea	**0x1ef**	0x2ab	**0x26f**	0x2d9	**0x2ef**	0x3ac	**0x36f**	0x31a	**0x3ef**	0x0c3
0x070	0x125	**0x0f0**	0x201	**0x170**	0x21f	**0x1f0**	0x0c7	**0x270**	0x2c1	**0x2f0**	0x0d5	**0x370**	0x0c8	**0x3f0**	0x33c
0x071	0x1e0	**0x0f1**	0x257	**0x171**	0x2ef	**0x1f1**	0x239	**0x271**	0x1f0	**0x2f1**	0x062	**0x371**	0x1a1	**0x3f1**	0x165
0x072	0x06c	**0x0f2**	0x143	**0x172**	0x280	**0x1f2**	0x209	**0x272**	0x3c0	**0x2f2**	0x0bb	**0x372**	0x2a8	**0x3f2**	0x206
0x073	0x3f9	**0x0f3**	0x370	**0x173**	0x0e6	**0x1f3**	0x248	**0x273**	0x0d0	**0x2f3**	0x01b	**0x373**	0x198	**0x3f3**	0x0a0
0x074	0x073	**0x0f4**	0x10d	**0x174**	0x050	**0x1f4**	0x19c	**0x274**	0x267	**0x2f4**	0x018	**0x374**	0x0ee	**0x3f4**	0x04e
0x075	0x1d1	**0x0f5**	0x0b8	**0x175**	0x0c0	**0x1f5**	0x105	**0x275**	0x0e7	**0x2f5**	0x335	**0x375**	0x234	**0x3f5**	0x074
0x076	0x122	**0x0f6**	0x221	**0x176**	0x386	**0x1f6**	0x371	**0x276**	0x30d	**0x2f6**	0x27f	**0x376**	0x1d7	**0x3f6**	0x39c
0x077	0x263	**0x0f7**	0x3ff	**0x177**	0x169	**0x1f7**	0x0e3	**0x277**	0x3a1	**0x2f7**	0x25b	**0x377**	0x2eb	**0x3f7**	0x32b
0x078	0x116	**0x0f8**	0x032	**0x178**	0x2cd	**0x1f8**	0x20c	**0x278**	0x2a1	**0x2f8**	0x034	**0x378**	0x2af	**0x3f8**	0x277
0x079	0x082	**0x0f9**	0x233	**0x179**	0x0a3	**0x1f9**	0x1fd	**0x279**	0x2f1	**0x2f9**	0x1e4	**0x379**	0x398	**0x3f9**	0x1a3
0x07a	0x2b5	**0x0fa**	0x3d7	**0x17a**	0x149	**0x1fa**	0x1fe	**0x27a**	0x316	**0x2fa**	0x18c	**0x37a**	0x378	**0x3fa**	0x11a
0x07b	0x028	**0x0fb**	0x0ab	**0x17b**	0x324	**0x1fb**	0x36d	**0x27b**	0x199	**0x2fb**	0x3f2	**0x37b**	0x375	**0x3fb**	0x31f
0x07c	0x129	**0x0fc**	0x213	**0x17c**	0x19d	**0x1fc**	0x012	**0x27c**	0x397	**0x2fc**	0x2f9	**0x37c**	0x35b	**0x3fc**	0x068
0x07d	0x0b5	**0x0fd**	0x10e	**0x17d**	0x21e	**0x1fd**	0x013	**0x27d**	0x3b5	**0x2fd**	0x07d	**0x37d**	0x09a	**0x3fd**	0x3ed
0x07e	0x348	**0x0fe**	0x22a	**0x17e**	0x09f	**0x1fe**	0x38f	**0x27e**	0x0ce	**0x2fe**	0x359	**0x37e**	0x1ca	**0x3fe**	0x39e
0x07f	0x30e	**0x0ff**	0x17d	**0x17f**	0x383	**0x1ff**	0x100	**0x27f**	0x2ba	**0x2ff**	0x189	**0x37f**	0x28d	**0x3ff**	0x321

References

1. Alekseev, E.K., Karelina, E.K.: Classification of correlation-immune and minimal correlation-immune Boolean functions of 4 and 5 variables. Discrete Mathematics and Applications (2015). https://doi.org/10.1515/dma-2015-0019
2. Arshad, B., Siddiqui, N., Hussain, Z., Ehatisham-Ul-Haq, M.: A novel scheme for designing secure substitution boxes (S-boxes) based on Möbius group and finite field. Wireless Personal Communications (2022). https://doi.org/10.1007/s11277-022-09524-1
3. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: The keccak reference (2011). https://doi.org/10.1007/978-3-642-38348-9_19
4. Biham, E., Shamir, A.: Differential cryptanalysis of the data encryption standard. Springer Science & Business Media (2012). https://doi.org/10.1007/978-1-4613-9314-6
5. Biryukov, A., De Canniere, C.: Linear cryptanalysis for block ciphers. Encyclopedia of cryptography and security (2011). https://doi.org/10.1007/978-1-4419-5906-5_589
6. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J., Seurin, Y., Vikkelsoe, C.: Present: An ultra-lightweight block cipher. In: Cryptographic Hardware and Embedded Systems-CHES: 9th International Workshop, Vienna, Austria. Springer (2007). https://doi.org/10.1007/978-3-540-74735-2_31
7. Budaghyan, L., Carlet, C., Pott, A.: New classes of almost bent and almost perfect nonlinear polynomials. IEEE Transactions on Information Theory (2006). https://doi.org/10.1109/TIT.2005.864481
8. Canteaut, A., Duval, S., Leurent, G.: Construction of lightweight S-boxes using Feistel and MISTY structures. In: International conference on selected areas in cryptography. Springer (2015)
9. Carlet, C., Mesnager, S.: On the supports of the Walsh transforms of Boolean functions. BFCA'05: Boolean Functions: Cryptography and Applications (2005)
10. Carlet, C., Ding, C.: Nonlinearities of S-boxes. Finite fields and their applications (2007). https://doi.org/10.1016/j.ffa.2005.07.003
11. Daemen, J., Govaerts, R., Vandewalle, J.: A framework for the design of one-way hash functions including cryptanalysis of Damgård's one-way function based on a cellular automaton. In: Advances in Cryptology—ASIACRYPT'91: International Conference on the Theory and Application of Cryptology Fujiyosida, Japan. Springer (1993). https://doi.org/10.1007/3-540-57332-1_7
12. Daemen, J., Rijmen, V.: The block cipher Rijndael. In: International Conference on Smart Card Research and Advanced Applications. Springer (1998). https://doi.org/10.1007/10721064_26
13. Daemen, J., Rijmen, V.: Probability distributions of correlation and differentials in block ciphers. Cryptology (2005). https://doi.org/10.1515/JMC.2007.011
14. Dawson, E., Millan, W.: Efficient methods for generating mars-like S-boxes. In: Fast Software Encryption (2000). https://doi.org/10.1007/3-540-44706-7_21
15. Dimitrov, M.M.: On the design of chaos-based S-boxes. IEEE Access (2020). https://doi.org/10.1109/ACCESS.2020.3004526
16. Dobraunig, C., Eichlseder, M., Mendel, F., Schläffer, M.: Ascon v1. 2: Lightweight authenticated encryption and hashing. Journal of Cryptology (2021). https://doi.org/10.1007/s00145-021-09398-9
17. Feistel, H.: Cryptography and computer privacy. Scientific american (1973)

18. Formenti, E., Imai, K., Martin, B., Yunès, J.B.: Advances on random sequence generation by uniform cellular automata. Computing with new resources: essays dedicated to Jozef Gruska on the occasion of his 80th birthday (2014). https://doi.org/10.1007/978-3-319-13350-8_5
19. Gutowitz, H.: Cryptography with dynamical systems. In: Cellular Automata and Cooperative Systems. Springer (1993). https://doi.org/10.1007/978-94-011-1691-6_21
20. Haider, T., Azam, N.A., Hayat, U.: Substitution box generator with enhanced cryptographic properties and minimal computation time. Expert Systems with Applications (2024). https://doi.org/10.1016/j.eswa.2023.122779
21. Hussain, I., Shah, T., Mahmood, H., Gondal, M.A.: Construction of S_8 Liu J S-boxes and their applications. Computers & Mathematics with Applications (2012). https://doi.org/10.1016/j.camwa.2012.05.017
22. Hussain, S., Jamal, S.S., Shah, T., Hussain, I.: A power associative loop structure for the construction of non-linear components of block cipher. IEEE Access (2020). https://doi.org/10.1109/ACCESS.2020.3005087
23. Jakobsen, T., Knudsen, L.R.: Attacks on block ciphers of low algebraic degree. Journal of Cryptology (2001). https://doi.org/10.1007/s00145-001-0003-x
24. Jeon, Y., Baek, S., Kim, H., Kim, G., Kim, J.: Differential uniformity and linearity of S-boxes by multiplicative complexity. Cryptography and Communications (2022). https://doi.org/10.1007/s12095-021-00547-2
25. John, A., Jose, J.: Hash function design based on hybrid five-neighborhood cellular automata and sponge functions. Complex Systems (2023). https://doi.org/10.25088/ComplexSystems.32.2.171
26. Langevin, P., Leander, G.: Counting all bent functions in dimension eight 99270589265934370305785861242880. Designs, Codes and Cryptography (2011). https://doi.org/10.1007/s10623-010-9455-z
27. Li, Y., Wang, M.: Constructing S-boxes for lightweight cryptography with Feistel structure. In: International Workshop on Cryptographic Hardware and Embedded Systems. Springer (2014)
28. Luby, M., Rackoff, C.: How to construct pseudorandom permutations from pseudorandom functions. SIAM Journal on Computing (1988). https://doi.org/10.1007/3-540-39799-X_34
29. Malal, A., Tezcan, C.: FPGA-friendly compact and efficient AES-like 8× 8 S-box. Microprocessors and Microsystems (2024). https://doi.org/10.1016/j.micpro.2024.105007
30. Mariot, L., Picek, S., Leporati, A., Jakobovic, D.: Cellular automata based S-boxes. Cryptography and Communications (2019). https://doi.org/10.1007/s12095-018-0311-8
31. Marochok, S., Zajac, P.: Algorithm for generating S-boxes with prescribed differential properties. Algorithms (2023). https://doi.org/10.3390/a16030157
32. Martin, B.: A Walsh exploration of elementary CA rules. Journal of Cellular Automata (2008)
33. Millan, W.: Low order approximation of cipher functions. In: International Conference on Cryptography: Policy and Algorithms. Springer (1995). https://doi.org/10.1007/BFb0032354
34. Naor, M., Reingold, O.: On the construction of pseudo-random permutations: Luby-Rackoff revisited. In: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing (1997). https://doi.org/10.1007/PL00003817

35. Naseer, M., Tariq, S., Riaz, N., Ahmed, N., Hussain, M.: S-box security analysis of NIST lightweight cryptography candidates: A critical empirical study. arXiv preprint arXiv:2404.06094 (2024). https://doi.org/10.48550/arXiv.2404.0609
36. Nyberg, K., Knudsen, L.R.: Provable security against differential cryptanalysis. In: Annual international cryptology conference. Springer (1992). https://doi.org/10.1007/3-540-48071-4_41
37. Patarin, J.: Luby-Rackoff: 7 rounds are enough for $2n(1-\varepsilon)$ security. In: Advances in Cryptology-CRYPTO: 23rd Annual International Cryptology Conference, Santa Barbara, California, USA. Springer (2003). https://doi.org/10.1007/978-3-540-45146-4_30
38. 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 (2017). https://doi.org/10.1145/3067695.3076084
39. Picek, S., Mariot, L., Yang, B., Jakobovic, D., Mentens, N.: Design of S-boxes defined with cellular automata rules. In: Proceedings of the computing frontiers conference (2017). https://doi.org/10.1145/3075564.3079069
40. Poinsot, L.: Boolean bent functions in impossible cases: odd and plane dimensions. International Journal of Computer Science and Network Security (2006)
41. Preneel, B.: Analysis and design of cryptographic hash functions. Ph.D. thesis, Citeseer (1993)
42. Pub, F.: Security requirements for cryptographic modules. FIPS PUB (1994). https://doi.org/10.6028/NIST.FIPS.140-2
43. Tuncer, T., Avaroğlu, E.: Random number generation with LFSR based stream cipher algorithms. In: 2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO). IEEE (2017)
44. Wagner, D.: The boomerang attack. In: International Workshop on Fast Software Encryption. Springer (1999). https://doi.org/10.1007/3-540-48519-8_12
45. Waheed, A., Subhan, F., Suud, M.M., Alam, M., Ahmad, S.: An analytical review of current S-box design methodologies, performance evaluation criteria, and major challenges. Multimedia Tools and Applications (2023). https://doi.org/10.1007/s11042-023-14910-3
46. Webster, A.F., Tavares, S.E.: On the design of S-Boxes. In: Conference on the theory and application of cryptographic techniques. Springer (1985). https://doi.org/10.1007/3-540-39799-X_41
47. Wolfram, S.: Statistical mechanics of cellular automata. Reviews of modern physics (1983). https://doi.org/10.1103/RevModPhys.55.601
48. Xiao, G.Z., Massey, J.L.: A spectral characterization of correlation-immune combining functions. IEEE Transactions on information theory (1988). https://doi.org/10.1109/18.6037
49. Zahid, A.H., Arshad, M.J.: An innovative design of substitution-boxes using cubic polynomial mapping. Symmetry (2019). https://doi.org/10.3390/sym11030437
50. Zahid, A.H., Rashid, H., Shaban, M.M.U., Ahmad, S., Ahmed, E., Amjad, M.T., Baig, M.A.T., Arshad, M.J., Tariq, M.N., Tariq, M.W.: Dynamic S-box design using a novel square polynomial transformation and permutation. IEEE Access (2021). https://doi.org/10.1109/ACCESS.2021.3086717
51. Zhang, L., Ma, C., Zhao, Y., Zhao, W.: A novel dynamic S-box generation scheme based on quantum random walks controlled by a hyper-chaotic map. Mathematics (2023). https://doi.org/10.3390/math12010084