International Journal of Engineering Trends and Technology
ISSN: 2231–5381 / https://doi.org/10.14445/22315381/IJETT-V73I2P119

Volume 73 Issue 2, 224-229, February 2025

Original Article

Key Generation Algorithm based on Array Element Substitution

Juraev G.U[1], Bozorov Asqar[2], Sindorov Davlatbek[3], Salimov Sirojiddin[4]

[1]*National Universitet of Uzbekistan, Tashkent, Uzbekistan.*
[2,3]*Tashkent State University of Economics, Tashkent, Uzbekistan.*
[4]*Tashkent University of Information Technologies named after Muhammad al-Khwarizmi, Tashkent, Uzbekistan.*

[2]*Corresponding Author : asqarbozorov1990@gmail.com*

Received: 13 November 2024 Revised: 16 January 2025 Accepted: 04 February 2025 Published: 21 February 2025

Abstract *- In this article, using a 128-bit key, 2^{64} A stream coding algorithm has been developed that is easy to implement in software, allowing random bit sequences to be generated up to the bit length, i.e. the key stream. This algorithm is similar to and simpler than the RC4 algorithm, which retains its security, speed, and flexibility features without using an initialization vector. The pseudorandom sequences generated by this algorithm meet National Institute of Standards and Technology (NIST) requirements for randomness.*

Keywords *- Stream cipher, RC4, Spritz, Array, NIST, Randomness, KSA, PRNG, Key generation.*

1. Introduction

A pseudorandom number generator based on the permutation of array elements is widely used in constructing continuous encryption algorithms. For example, popular continuous coding methods RC4 and Spritz are based on permutation of array elements [1]. RC4 algorithm is a basic variable-length key stream generator specifically developed by R. Rivest [2]. Pseudorandom number generators using algorithms such as RC 4 generally run significantly faster than block code-based generators.

The RC4 algorithm is widely used in various information security systems and computer networks (for example, in the protocol). SSL for password encryption Windows NT, etc.). Spritz is a lightweight stream cipher developed by Bruce Schneier and Daniel Whiting. It is known for its simplicity, speed, and security. Spritz is particularly suitable for resource-constrained devices such as microcontrollers and smart cards. Spritz is essentially an improved version of the RC4 algorithm, considering modern cryptographic tools and algorithms. It also uses a 256-element byte array. Spritz uses an archaic alphabet and the concept of a spinning wheel to generate pseudorandom sequences that are used to encrypt data. The algorithm has a small internal state, which allows it to be implemented efficiently on devices with limited memory.

2. Literature Review

RC4 is a byte stream cipher that uses a permutation mode to generate permutations using a table of numbers from 0 to 255 and two-byte index pointers [4]. RC4 keys typically range in length from 40 to 128 bits, and modern stream encryption does not require a separate key. The table is initialized with a key combination. Then, in the keystream generation stage, the table is modified and outputs one key byte at each iteration. Its speed and simplicity allow for efficient software implementation and easy hardware development. In 2001, Fluhrer, Mantin, and Shamir published a paper on a vulnerability in the RC4 key table [2]. They showed that the first few bytes of the keystream are not random among all possible keys. From these bytes, information about the encryption key used can be inferred with high probability.

If the long-term and short-term keys are simply concatenated to create an RC4 encryption key, then this long-term key can be obtained by analyzing a large number of messages encrypted with that key. This vulnerability and some of its associated effects were used to break WEP encryption in IEEE 802.11 wireless networks. This marked the need for a quick replacement of WEP, which led to the development of a new wireless security standard, WPA, and many of its flaws were mysteriously fixed. The RC4 algorithm worked in a wide range of applications until it was deprecated for all versions of TLS in 2015. Although the initialization phase of RC4 and the statistical properties of its first few bytes have been seriously questioned, the key generation phase is still considered secure, especially due to the provision of enough keystream bytes for 128-bit security. In this work, based on the Sponge design, a stream data encryption algorithm has been developed that is sufficiently resistant to the most well-known cryptanalysis attacks. Due to the shortcomings mentioned above, the developer updated the algorithm called Spritz [5], which still uses a similar structure to RC4, but the state update function is more complex in order to achieve better randomness.

3. Materials and Methods

Most stream encoding algorithms are based on a Linear Feedback Shift Register (LFSR). This allows high-speed encryption to work in the IP format. However, LFSRs make it difficult to implement codes in software. Since RC4 is based on byte operations instead of the LFSRs in Spritz encoding algorithms, it is easy to implement in software. In RC4, a simple program executes 8 to 16 machine instructions for each byte of text, so encryption software must be very fast. The basic RC4 algorithm is shown below. The next step of the RC4 algorithm is called the Pseudorandom Generation Algorithm (PRGA). This step generates pseudorandom values, which are then XORed with the plaintext for the encryption process or the ciphertext for the decryption process). The first step is to initialize the values i and j to zero. For k = 0, k = message length –1, the new values i and j are calculated as follows: i = (i + 1) mod 256 j = (j + S[i]) mod 256The value of S[i] and S[j] are swapped. Then t = (S[i] + S[j]) is the value of S with index mod256. The value s[t] is finally XORed with the plaintext or ciphertext with index k. Here is an arrowhead view of the kernel, where each step defines the step-by-step process of the algorithm:

1. i=i+1
2. j=j+S[i]
3. SWAP (S [i],S [j])
4. z=S[S[i]+S[j]]
5. return z

RC4 has several well-documented vulnerabilities. The initial bytes of its keys are inconsistent, and it is vulnerable to the Fleurer-Mantin-Shamir (FMS) attack making it vulnerable to attack. Over time, many RC4-based protocols (such as WEP and TLS) were abandoned due to these vulnerabilities [1].

3.1. Application of the Spritz Algorithm

The Spritz algorithm, developed by Schneier and Whiting, is improved in RC4 by using a more complex conditional array processing process to overcome its shortcomings. Spritz is lightweight, secure, and optimized for use in constrained environments such as smart cards and IoT devices. It generates a pseudorandom sequence by continuously permuting a 256-element byte array, similar to RC4 but with random bases. The famous Spritz RC4 weakness removal does achieve For Job development He made it more complicated task keys planning and keys create processes that found in RC4 were error thoughts No does in the meantime What she is on the attack relatively more stable Spritz Same So does Available in RC4 version is was key from flow elementary from incline runs away. A graphical representation of the Spritz stream encoding algorithm.

Detailed Spritz Stream Cipher Algorithm (Flow Diagram)

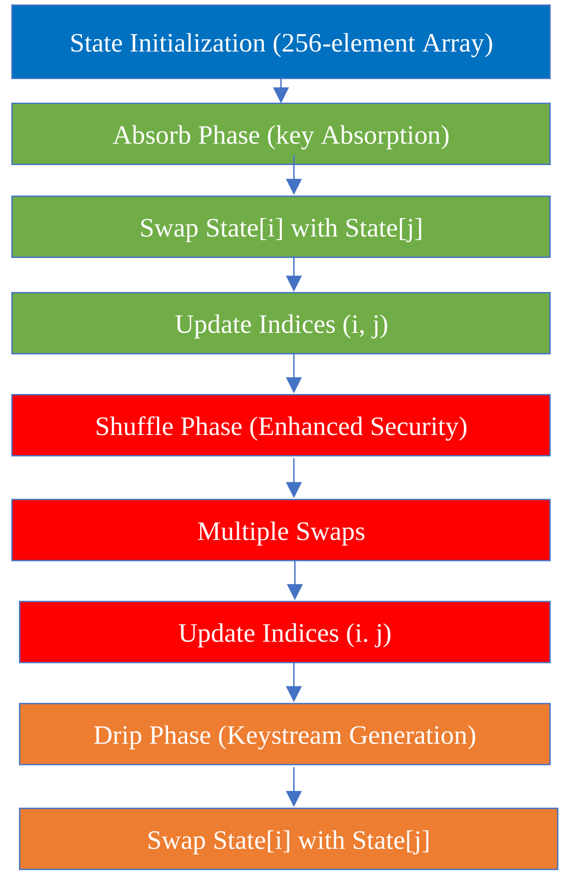

Fig. 1 Block diagram of the SPRITZ algorithm

This diagram shows the main steps: state initialization, key derivation, a mixing step, and key stream generation yield do it This). Below is a view of the kernel with an arrow, where each step defines the step-by-step process of the algorithm:

1. i=i+1
2. j=k+S[j+S[i]]
3. k=i+k+S[j]
4. SWAP(S[i],S[j])
5. z=S[j+S[i+S[z+k]]]
6. return z

However, in FSE 2016, Banik and Isobe found that the randomness of the first two bytes of Spritz was still insufficient to resist attacks [5]. For this reason, the pseudorandom number generator was replaced by array elements.

4. Results and Discussion

The proposed algorithm follows the structure of traditional stream coding but with nonlinear transformations. It improves the basic planning process by replacing array elements. In this paper, we focus on the description of the algorithm oriented to the creation stage. The key generation mechanism in the proposed Measuring Array Element Generator (MEAG) algorithm is based on array element permutation. This process involves repeated permutation of the state array based on the key and two auxiliary variables, as well as the variables i, j, r, rr, which complicates the main planning process. The key generation process in this algorithm is similar to RC4 but with the added complexity of auxiliary variables based on the constant change of array elements. Key generation is performed in two stages: the basic scheduling algorithm (KSA) and the Pseudorandom Generation Algorithm (PRGA). This coding algorithm is based mainly on stream coding algorithms and is used to generate a stream of random bits by continuously permuting the elements of an array. The sequence of steps of the algorithm is presented below.

4.1. Initialization of S-Array

The first part of the algorithm is based on the initialization of the array (array S). The size of this array is initially 256: $S[\ ] = i$ for $0 \leq i < 256$, i.e. initially, each index has its own value. The array of keys is initialized using the key bytes $K\,[]$. Key Scheduling Algorithm (KSA)The key scheduling algorithm "rotates" the array. Here, the position of the array elements is changed using a switch. At each stage:

r =S[(r+S[i]+K[imodc]+S[rr])mod256];
rr=S[(r+i+j) mod256]⊕rr; S[i], S[i]=S[r], S[r] therein:
r - an auxiliary variable that updates the index;
K[i]=key[i mod key length] - each key element;
rr - a parameter that further improves the randomization process.

At every step $Sr=Si$, values to be replaced:

This process is repeated 256 times, completely shuffling the array. At each step of the replacement, the elements of the array are randomly changed, creating a random stream of bytes.

4.2. Generating A Pseudo-random Byte Stream (PRGA)

During encryption, a stream of random bytes is generated from the array. At this stage, a sequence of encoded bytes is generated for each message. Mathematical representation of the algorithm steps: A PRGA algorithm step consists of generating a sequence of bytes during the encryption process. The basic mathematical process here is as: B=S[(b+S[(i+j)mod255])mod255]

Where: i- is the index that updates the position in the array;

j - random byte control index;
b is the byte value calculated in the previous steps.
After this, a new value is generated at each step:
v[i1]=S[(S[S[b]]+1)mod 256].

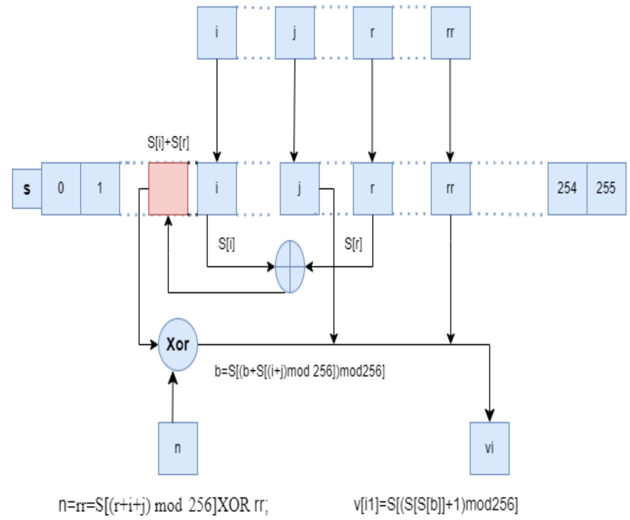

Fig. 2 Generator replacing array keys

S[S[b]] represents the internal value of the array, i.e. this value depends on another value inside it. Internal indices are used to increase randomness. When creating an encrypted message, each byte is randomly replaced with a key stream using the XOR operation: j=v[i1] ⊕ j; after this, the array values are replaced as follows:

S[i], S[b] = S[b], S[i]

During this process, each byte is randomized, and the array is updated at each step. Thus, the array elements are swapped at each iteration, resulting in a new random value being generated at each step. In this study, we will use the sample values shown in the binary_viewer file as part of the sample. Before encryption, the plain cipher value is: Then the algorithm key calculation is performed for the value j. The key used to calculate j must be converted to ASCII code. Plain ASCII for "market": 'B=66', 'o=79', 'z=90', '0=79', 'r=82', 'o=79', 'v=86'. ASCII value for the key "soldier": 'A'=65', 's=83', 'd=81', 'a=65', 'r=82'. Key steps of the Key Scheduling Algorithm (KSA). The main scheduling step begins with initializing the initial state as an array of 256 elements. The initial state array looks like this:

First, we initialize array S with 256 elements. Each index has its own value, i.e., S[i]=i for 0≤i<256.

Then the first value of σ with initial value r = 0 and rr = 0, j=0 is calculated as follows:

r=S[(r+S[i]+K[key_length]+S[rr])mod256]
r=(0+0+66)mod256=66
rr=(S[(r+i+j))mod 256] ⊕ rr; rr=[(66+1+1)mod 256] ⊕0=68

Replace the value of S[0] with S(68). This step is repeated until the value of i reaches 255, i.e. the process is repeated 256 times until the array S is completely shuffled. The results of the basic planning level can be seen in Table 2, where the value of i is located on the blue line.

Table 1. Array table S

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Table 2. Results of the Key Scheduling Algorithm (KSA)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
34	192	22	217	38	93	17	57	162	29	104	230	99	253	98	161
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
36	31	25	215	48	123	133	42	74	87	97	112	7	160	167	89
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
54	108	182	100	116	125	51	146	224	124	122	147	0	174	80	240
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
132	231	151	61	197	120	255	152	90	169	183	131	28	24	232	117
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
19	148	1	46	58	105	27	92	96	168	35	47	45	109	158	159
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
235	196	154	140	181	68	39	30	60	191	177	198	143	103	70	37
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
72	228	227	23	214	190	149	5	216	71	63	76	248	178	163	56
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
26	83	212	59	32	186	145	170	75	220	14	86	249	106	139	157
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
200	82	165	107	129	4	43	180	156	150	155	238	194	171	6	130
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
102	65	179	110	184	113	172	94	202	62	135	16	164	50	173	219
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
188	52	250	141	66	78	193	199	142	121	204	229	206	136	222	252
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
33	137	41	64	15	251	85	21	205	205	101	237	201	67	55	208
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
84	95	81	243	254	226	241	91	77	10	127	246	18	9	225	187
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
166	44	185	40	13	118	223	247	2	115	213	114	207	134	175	11
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
210	8	138	245	20	3	211	195	244	218	49	126	239	128	242	144
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255
221	12	209	153	233	53	119	69	203	176	88	73	236	111	189	234

This process randomly changes the elements of array S at each iteration and prepares it for encryption. Pseudorandom Generation Algorithm (PRGA) Encryption Step After the key planning step is performed, the "challenge" message encryption process is performed using each encryption process. Initialize i = 0, j = 0, r = 0, rr = 0, and w is a random number, GCD, or a relatively prime number of length S, i.e. 256.

Then, carry out the procedure as follows: The symbol "B" is initialized with the values i = 0, j = 0, k = 0, and z = 0, and each row is modulated with the value 256. Using PRGA, we generate random bytes for encryption. At each stage, the following processes are performed:

- For each iteration: b=S[(b+S[(i+j)mod 256])mod 256]
- Generate the following random value: v[i1]=S[(S[S[b]]+1)mod 256]
- XOR operation: j=v[i1] $\oplus$ j
- Swap elements in array S: S[i],S[b]=S[b]
- Let's encode using the XOR operation

For example, Encrypted byte = Clear byte $\oplus$ K.

Plain text: ASCII values for 'Market' are [66, 79, 90, 79, 82, 79, 86]. After processing at each stage above, random bytes (PRGA result) [120, 45, 63, 190, 152, 101, 202] are generated, and each text byte is XORed with random bytes generated by PRGA:

66 $\oplus$ 120 = 58, 79 $\oplus$ 45=9879, 90 $\oplus$ 63=10190, 79 $\oplus$ 190=241, 82 $\oplus$ 152=202, 79 $\oplus$ 101=42, 86 $\oplus$ 202=148. The resulting encrypted byte sequence looks like this. The text code will be (58 , 98 , 101 , 241 , 202 , 42 , 148). In this method, each text byte is XORed with random bytes generated by PRGA to produce an encrypted result. Although the proposed algorithm is similar to RC4 and Spritz algorithms, the security and efficiency have been improved to serve as an effective encryption solution.

4.3. Comparison with RC4 and Spritz

Has structural similarities with RC4 and Spritz, as all three algorithms rely on permutation array elements to generate a pseudorandom keystream. However, the proposed algorithm offers several key improvements:

- Improved Security: Unlike RC4, which is vulnerable to key recovery attacks due to inaccuracies in the keystream, the proposed algorithm overcomes these inaccuracies by introducing non-linear state updates.
- Simplified Implementation: The algorithm is simpler than Spritz, while the level of security and randomness is similar.
- No Initialization Vector (IV): The proposed algorithm does not require IV, simplifying the encryption process while maintaining security.

Based on the above comparative structures, we will analyze them based on tables and graphs. When the pseudorandom sequence generated by the algorithm is tested with random conditions of 2^{20} keys, we get the following results.

We conduct statistical tests through National Institute of Standards and Technology (NIST). Based on an Excel spreadsheet, the proposed MEAG algorithm was compared with RC4 and Spritz algorithms with similar parameters. The comparison results can be seen in Figure 4.

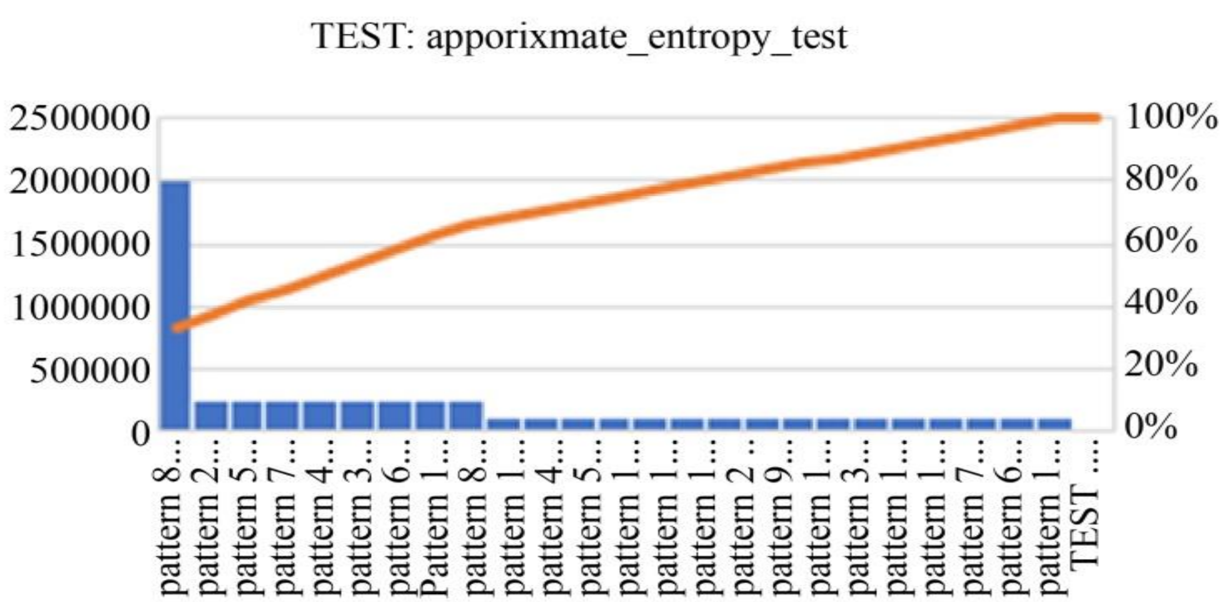

Fig. 3 Entopic test

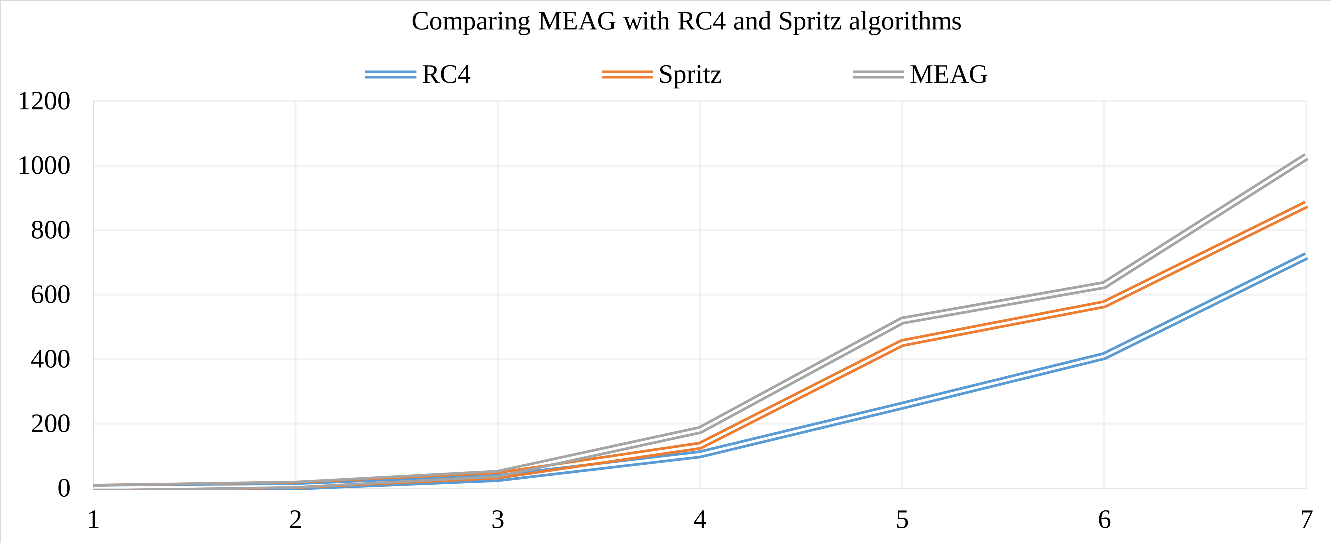

Fig. 4 Work development gave away algorithm volume building bar comparison

Table 3. Extract brought was algorithms functions analysis

Characteristics	RC4	Spritz	MEAG
Key planning	Simple, biased inclined	Complex, powerful mix	Compared to RC4 harder than Spritz easier
Key flow create	Simple, 1 index updates	More difficult, one How many indices? updates	Compared to RC4, more complex, based on XOR, is a mixture
Efficiency	High efficiency	More difficult, one effective	Almost as effective as RC4.
Safety	Against attacks Vulnerable (e.g. FMS)	Famous resistance to attacks	Improved over RC4, resistant to known attacks
From memory	Low (256 bytes) use	Low (256 bytes) set	Low (256 byte state array

5. Conclusion

The key generator algorithm presented in this paper provides a secure and efficient solution for stream ciphers based on the permutation of constant array elements. The algorithm in RC4 is improved by eliminating key scheduling errors and eliminating initial biases of the key stream. Using formal theorems and mathematical proofs, and we show that the algorithm exhibits desirable cryptographic properties such as nonlinearity, avalanche effect, and resistance to key recovery and differential cryptanalysis. Tests show that for the RC4 algorithm 2^{41} , N! For the 2^{64} algorithms, the Spritz 2^{81} algorithm requires samples to distinguish them from random ones. An algorithm for statistical attacks**,** basic reconstruction attacks, and Fleurer-Manten-Shamir (FMS) attacks has been developed. It resists some common types of cryptographic attacks, such as keystroke attacks. It offers a number of improvements over the RC4 algorithm. Thus, RC4 improves security by making key scheduling and keystream generation more difficult. Using multiple indices (r, rr, i,) and XOR-based hashing makes it difficult for an attacker j to predict the keystream. Structured The algorithm can handle big data efficiently in terms of performance and memory usage while maintaining security features verified by NIST tests.

References

[1] Deni Anggara, “Design of Encryption Application on Sound Files Using Spritz Algorithm,” *Journal of Information System Research (JOSH)*, vol. 1, no. 3, pp. 103-108, 2020. [Google Scholar] [Publisher Link]

[2] Ronald L. Rivest, and Jacob C. N. Schuldt, Spritz---A Spongy RC4-Like Stream Cipher and Hash Function, *Paper 2016/856: Cryptology ePrint Archive*, pp. 1-32, 2016. [Google Scholar] [Publisher Link]

[3] Pouyan Seperdad et al., “A Tornado Attack on RC4 with Applications to WEP and WPA,” *Paper 2015/254: Cryptology ePrint Archive*, pp. 1-65, 2015. [Google Scholar] [Publisher Link]

[4] P. Prasithsangaree, and P Krishnamurthy, “Analysis of Energy Consumption of RC4 and AES Algorithms in Wireless LANs,” *GLOBECOM '03. IEEE Global Telecommunications Conference (IEEE Cat. No.03CH37489)*, San Francisco, CA, USA, vol. 3, pp. 1445-1449, 2003. [CrossRef] [Google Scholar] [Publisher Link]

[5] Lin Jiao, Yonglin Hao, and Dengguo, “Stream Cipher Designs: A Review,” *Science China Information Sciences*, vol. 63, 2020. [CrossRef] [Google Scholar] [Publisher Link]