An Efficient Synchronous Data Scrambling Technique with Polynomial and LFSR State Prediction Models

Jaechan Lee

Received January 05, 2025
Accepted April 28, 2025
Electronic access May 31, 2025

Data bit scrambling is widely used for digital signal processing for display or network device due to its ability to generate a randomized bit stream. This makes it difficult to decipher the original information without the correct polynomial and helps improve the signal quality by preventing concentration of too much signal energy in a single frequency. Additionally, scrambling provides a known bit sequence for data synchronization and maintains the number of bit transitions, which helps flatten the signal power spectrum. In the transmitter, the scrambler randomizes the original data using a Linear Feedback Shift Register (LFSR) with a specified polynomial. The receiver's descrambler uses the same polynomial to recover the scrambled data. However, if an inappropriate shock, such as electrostatic discharge (ESD), is applied to the receiver, the LFSR output sequence becomes distorted. This leads to massive data errors, and synchronization between the scrambler and descrambler is lost until both are reset. This paper proposes a function model capable of generating a new polynomial for scrambled data error recovery. The generated polynomial allows the descrambler to compute the LFSR state value for the future symbol pattern in advance, without requiring LFSR operations and initialization. Since the LFSR pattern cannot be restored once it is corrupted, the computed state replaces the existing state for the specific data sequence, enabling the LFSR to generate the correct patterns from the new state. This method facilitates rapid recovery of LFSR synchronization between two devices, minimizing data errors.

Introduction

Scrambling is a logic operation that transforms a signal bit stream into a randomized sequence. The transmitter scrambles the data before transmission, and the receiver uses the same algorithm to descramble and recover the original data. In a transmission system, the receiver may struggle to recover data if problematic bit patterns, such as long strings of zeros, are transmitted. Maintaining a balance between binary zeros and ones is crucial for the transmitter to provide sufficient bit transitions, which are essential for the receiver to recover the data. However, in real-world scenarios, most data are not inherently random. Scrambling ensures that problematic patterns are evenly randomized, spreading and its signal density and increasing noise tolerance. Furthermore, it enhances security by making recovery impossible unless the receiver uses the same scrambling algorithm as the transmitter. Figure 1 illustrates the basic LFSR based scrambling scheme.

LFSR in the scrambler generates pseudo-random bit patterns, which follow a set pattern, rather than being completely random[1]. It consists of linearly connected n registers $X_n \in \{0,1\}$ and a set of coefficients $C_n \in \{0,1\}$, and its feedback function computes the new state of the LFSR using modulo-2 additions and multiplications based on the initial state value of the shift registers[2]. The next input binary value in a shift register is a linear function of its previous state. We can characterize the

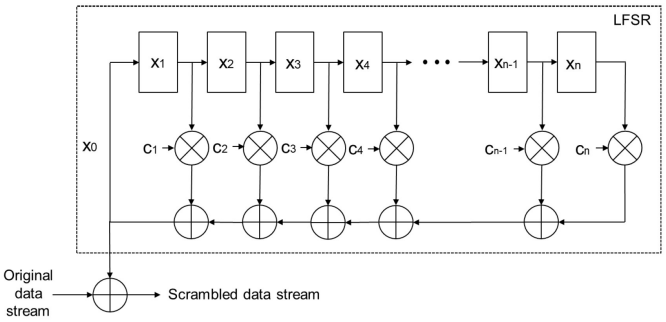

Fig. 1 LFSR based scrambling of the original data stream

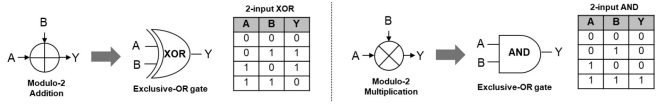

Fig. 2 Modulo-2 Arithmetic Operation

LFSR by defining a primitive polynomial as:

$$P(x) = 1 + C_1x^1 + C_2x^2 + \cdots + C_{n-1}x^{n-1} + C_nx^n$$

Figure 2 illustrates the Modulo-2 operation. In this operation,

addition is performed using XOR (exclusive OR) operation and the multiplication is done using the AND operation. These operations are performed on binary numbers, digit by digit. Each digit is processed independently and no carry is involved in the calculation. For example, consider the polynomial $x^4 + x^3 + 1$, with the coefficient vector $(c_1, c_2, c_3, c_4) = (0, 0, 1, 1)$. If the initial register state is $(x_1, x_2, x_3, x_4) = (0, 1, 0, 1)$, the new state x_0 can be computed as:

$$X_0 = 0 \cdot X_1 + 0 \cdot X_2 + 1 \cdot X_3 + 1 \cdot X_4$$

$$= 0 \cdot 0 + 0 \cdot 1 + 1 \cdot 0 + 1 \cdot 1 = 0$$

Thus, the new state x_0 is 0. The new state is fed back to x_1 and all current states are shifted, resulting in $(x_1, x_2, x_3, x_4) = (0, 0, 1, 0)$. The subsequent state can be computed as:

$$X_0 = 0 \cdot 0 + 0 \cdot 0 + 1 \cdot 1 + 1 \cdot 0 = 1$$

This process generates a sequence that periodically repeats with a fixed period. However, if the logical structure is well organized, the LFSR can generate a sequence over a very long period that appears random [3]. The randomness of the LFSR is determined by both the length of the register and the polynomial used, which defines which register states are involved in the XOR operation [4,5]. The output of the LFSR produces a random bit sequence that repeats at periodic intervals $2^n - 1$, where n is the highest degree of the polynomial. For example, if the maximum degree of the polynomial is 8, the period of the output bit sequence is $2^8 - 1$ (255) [1]. The LFSR starts with an initial value, called seed. Different initial seeds may produce different random sequences, so the seed value is very important for the synchronization between the transmitter's scrambler and receiver's descrambler.

In the transmitter, the scrambler performs an XOR operation between the original bit stream and the LFSR output to generate a scrambled bit stream. Recovery at the receiver is straightforward. As shown in the logical formula below, the descrambler in the receiver performs an XOR operation between the scrambled bit stream Z and its own LFSR output Y. Since both the initial seed and the polynomial are identical on both sides, the LFSR output will be the same, and the descrambled data will match the original bit stream X:

$$X \oplus Y = Z, \quad Z \oplus Y = X$$

This descrambler design has one major drawback. As the article introduced, due to phenomena such as ESD, the descrambler's LFSR itself in the receiver can sometimes lose synchronization with the scrambler, causing the descrambler to generate a large burst of errors [6].

Figure 3 shows an example of how ESD affects the descrambling function causing burst errors in digital logic. The scrambler in the transmitter generates scrambled data S_d based on the

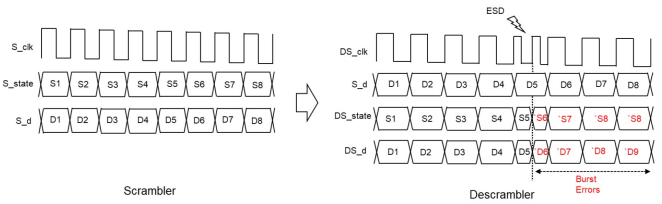

Fig. 3 Synchronization Loss Example on the Descrambler due to ESD

LFSR seed S_state in every S_clk cycle. The generated scrambled data is transmitted to the receiver side and the descrambler recovers the original data SD_d by performing an XOR operation on each data. ESD can occur during the descrambling process and can cause unwanted LFSR state 'S6 to be generated due to the distorted clock DS_clk. Error data 'D6 descrambled by 'S6 can occur and the receiver loses the LFSR state synchronization causing burst errors. When a synchronization failure occurs, a large burst of errors will persist until the scrambling system is reset. Unlike bit errors caused by channel noise or equalization during data recovery, this issue arises from completely incorrect data generated inside the receiver block. As a result, hundreds or even thousands of data symbols are continuously corrupted until synchronization is restored. These burst errors cannot be corrected by error correction blocks like Reed-Solomon or Viterbi. Reed-Solomon, for example, can only correct a limited number of symbol errors [7], and the number of errors can easily exceed this limit. Additionally, since the descrambler generates entirely different symbol data, burst errors cannot be recovered by bit stream decoders like Viterbi [8]. Because the bit stream is completely different from the original (not caused by channel noise), and the number of burst errors is too large, the decoder cannot detect the correct pattern. To prevent massive burst errors, the transmitter should periodically send sync symbols to restart the LFSR. Once the sync symbol is received, the descrambler is reset. Since both the transmitter and receiver LFSRs are reset with the same sync symbol, they can produce identical LFSR outputs thereafter. However, if display devices that transmit and receives video images only send theses sync symbols every few frames, and the synchronization is broken, which can result in several broken frames and cause flickering on the screen. Additionally, if the receiver misses the sync symbols, it may be in an even worse condition. The other article introduces a different type of scrambling technique which is a self-synchronous scrambler [9]. This method uses the scrambled bit stream as a seed, allowing for quick recovery when the LFSR fails to generate the random sequence. However, even a single bit error can lead to multiple bit errors since the bit error is used as the seed in the LFSR. We can also think about a seed estimator that can extract the seed when the input scrambled data consists of bursts of zeros, such as those for the display device. If the original N-bit stream

is all zero, the scrambled N-bit data will match the seed, and the descrambler can use this as the new seed for the LFSR to recover subsequent data. It is crucial for the system to detect when certain bit streams are entirely all zeros. While all video frames have a blank period with zero data, secondary data like video control or sound can be transmitted during this period. In such case, receiving zeros may not be possible if sound data is added to the video stream. In this paper, we present a method to directly compute the future LFSR state from the previous LFSR state without shifting operation. This is achieved by creating a new polynomial function that can calculate the desired number of shift steps at once. When the descrambler loses the synchronization with the scrambler, the LFSR state at a specific future location can be computed using a polynomial prediction model. This computed state can then replace the LFSR state. From this point, the LFSR with the new state can continue to generate correct random pattern, allowing the descrambler to function properly without needing to wait for the sync symbol from the scrambler. This approach minimizes the bit errors and eliminates the need for the LFSR to be reset.

Materials and Methods

Scrambler Model

Polynomial Decision

To determine the degree of the polynomial required for implementing the algorithm, the maximum run length of the polynomials was considered. Table 1 shows the maximum run length for irreducible primitive polynomials of various degrees.

Table 1: Comparison of the LFSR maximum run-length

Highest Degree (N)	Maximum Run Length ($M = 2^N$-1)
8	255
9	511
10	1023
11	2047
12	4095
13	8191
14	16383
15	32767
16	65535
17	131071
18	262143

To ensure sufficient randomness when scrambling a test image, the maximum period of the LFSR should not repeat more than twice while scrambling the bit stream of a horizontal line in the image. For example, in a Full HD (1920x1080) 24-bit pixel image, one horizontal line consists of a 46,080-bit sequence. If the polynomial's degree is 8, the randomized sequence will repeat every 255 bits, meaning randomness cannot be guaranteed. Therefore, we concluded that the degree must be at least 16 to maintain randomness considering maximum horizontal line period. Based on this, we reviewed several primitive polynomials with degrees ranging from 16 to 24 in relevant literature [10,11], and conducted an analysis of randomness and hardware cost, as shown in Table 2.

Table 2: Comparison of the different degrees of polynomials

Irreducible Polynomial	Degree(N)	Run Length ($M = 2^N$-1)	Frequency Test (P-value)	Entropy Test	Hardware Cost	Computation Speed
$x^{16}+x^5+x^4+x^3+1$	16	65535	0.7584	0.9999985	100%	100%
$x^{17}+x^{14}+1$	17	131071	0.5889	0.9999954	106%	94%
$x^{18}+x^{11}+1$	18	262143	0.7093	0.9999978	113%	88%
$x^{19}+x^{18}+x^{17}+x^{14}+1$	19	524287	0.9925	0.9999999	119%	84%
$x^{20}+x^{17}+1$	20	1048575	0.8814	0.9999996	125%	80%
$x^{21}+x^{19}+1$	21	2097151	0.0451	0.9999371	131%	76%
$x^{22}+x^{1}+1$	22	4194303	0.0917	0.9999554	137%	72%
$x^{23}+x^{18}+1$	23	8388607	0.4229	0.9999899	144%	69%
$x^{24}+x^{23}+x^{22}+x^{17}+1$	24	16777215	0.8741	0.9999996	150%	66%

All the polynomials in Table 2 are irreducible primitive polynomials, meaning they have a maximum run length of $2^N - 1$, where N is the highest degree of the given polynomial. Two statistical measurements, frequency and entropy tests, were conducted to assess LFSR's randomness. Frequency test measures the frequency of 0's and 1's in a random bit stream. The p-value represents the probability that the chosen test statistic will assume values that are equal to or worse than the observed test statistic value assuming the null hypothesis is true. For the frequency test of a random bit sequence, an ideal p-value should be greater than 0.01 to indicate a statistical randomness [12–14]. It measures how unpredictable or random a sequence is, reflecting the uncertainty or unpredictability of the information contained within it. Entropy is typically calculated using Shannon entropy. For a sequence where 0's and 1's appear equally, the entropy will be close to its maximum value of 1, indicating a completely random sequence [12–14]. Conversely, a low entropy value suggests that the sequence is more predictable. These two statistical metrics were performed using Python script for each polynomial. Comparing the results of the frequency and entropy tests, we see that they show a sufficient level of randomness. However, regarding hardware cost, each time the polynomial degree increases by 1 bit, the shift register, memory, and other combinational logic required for the proposed model also increase proportionally. For example, each increase in the polynomial degree by 1-bit results in an increase of 1 flip-flop in memory size for storing the LFSR state, the polynomial prediction model, and the initial polynomial, respectively. This also affects the computation time for the prediction model due to the increased

logic, as shown in Table 2. However, the increase in memory does not affect the recovery performance of the proposed algorithm. In fact, the synchronization failure discussed in this paper is not caused by the input data from the scrambler but rather by external factors such as electrostatic discharge (ESD), which generates false clock pulses that disrupt the synchronization of the descrambler inside the receiver. Therefore, synchronization failure due to ESD occurs regardless of the polynomial degree, and the proposed algorithm can be modeled using any irreducible binary polynomial. The polynomial primarily affects hardware complexity and computation time for modeling. Based on the test results, we chose a polynomial degree of 16 to provide sufficient randomness while minimizing hardware costs. To select the 16-degree polynomial for this paper, three irreducible polynomials from the literature [10,11], along with two non-irreducible polynomials, were tested and compared. Table 3 shows five polynomials, three of which are irreducible primitive polynomials.

Table 3: Comparison of the 16-degree polynomials

Polynomial	Degree(N)	Irreducible Polynomial	Maximum Run Length (M = 2^N-1)	Frequency Test (P-value)	Entropy Test
$x^{16}+x^5+x^4+x^3+1$	16	Yes	65535	0.7584	0.9999985
$x^{16}+x^{15}+1$	16	No	255	0.0000	0.8972440
$x^{16}+x^{14}+x^{13}+x^{11}+1$	16	Yes	65535	0.2635	0.9999804
$x^{16}+x^{14}+x^{12}+x^7+x^6+x^4+x^2+x^1+1$	16	Yes	65535	0.8521	0.9999994
$x^{16}+x^{14}+x^{12}+x^8+1$	16	No	7665	0.0000	0.9182958

The maximum run length was first tested using a Python script. When the polynomial is irreducible, the maximum period of $2^n - 1$ is consistently achieved, regardless of the number of XORs. Both the first and third polynomials in the table are irreducible and involve the same number of XORs, yet the first polynomial exhibits better randomness properties. This shows that the feedback function plays an important role in determining randomness. If the feedback function of the polynomial is poorly defined, the maximum period is not met, regardless of the number of XORs used. The results of the frequency test and entropy test for the non-irreducible polynomial show that the bit sequence generated by this polynomial has very poor randomness. Furthermore, while increasing the number of XORs in a primitive irreducible polynomial can enhance randomness, it also increases computational complexity and hardware costs. In this study, to balance adequate randomness with minimized cost, we chose to use the first polynomial in the table, $x^{16} + x^5 + x^4 + x^3 + 1$.

Implementation and Randomness Test for Scrambler

The selected LFSR generates a $2^{16} - 1$ random sequence and requires three modulo-2 additions with 16 shift registers. Figure 4 illustrates the logic diagram for the implemented scrambler. Given that the polynomial's highest degree is 16, the LFSR requires 16 shift registers and 3 two-input XOR gates. An additional XOR gate is used to scramble the original input. The bit stream is continuously fed into the LFSR to generate the scrambled data, with each register state shifting from one to the next while the scrambler remains active. The outputs from the 16th, 5th, 4th, and 3th registers are combined via XOR and fed back into the first shift register. This scrambler model requires an initialization seed, which is set to all ones.

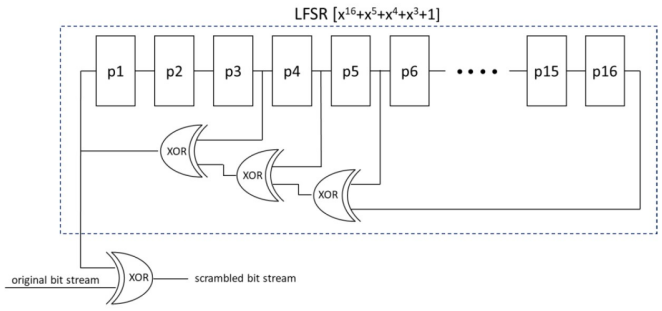

Fig. 4 Implemented scrambler model diagram

After selecting the binary polynomial, we tested the randomness of the scrambler output. In practice, most data streams exhibited periodic patterns. Power spectrum analysis, a technique used to detect periodicity in data streams, is effective for identifying such patterns. Also known as spectral or frequency domain analysis, this method involves transforming the data stream from the time domain to the frequency domain. Fast Fourier Transform (FFT) is commonly used for this transformation. This technique enables us to examine the frequency characteristics of the data stream and assess the energy distribution across different frequencies. For our analysis, we utilized the FFT function provided by Python, based on methodologies outlined in some articles [15,16]. As explained before, the scrambler should reduce the peak signal energy of the original data and distribute the energy over a broad frequency range. To evaluate its effectiveness, the designated scrambler was tested with a sample grey bar image as shown in Figure 5. The test image is a 640 x 480 resolution with 24bit RGB per pixel.

The total number of binary data is 7,373,800 bits. Using Python script, this image was converted into a binary bit stream, which was then analyzed using power spectrum function. This method for plotting the image using Python was introduced in the article [17]. Figure 6 displays the power spectrum of the original grey bar image data. As anticipated, the peak energy of the original data is concentrated at specific frequencies. In contrast, Figure 7 shows the power spectrum of the LFSR output. Since the LFSR generates a pseudo-random bit sequence, the signal energy is almost evenly distributed across a wide frequency range, resembling white noise. Figure 8 shows how the scrambled signal energy is flattened and spread over a broad frequency range by randomizing the original data through XOR

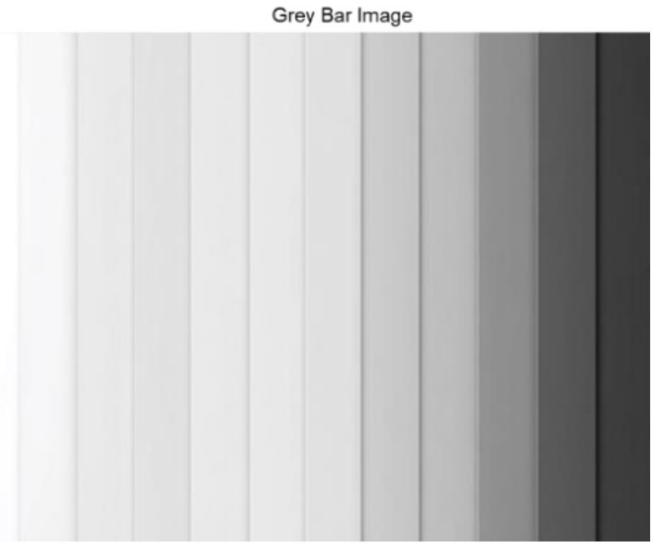

Fig. 5 Test image for power spectrum analysis

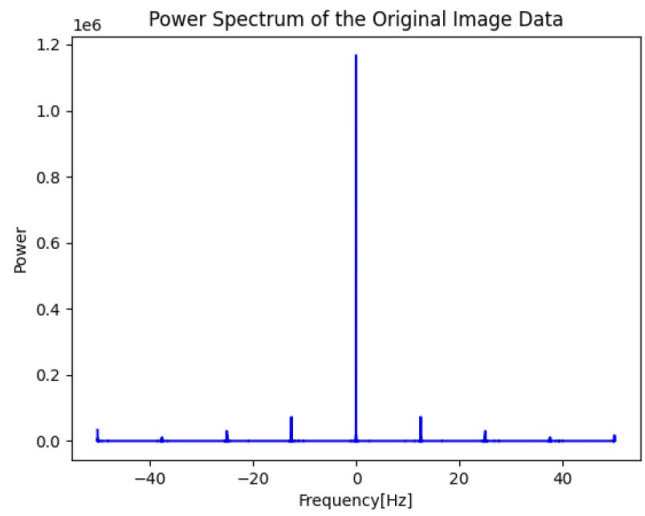

Fig. 6 Spectrum for the original test image

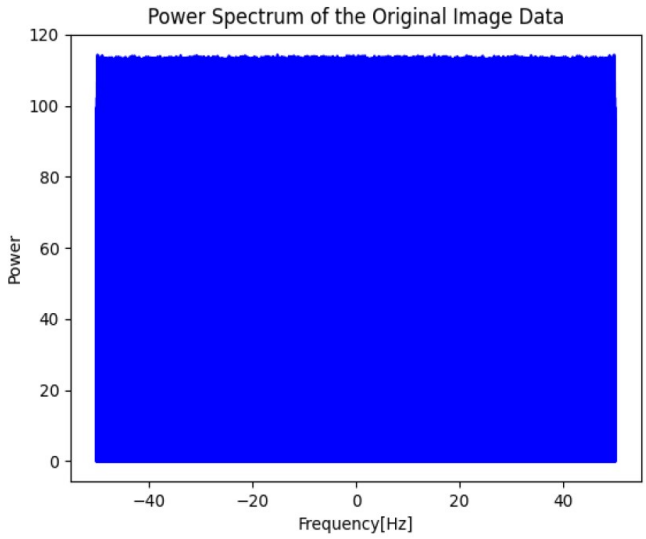

Fig. 7 Spectrum for the LFSR output

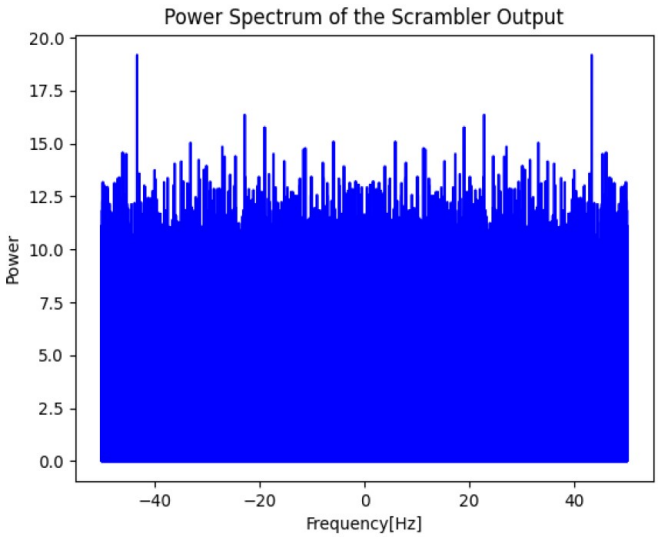

Fig. 8 Spectrum for the scrambled image

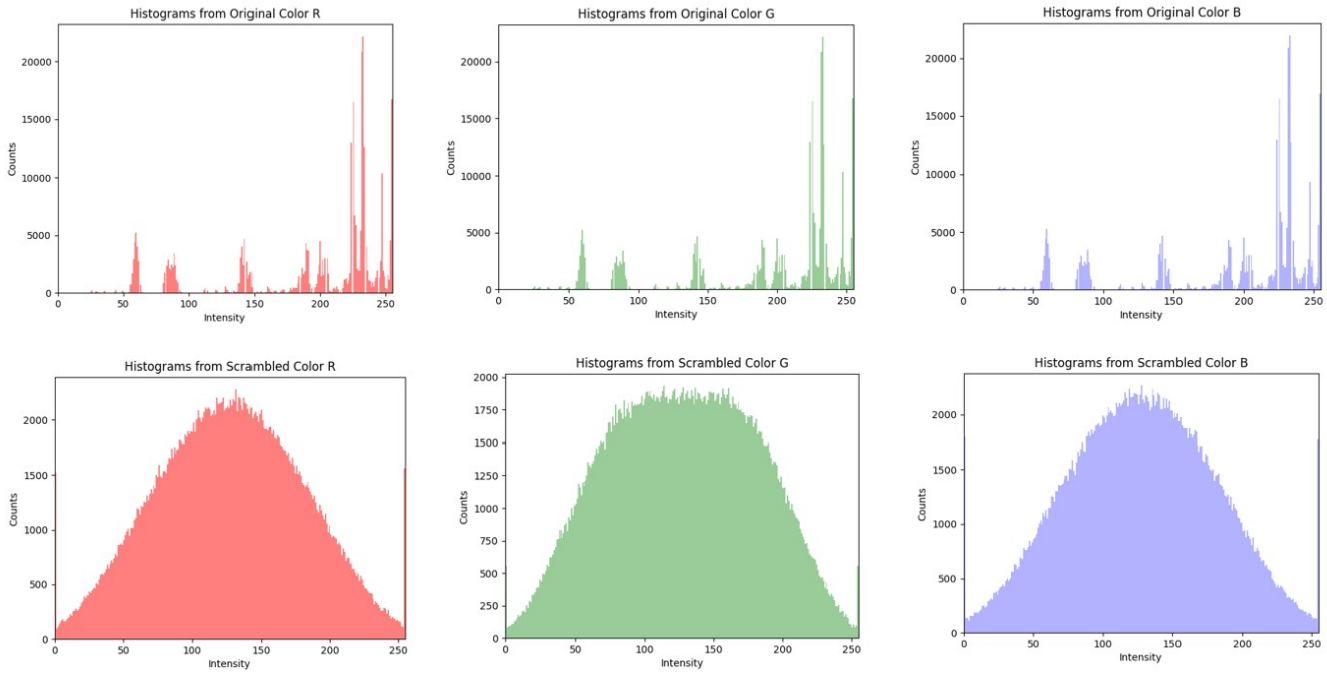

Fig. 9 Histograms of the pixels of each color in the original and scrambled test image

operations with the LFSR.

To assess the randomness of the scrambled image, image entropy was calculated using Shannon entropy, a widely used measure of randomness or uncertainty. Applying Shannon entropy to an image allows us to evaluate its information content or complexity, with higher entropy indicating a more complex image. The entropy was computed using the Shannon entropy function from Python's *skimage* library. Entropy is determined based on the probability distribution of pixel values within the image. According to existing research, the ideal information entropy for grayscale images is 8. The entropy values measured for the test images are shown in Table 4. Compared to the original test image, the entropy of the scrambled image is closer to the ideal value of 8, indicating improved randomness [18].

Table 4: Entropy test results of the original and scrambled test image

Polynomial	Degree(N)	Irreducible Polynomial	Maximum Run Length ($M = 2^N$-1)	Frequency Test (P-value)	Entropy Test
$x^{16}+x^5+x^4+x^3+1$	16	Yes	65535	0.7584	0.9999985
$x^{16}+x^{15}+1$	16	No	255	0.0000	0.8972440
$x^{16}+x^{14}+x^{13}+x^{11}+1$	16	Yes	65535	0.2635	0.9999804
$x^{16}+x^{14}+x^{12}+x^7+x^6+x^4+x^2+x^1+1$	16	Yes	65535	0.8521	0.9999994
$x^{16}+x^{14}+x^{12}+x^8+1$	16	No	7665	0.0000	0.9182958

The pixel value distribution across different image locations was also analyzed [18]. Figure 10 illustrates the pixel value distribution of the grey bar test image along a horizontal line at each pixel location. The results show that in the original image, the pixel values are concentrated around specific values. In contrast, the scrambled image exhibits a more random distribution of pixel values across the horizontal line, with no visible concentration at any particular location.

Polynomial Prediction Model

An LFSR operates on an n-bit binary state vector S, which can be expressed as:

$$S_n = \{B_0, B_1, B_2, B_3, \ldots, B_{n-2}, B_{n-1}\}$$

The next vector component B_n is computed as the XOR sum of the products of the corresponding coefficients:

$$B_n = B_0C_0 + B_1C_1 + B_2C_2 + \cdots + B_{n-2}C_{n-2} + B_{n-1}C_{n-1}$$

The next LFSR state S_{n+1} can be obtained by replacing the first component of the state vector with the updated value B_n. The next vector component B_{n+1} is calculated by applying the updated state S_{n+1}:

$$B_{n+1} = B_nC_0 + B_0C_1 + B_1C_2 + \cdots + B_{n-3}C_{n-2} + B_{n-2}C_{n-1}$$

The update can be expressed as:

$$B_n = (B_0 + \sum_{k=1}^{n-1}(C_kB_k)) \mod 2$$

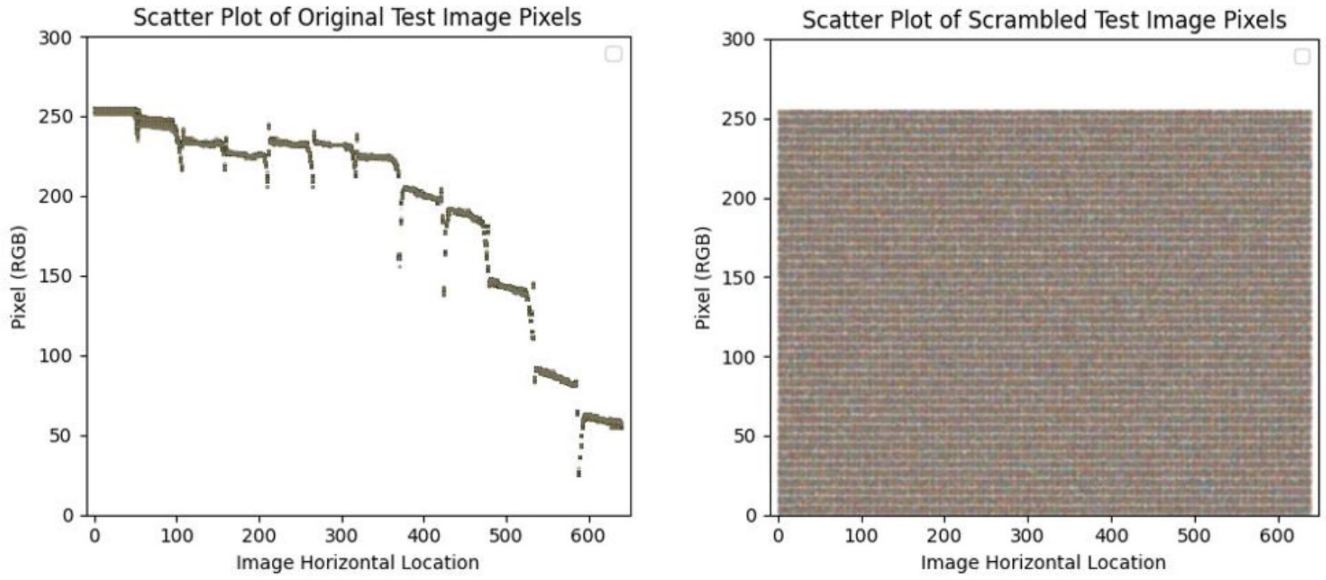

Fig. 10 Distribution of the pixel values in horizontal direction

Using the coefficient vector C of the given polynomial function P and the current LFSR state S_n, the relationship can be simplified into the following formula:

$$P(S_n) = S_{n+1}$$

Using this formula, the future states S_{n+2} and S_{n+3} can be directly obtained:

$$P(S_{n+1}) = P(P(S_n)) = S_{n+2}$$

$$P(S_{n+2}) = P(P(P(S_n))) = S_{n+3}$$

Based on the relationship in these formulas, the future M-bit LFSR state S_{n+m+1} can be predicted. The new polynomial $G(S_n)$ is defined as:

$$P^M(P(S_n)) = G(S_n) = S_{n+m+1}$$

The predicted polynomial model G is useful in situations where scrambling synchronization is lost. For example, once the new polynomial G is computed and the M-bit previous LFSR state is stored, the next LFSR state S_{n+1} can be immediately recovered without shifting operation:

$$G(S_{n-m}) = S_{n+1}$$

The current LFSR state is typically generated by shifting the previous state and performing the XOR operation with the coefficients of the given polynomial. However, in this study, we found that the future state can also be directly computed prior to the shift operation by aligning the polynomial's coefficients with the corresponding positions in the previous state.

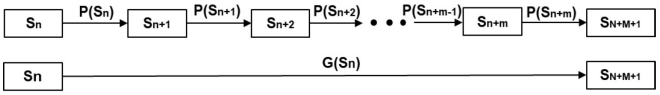

Fig. 11 Seed generation comparison between traditional and proposed model

By repeating this process n-bit times, a new polynomial can be generated that directly computes the future state, which requires n-bit shifts from the current state as shown in Figure 11.

Figure 12 illustrates an example computation of the predicted polynomial G.

The initial polynomial $P(0)$ is represented as a coefficient vector $(p_1, p_2, p_3, p_4, \ldots, p_{15}, p_{16})$, where each p_i is a binary value indicating the position where an XOR operation is required (with 1 for XOR and 0 for no operation). In this computation, $n = 0$ and $m = 5$. It is important to note that p_1 represents the feedback value from the XOR operation in the initial polynomial. Although it may seem that p_1 should always trigger an XOR with the initial polynomial, if $p_1 = 0$, the result of the initial polynomial is effectively ignored. Therefore, $P(0)$ is only obtained when $p_1 = 1$, which simplifies the design process by eliminating unnecessary operations. Figure 12 can be expressed by the following formula.

In an XOR operation, if both inputs are the same, the result is always 0. As a result, the formula can be simplified by eliminating the terms corresponding to overlapping polynomials. The highest degree of the predicted polynomial remains always less

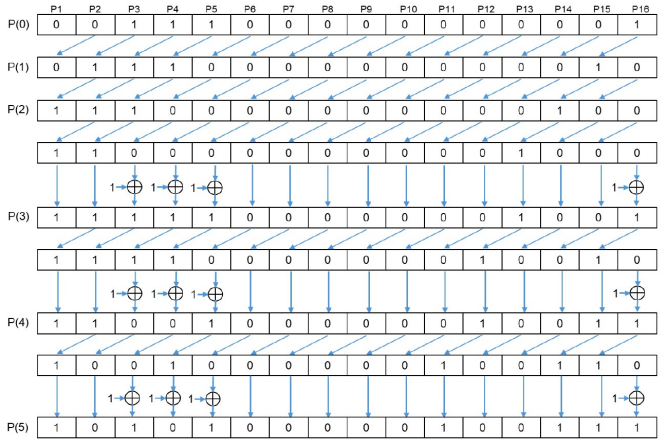

Fig. 12 Computation Example for the predicted polynomial G

$$
\begin{aligned}
P(0) &= P_{16}\text{^}P_5\text{^}P_4\text{^}P_3 \\
P(1) &= P_{15}\text{^}P_4\text{^}P_3\text{^}P_2 \\
P(2) &= P_{14}\text{^}P_3\text{^}P_2\text{^}P_1 \\
P(3) &= P_{13}\text{^}P_2\text{^}P_1\text{^}P(0) \\
&= P_{16}\text{^}P_{13}\text{^}P_5\text{^}P_4\text{^}P_3\text{^}P_2\text{^}P_1 \\
P(4) &= P_{15}\text{^}P_{12}\text{^}P_4\text{^}P_3\text{^}P_2\text{^}P_1\text{^}P(0) \\
&= P_{16}\text{^}P_{15}\text{^}P_{12}\text{^}P_5\text{^}P_4\text{^}P_4\text{^}P_3\text{^}P_3\text{^}P_2\text{^}P_1 \\
&= P_{16}\text{^}P_{15}\text{^}P_{12}\text{^}P_5\text{^}P_2\text{^}P_1 \\
P(5) &= P_{15}\text{^}P_{14}\text{^}P_{11}\text{^}P_4\text{^}P_1\text{^}P(0) \\
&= P_{16}\text{^}P_{15}\text{^}P_{14}\text{^}P_{11}\text{^}P_5\text{^}P_4\text{^}P_4\text{^}P_3\text{^}P_1 \\
&= P_{16}\text{^}P_{15}\text{^}P_{14}\text{^}P_{11}\text{^}P_5\text{^}P_3\text{^}P_1
\end{aligned}
$$

or equal to 16 even though we shift the polynomial infinitely.

We implemented this polynomial prediction model to obtain the LFSR states for the desired number of shift steps using Python's loop function. To maintain the integrity of the polynomial prediction model G, it must be generated and stored in memory during a time period in which no bit errors occur. This requires that the computation proceeds only in regions where the error detection block does not indicate any errors. If an error is detected during the process, the system should be designed to restart the computation.

LFSR State Prediction Model

The highest degree of the polynomial used in the design is 16, so the initial seed must also be 16 bits to initiate the LFSR. The 16-bit seed, after the LFSR has been shifted n-bit times, is defined as S_n. The state vector S_n can be represented as a 16-bit LFSR state vector as shown below. The predicted polynomial G is represented as a coefficient vector, as follow. S_0 denotes the initial seed, which must be specified in the design specification.

$$S = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}, s_{13}, s_{14}, s_{15}, s_{16}\}$$

$$G = \{g_1, g_2, g_3, g_4, g_5, g_6, g_7, g_8, g_9, g_{10}, g_{11}, g_{12}, g_{13}, g_{14}, g_{15}, g_{16}\}$$

The future LFSR state S_{n+m+1} can be computed as:

$$S_{n+m+1} = \sum_{i=1}^{16} g_i s_i$$

$$= g_1 \cdot s_1 + g_2 \cdot s_2 + \cdots + g_{15} \cdot s_{15} + g_{16} \cdot s_{16}$$

In this formula, the modulo-2 additions and multiplications can be replaced with XOR and AND operations. We can write it as:

$$S_{n+m+1} = (g_1 \& s_1) \oplus (g_2 \& s_2) \oplus (g_3 \& s_3) \oplus (g_4 \& s_4) \oplus (g_5 \& s_5) \oplus$$

$$(g_6 \& s_6) \oplus (g_7 \& s_7) \oplus (g_8 \& s_8) \oplus (g_9 \& s_9) \oplus (g_{10} \& s_{10}) \oplus$$

$$(g_{11} \& s_{11}) \oplus (g_{12} \& s_{12}) \oplus (g_{13} \& s_{13}) \oplus (g_{14} \& s_{14}) \oplus (g_{15} \& s_{15}) \oplus (g_{16} \& s_{16})$$

$$= \oplus(G(N) \& S(N))$$

Since the actual hardware design operates in symbol units rather than bit units, the 16-bit future LFSR state values may need to be computed simultaneously. In this case, the 16-bit states can be calculated in parallel by performing an XOR operation with 16 states, each shifted by 1 from the previously predicted polynomial vector G, as shown below:

$$S_{n+m+k} = \oplus(G \& S_{n+k}) \quad \text{for} \quad 0 \le k \le 16$$

This approach requires the logic to store the past 16 specific states in memory.

Error Control Model

A model is needed to manage the process of storing past LFSR states and calculating future LFSR states using a prediction model particularly when synchronization failures occurs, leading to a significant error. The model must store the previous seed in the memory only if no data errors are detected. If an error is present, storing the seed could cause another synchronization failure. When the LFSR fails to generate the correct

random bit sequence, and the related symbol error is detected, the error control model should replace the LFSR state with the predicted state value once the internal counter reaches M. Once the predicted state is updated, the LFSR can resume generating the correct random bit sequence using the initial polynomial. This paper does not cover error checking, however, we assume that the system employs its own error checking scheme, such as cyclic redundancy check (CRC) or a special coding scheme like 8B10B, which provides error detection capabilities. For example, consider the use of an 8B10B decoder in conjunction with the proposed model. If a running disparity or code error occurs due to a synchronization failure, the corresponding error indication signal is transmitted to the error control logic. The control module will then replace the computed next LFSR state with the predicted LFSR state, which has already been computed and stored in memory for the next descrambling operation. While the prediction model can recover from synchronization loss within a single cycle, error detection logic typically takes a few cycles to report an error in a real system. Therefore, recovery action by the prediction model may occur several cycles after the error detection logic signals the presence of an error.

Figure 13 illustrates a flowchart for the error control model. The model first checks if the $(N+1)th$ received symbol contains an error. If no error is detected, the 16-bit LFSR state value used for the Nth symbol is stored in the internal memory. Upon detecting an error for $(N+1)th$ symbol, the model computes the predicted state and prepares to apply it just before the target symbol is processed. The model then replaces the original LFSR state for $(N+M+1)th$ symbol with the prediction state.

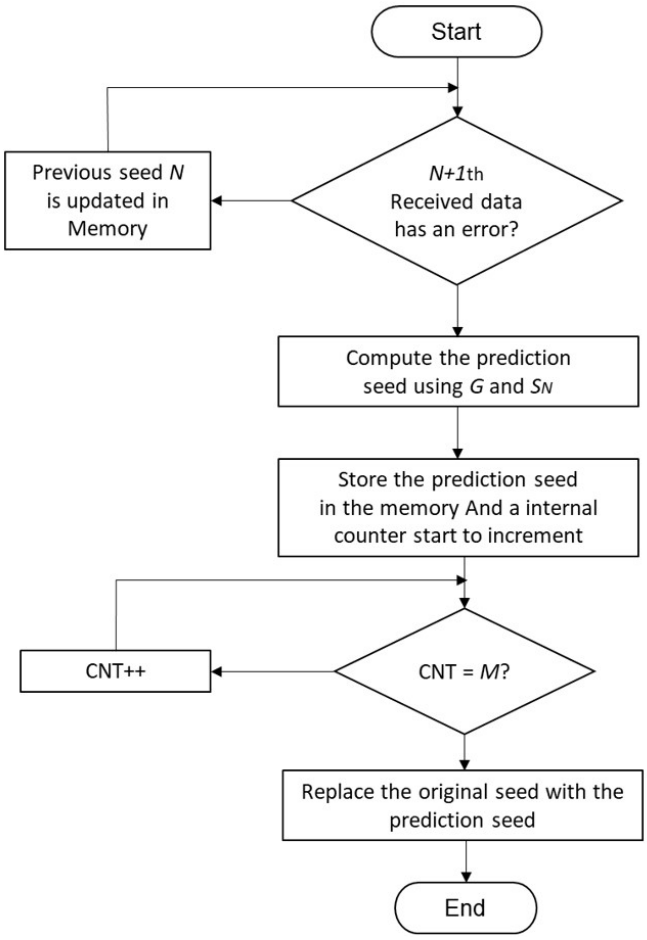

Fig. 13 A flowchart for the error control model

Descrambler Model

The logical architecture of the descrambler mirrors that of the scrambler. The same LFSR output is XOR'd with received scrambled data. In the descrambler, the LFSR is controlled by the error control model to handle cases where it fails to generate the correct random sequence. To enable this functionality, a multiplexer is added, allowing the error control model to replace the LFSR state in the event of a synchronization failure. Figure 14 illustrates the complete scrambler and descrambler model, including the proposed prediction models and error control logic. the system includes two memories: one for storing previous and one for the future state, these memories are relatively small size, as only 16 previous states and one future state need to be stored.

Implementation Cost Analysis

Table 5 shows the implementation cost for the proposed design. The required logic cost was analyzed if the degree of the polynomial is 16.

To compute the polynomial, 16 shift registers and 16 XOR gates are required. For LFSR state prediction, 16 XOR gates

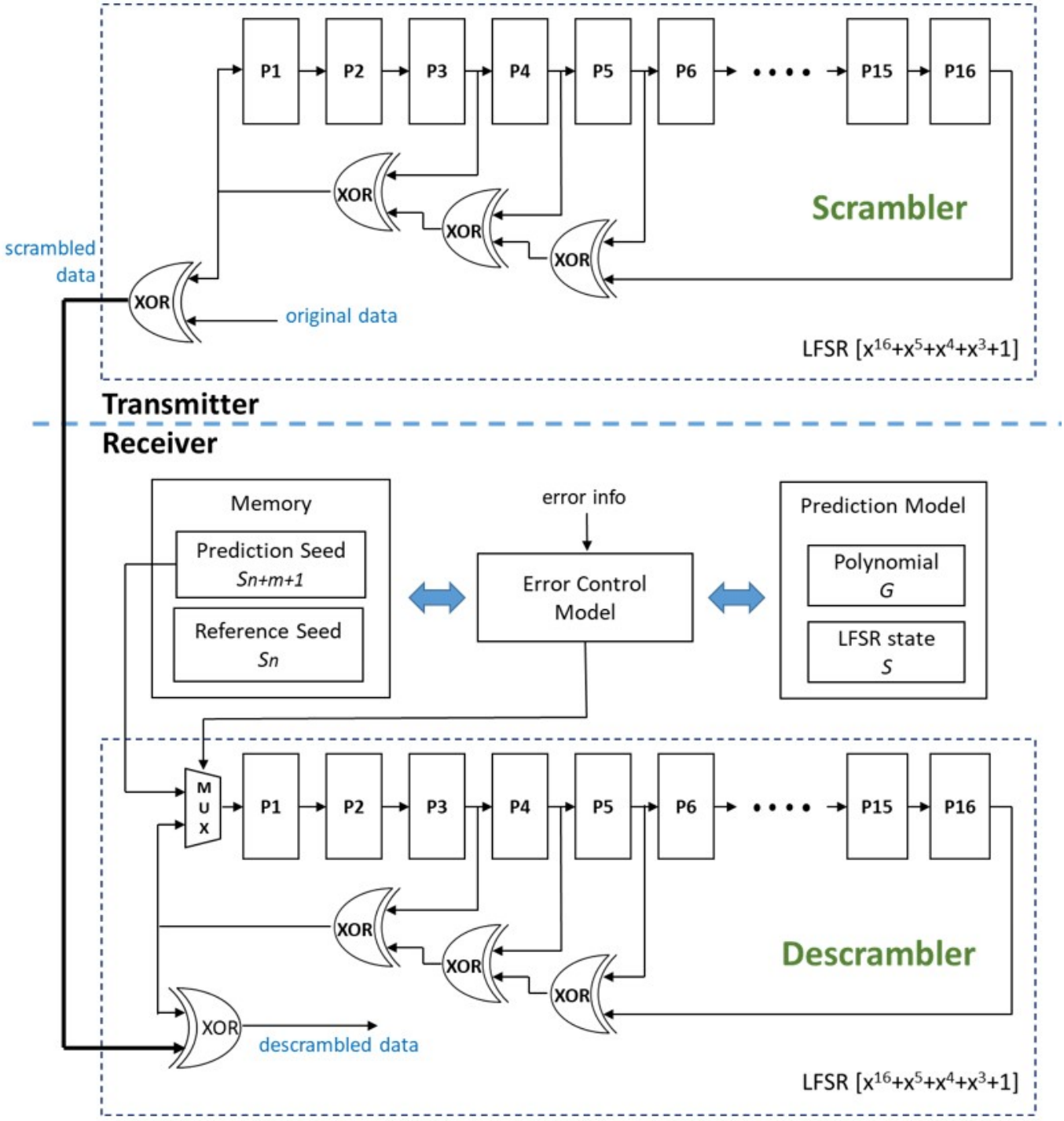

Fig. 14 An entire scrambling block diagram with the proposed prediction models

Table 5: Implementation cost for the proposed model (Polynomial Degree = 16)

Polynomial	Degree(N)	Irreducible Polynomial	Maximum Run Length (M = 2^N-1)	Frequency Test (P-value)	Entropy Test
$x^{16}+x^5+x^4+x^3+1$	16	Yes	65535	0.7584	0.9999985
$x^{16}+x^{15}+1$	16	No	255	0.0000	0.8972440
$x^{16}+x^{14}+x^{13}+x^{11}+1$	16	Yes	65535	0.2635	0.9999804
$x^{16}+x^{14}+x^{12}+x^7+x^6+x^4+x^2+x^1+1$	16	Yes	65535	0.8521	0.9999994
$x^{16}+x^{14}+x^{12}+x^8+1$	16	No	7665	0.0000	0.9182958

and 16 AND gates are needed to compute one bit of the LFSR state. Memory is also required to store the predicted polynomial, the LFSR state, and the previous LFSR states needed for the calculation, with 16 flip-flops required for each. If parallel processing is needed to compute multiple states simultaneously, the logic required increases proportionally to the number of parallel states. For example, if 16 state predictions need to be computed simultaneously, a total of 16×16 XOR gates and 16×16 AND gates are required. The memory size needed to store the computed prediction model and LFSR state remains the same, requiring 16 flip-flops. However, to compute 16 LFSR states simultaneously, 256 flip-flops are required to store the previous 16 16-bit seeds. Additionally, as the degree of the polynomial increases, the cost of all blocks increases proportionally to the increased degree.

Simulation

All models were designed in Python for simulation. A 4K resolution image (3840x2160) was used to test the design, containing a total of 8,294,400 pixels. Each pixel is a 24-bit symbol, consisting of three 8-bit RGB values. We used an open-source online tool to convert the image into HEX arrays, and then the file was converted into an 8,294,400-bit binary stream using Python[19]. The code saved the file at each step to check the results and to prepare for the next process. Figure 15 dipicts a simplified psedo-code for the implemented scrambler model. It uses 16 shift registers and includes a feedback path after performaing XOR operation. The initial seed applied is all ones. The LFSR output is XOR'd with the original input stream to scramble the image symbols. Figure 16 shows the simulation results for the scrambled image which is fully randomized.

In the simulation, a test was conducted to recover from synchronization failure by predicting the first LFSR state of the next horizontal line. Since the test image has 2160 pixels per line and each pixel is represented by a 24-bit symbol, the model needs to generate the new LFSR state for the first bit of the image line in the simulation. Given that the model requires 16-bit shifts to compute the new seed bit value, the polynomial G must be computed until M reaches 16 bits before the first bit of a new line. Therefore M should be $92,144(3840 \times 24 - 16)$.

In this Python model, the polynomial is represented as a binary array vector. Since $N = 0$, the initial polynomial is $x^{16} + x^5 + x^4 + x^3 + 1$, and the model continues to compute the new polynomial until $M = 92,144$. If $P[0] = 1$, the initial polynomial is XOR'd with the shifted polynomial. As a result, the polynomial G computed by the model becomes $x^{15} + x^{12} + x^9 + x^6 + x^5 + x^4 + 1$.

Once this polynomial is computed, the model can calculate a future state that is M bits away from that past state applied. Therefore, the polynomial does not need to be recalculated unless a different M value is set. Figure 17 illustrates a simplified pseudo-code for the polynomial prediction model.

To recover the scrambled image, the descrambler model, which includes the LFSR state prediction model that computes the future state value of the pixel location to be restored, was implemented. After reading the scrambled image and converting it into an array vector, the data is recovered through an XOR operation with the random bit LFSR output in the descrambler. The LFSR operates using the same algorithm as the transmitter's scrambler, ensuring that the recovered image matches the original image. Figure 18 illustrates the simplified pseudo-code for the LFSR state prediction model, which is capable of computing the future LFSR state vector when an error is detected.

In this simulation, we assumed that the descrambler fails to generate the correct random sequence in the middle of the image due to ESD, An LFSR state error was intentionally injected into the LFSR to simulate this scenario. Once the error was injected, synchronization between the scrambler and descrambler was disrupted, leading to a burst of massive errors at the receiver, as shown in Figure 19. The failure persisted until the sync symbol was received from the scrambler. We then confirmed that the proposed state estimator with error control logic successfully restored the LFSR by generating a new 16-bit state value using the pre-calculated G and the 16-bit state that was 3840 pixels prior. Figure 19 also shows the recovered image after applying the prediction models.

Mean Square Error / Root Mean Square Error Analysis

Mean Squared Error (MSE) is a commonly used metric for evaluating image quality[20,21]. It measures the average squared difference between the original and descrambled pixel values, quantifying how much the two images deviate from each other. Smaller MSE values indicate a better match between the images. In this study, we compute the sum of the squared differences for each R, G, and B pixel symbol separately. These three sums are then averaged to calculate the overall MSE. The MSE is obtained by dividing the sum by the total number of pixels in the test image. The MSE between two images can be expressed as:

$$MSE = \frac{1}{HW} \sum_{x=0}^{H} \sum_{y=0}^{W} \left(Y_{xy} - \hat{Y}_{xy}\right)^2$$

Where H and W are the number of rows and columns in the

Pseudo-code 1 : Scrambler

R : current LFSR state array vector expressed as R [0:15]
Rp : previous LFSR state array vector expressed as Rp [0:15]
XOR : Exclusive OR operation

Input: Initial seed *seed* [0:15] Origianl bit stream *org_in*

1. R [0:15] ← *seed* [0:15]
2. $i \leftarrow 0$, integer variable
3. $j \leftarrow 0$, integer variable
4. $N \leftarrow$ Total number of bits to be scrambled in the simulation
5. **while** $i < N$ **then**
6. **for** $j = 0$ to 15 **do**
7. Rp [j] = R [j]
8. **end for**
9. R [0] = R [15] *XOR* R [4] *XOR* R [3] *XOR* R [2]
10. sc_out = R [0] *XOR* *org_in* [i]
11. **for** $j = 0$ to 15 **do**
12. R [j +1] = Rp [j]
13. **end for**
14. $i \leftarrow i + 1$
15. **return** *sc_out*
16. **end while**

Output: Scrambled bit *sc_out*

Fig. 15 Simplified pseudo-code for the scrambler function

Fig. 16 Scrambled test image

test image respectively. Y_{xy} and $\hat{Y}_{xy}$ are the original and the descrambled pixel values at pixel location $[x, y]$.

Root Mean Square Error (RMSE) is another metric that measures the standard deviation of the variance between the two images[21,22]. It is simply the square root of the MSE, which reflects the difference between the descrambled pixel values and the original pixel values. RMSE normalizes the difference by using the maximum pixel value for scaling:

$$RMSE = \sqrt{MSE} = \sqrt{\frac{1}{HW}\sum_{x=0}^{H}\sum_{y=0}^{W}\left(Y_{xy} - \hat{Y}_{xy}\right)^2}$$

Table 6 presents the MSE and RMSE results for the proposed technique. The table also includes the scrambling synchronization failure rate (SFR), which indicates how often synchronization failures occur.

Pseudo-code 2 : Polynomial Prediction

P: Polynomial array vector expressed as P[0:15]
$ROLL$: Left-shifts the number by 1-bit
XOR: Exclusive OR operation

Input: Initial polynomial vector G[0:15] Target bit steps M

```
P[0:15] ← G[0:15]
i ← 0, integer variable
j ← 0, integer variable
while i < M then
    if P[0] = 0 then
        ROLL(P[0:15])
    else
        ROLL(P[0:15])
        P[15] = 0
        for j = 0 to 15 do
            P[j] XOR G[j]
    end if
    i ← i + 1
end while
return P[0:15]
```

Output: Predicted polynomial P

Fig. 17 Simplified pseudo-code for the polynomial prediction function

For example, SFR = 1 represents a scenario where synchronization failure occurs every frame, while SFR = 1/600 indicates a failure rate of one failure every 600 frames. The results show that the MSE and RMSE of the proposed technique remain consistently low, even in cases of synchronization failure, outperforming traditional schemes that recover per line or per frame.

Peak Signal-to-Noise Ratio Analysis

Peak Signal-to-Noise Ratio (PSNR) is a widely used metric that quantifies the quality of a descrambled image by comparing the maximum signal-to-noise ratio between the original and descrambled images, expressed in decibels (dB)[18,20]. Higher PSNR values indicate better quality, as they suggest that the descrambled image is closer to the original one. The PSNR is calculated using the following equation:

$$PSNR(\text{dB}) = 20 \times \log_{10}\left(\frac{P_{\max}}{\sqrt{MSE}}\right)$$

Where $P_{\max}$ is the maximum pixel value in the image, which is 255 for RGB pixels, and MSE is the Mean Square Error, representing the average squared difference between the original and descrambled pixel values. Since each RGB pixel has a maximum value of 255, we use $P_{\max} = 255$ for the PSNR calculation.

Table 7 presents the measured PSNR results. The PSNR of the proposed technique is significantly higher in all cases of synchronization failure compared to traditional schemes that recover per line or per frame. This shows that the proposed model is more effective in maintaining image quality even in the presence of synchronization failure.

Bit Error Rate and Recovery Time Analysis

The Bit Error Rate (BER) measures the ratio of pixel bits that are descrambled with errors to the total number of pixel bits transmitted[?]. A lower BER indicates better accuracy in recovering the image without errors, while a higher BER reflects a greater level of distortion or corruption in the descrambled data. BER is calculated using the following formula:

The Recovery Time refers to the number of pixel cycles required to recover from a synchronization failure. A short recovery time indicates that the system can quickly restore synchronization and minimize the impact of errors, which is important

Pseudo-code 3: LFSR State Prediction

AND : AND operation
XOR : Exclusive OR operation
R : current LFSR state array vector expressed as R [0:15]
REVERSE : Reverse the vector array from [0:15] to [15:0]

Input: Predicted polynomial vector P [0:15],
16 stored LFSR state vectors S [0:15] [0:15]. This is a 2-D 16x16 array vector

```
for i = 0 to 15 do
    for j = 0 to 15 do
        if j = 0 then
            state_out[i] = P[j] AND S[i][j]
        else
            state_out[i] = state_out[i] XOR (P[j] AND S[i][j] )
        end if
    end for
end for
state_out = REVERSE (state_out[0:15])
while i < N then
    if i = M+16 then
        for j = 0 to 15 do
            R[j] = state_out [j]
        end for
    else
        R[j] = R[j]
    end if
    return R[j]
end while
```

Output: Predicted LFSR state vector R [0:15]

Fig. 18 Recovered image using the prediction models in massive burst error situations

Table 6: MSE / RMSE measurement results

Polynomial	Degree(N)	Irreducible Polynomial	Maximum Run Length ($M = 2^N-1$)	Frequency Test (P-value)	Entropy Test
$x^{16}+x^5+x^4+x^3+1$	16	Yes	65535	0.7584	0.9999985
$x^{16}+x^{15}+1$	16	No	255	0.0000	0.8972440
$x^{16}+x^{14}+x^{13}+x^{11}+1$	16	Yes	65535	0.2635	0.9999804
$x^{16}+x^{14}+x^{12}+x^7+x^6+x^4+x^2+x^1+1$	16	Yes	65535	0.8521	0.9999994
$x^{16}+x^{14}+x^{12}+x^8+1$	16	No	7665	0.0000	0.9182958

Table 7: PSNR measurement results

Descrambling Scheme	PSNR (dB)		
	SFR = 1	SFR = 1/60	SFR = 1/600
Ours	62.579	80.375	90.380
Descrambler initialized per line	37.546	55.315	65.288
Descrambler initialized per frame	10.534	28.260	38.278

for maintaining image quality and consistency in real-time systems. BER is calculated by dividing the number of bits received incorrectly by the total number of bits transmitted over a given period of time.

$$BER = \frac{\text{number of error bits}}{\text{total number of received bits}}$$

Table 8: BER / Recovery time analysis results

Descrambling Scheme	BER			Recovery Time (pixel cycles)		
	SFR = 1	SFR = 1/60	SFR = 1/600	SFR = 1	SFR = 1/60	SFR = 1/600
Ours	1.66e-06	3.43e-08	3.25e-09	2	2	2
Descrambler initialized per line	4.89e-03	9.76e-06	9.77e-07	394	397	389
Descrambler initialized per frame	2.57e-01	5.12e-03	5.08e-04	221317	219682	210523

In Table 8, the proposed technique demonstrates a low BER and fast recovery time, highlighting the efficiency of the recovery process.

Discussion

This study aimed to develop an effective polynomial and LFSR state prediction model to address synchronization problem in LFSR-based descrambling system, particularly in the presence of electrostatic discharge. The proposed scrambling technique was compared to traditional LFSR-based scrambling and self-synchronous scrambling methods, as shown in Table 9.

Table 9: Comparison between traditional and proposed scrambling techniques

	LFSR-based Scrambling Technique	Self-synchronous Scrambling Technique	Proposed Scrambling Technique
Sync Symbol	Additional special symbol is required	Does not require sync symbol	Does not require sync symbol
Synchronization	Periodic synchronization is required	Initial seeding is required, but not require separated synchronization	Initial seeding is required, but not require separated synchronization
Recovery Time	Cannot recover until the re-synchronization	16 bit period	16 bit period + error detection time (1 or 2 cycles)
Error multiplication	No	Single bit error can produce 16 bit errors	No
Hardware Cost	Scrambler : 16 flops with 4 XORs Descrambler : 16 flops with 4 XORs	Scrambler : 16 flops with 4 XORs Descrambler : 16 flops with 4 XORs	Scrambler : 16 flops, 4 XORs Descrambler : 16 flops, 4 XORs Prediction : 16 flops, 16 XORs, 16 ANDs Memory : 32 flops

Unlike the LFSR-based method, both the self-synchronous and proposed methods do not require a separate sync symbol or LFSR state synchronization. Furthermore, the recovery time is significantly shorter for both the self-synchronous and proposed methods. The BER results in Section 3.3 demonstrate that the LFSR-based scrambling technique induces significant burst errors when synchronization fails, persisting until the next synchronization symbol is received. This can cause screen flickering in display systems and generate garbage data in network devices, potentially degrading performance until synchronization is re-established. However, if the system is designed to recover data from any point and can tolerate an increase in BER due to bit error multiplication, a self-synchronous scrambling technique, enabling data reception without relying on a predefined synchronization point, could be employed. The measurement results, along with comparisons to existing scrambling techniques, emphasize the proposed recovery method's ability to address the limitations of LFSR-based scrambling. This makes it particularly suitable for display systems or network devices that require robustness and minimal error propagation.

While the proposed method incurs additional logic due to the inclusion of a prediction model and memory logic, it successfully avoids the issue of error multiplication present in the self-synchronous method. In the self-synchronous approach, a single-bit error can propagate into 16 multi-bit errors, which would make it unsuitable for applications like video transmission where retransmission may not be feasible. In contrast, the proposed technique allows for rapid recovery from synchronization failures without introducing error multiplication. This makes it particularly well-suited for systems using LFSR-based scrambling methods, where fast and reliable recovery is essential. The polynomial prediction model's computation time is similar to that of the LFSR in the descrambler. Specifically, for LFSR state prediction, 16 XOR and 16 AND operations need to be completed within a single clock cycle. By employing a 5-stack structure (1 stack for AND operations and 4 stacks for XOR operations), we can reduce propagation delays. Although we did not conduct gate level synthesis in this study, even given the high clock frequencies in modern video standards (in the hundreds of MHz), these operations are expected to be manageable within a single clock cycle. In the simulation, the proposed model successfully restored at the desired pixel location, preventing errors across the entire frame until the descrambler received a synchronization reset. However, as shown in Table 8, the errors were minimized, though they could not be entirely eliminated. Errors persisted in the descrambler until the predicted state value was applied to the LFSR. Further research on this topic is likely required in the future.

The design was modeled in Python, but due to some environmental constraints, it was not possible to implement it on a hardware platform like FPGA using hardware description languages such as Verilog or VHDL. If given the opportunity, I would like to implement the algorithm on an FPGA, connect it to a display panel to measure real-world performance, compare power consumption, and evaluate logic size. I believe this would contribute to enhancing the recovery algorithm and make it more practical for real-world applications.

Conclusion

This paper proposes a polynomial and seed prediction model to recover synchronization failures in the LFSR of the descrambler. The main advantage of the proposed polynomial prediction is that it only needs to be computed once, after which it can efficiently pre-compute the LFSR state for descrambling specific scrambled data with the previously stored state. The prediction process is quick, and there is no need to shift the LFSR. Although the proposed technique requires additional logic, the efficiency of the algorithm, proven through simulation, can compensate for the drawbacks of systems using LFSR-based scrambling which requires additional sync symbol and periodic synchronization.

Acknowledgment

I would like to express gratitude to my advisor, Byung-wook Cho, at Samsung Electronics for the valuable insight provided to me on this topic.

References

1 K. Mishra, *Advance Chip Design*.

2 M. Naim, H. Pacha, A. Pacha and N. Said, *Lengthening the Period of a Linear Feedback Shift Register*.

3 M. A. Mioc and M. Stratulat, *Study of Software implementation for Linear Feedback Shift Register based on 8th degree irreducible polynomials*.

4 M. A. Mioc, *Simulation study of the functioning of LFSR for grade 4 irreducible polynomials*.

5 S. Xp Liu, X. Wu and C. Chui, *Primitive polynomials for robust linear feedback shift registers-based scramblers and stream ciphers*.

6 B. G. Lee and S. C. Kim, *Scrambling Techniques for Digital Transmission*.

7 C. Jeff Reid and R. S. ECC, https://rcgldr.net/misc/ecc.pdf.

8 S. Shpigelblat, *Implementing the Viterbi Algorithm in Today's Digital Communications Systems*, https://www.design-reuse.com/articles/21107/viterbi-algorithm.html.

9 E. Geiger, *A Look at Some Scrambling Techniques Used in Various Data Transport Protocols*, https://www.ieee802.org/11/Documents/DocumentArchives/1993_docs/1193216_scan.pdf.

10 S. W. Golomb, *Shift register sequences*.

11 E. Billauer, *Conversion between Galois and Fibonacci polynomials of Linear-Feedback Shift Register*, https://www.01signal.com/other/lfsr-galois-fibonacci/.

12 M. Siswanto, G. Witjaksono, M. Soeheila and Z. Hamdan, *Study on the Effects of Characteristic Polynomial in LFSR for Randomness Quality*.

13 J. AndrewRukhin, M. S. JamesNechvatal, S. L. ElaineBarker, M. V. MarkLevenson, A. DavidBanks and S. JamesDray, *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*.

14 Unitychain, *Provable Randomness: How to Test RNGs*, https://medium.com/unitychain/provable-randomness-how-to-test-rngs-55ac6726c5a3.

15 *Moneesh Nagireddy. Spectrum Analysis in Python*, https://www.geeksforgeeks.org/spectrum-analysis-in-python/.

16 M. Kramer, *The Power Spectrum - Part 1*, https://mark-kramer.github.io/Case-Studies-Python/03.html.

17 Q. Kong, T. Siauw and A. Bayen.

18 J. Dong, G. Wu, T. Yang and Y. Li, *The Improved Image Scrambling Algorithm for the Wireless Image Transmission Systems of UAVs*.

19 https://javl.github.io/image2cpp/, GitHub, Image2cpp,.

20 Z. Chen, Y. Yang and X. Jiang, *An Image-Encryption Algorithm Based on Stage-Merging Bit Scrambling*.

21 U. M. Sabbha, *MSE*, https://dev.to/mondal_sabbha/understanding-mae-mse-and-rmse-key-metrics-in-machine-learning-4la2.

22 https://coralogix.com/ai-blog/root-mean-square-error-rmse-the-cornerstone-for-evaluating-regression-models/, Or Jacobi, Root Mean Square Error (RMSE),.