Sādhanā (2019)44:223
https://doi.org/10.1007/s12046-019-1209-7

An approach of refining RC4 with performance analysis on new variants

SUMAN DAS*, RANJAN GHOSH and RAJAT KUMAR PAL

Department of Computer Science and Engineering, University of Calcutta, Kolkata, India
e-mail: aami.suman@gmail.com; rghosh47@yahoo.co.in; pal.rajatk@gmail.com

MS received 31 January 2018; revised 19 December 2018; accepted 16 August 2019

Abstract. Many years of research on the RC4 stream cipher proves it to be strong enough, but there are claims that its swap function is responsible for essential biases in the output. There are suggestions to discard some initial bytes from the key-stream, to get rid of this, before the actual encryption starts, though no optimum value has been defined. In this paper, by analysing different variants of RC4, the authors have attempted to find out whether this cipher becomes more secure by discarding initial bytes and, if so, what is its optimum limit. Also, multiple *S*-boxes generated by different logics and a unique key-mixing procedure have been implemented, which made k RC4 more robust.

Keywords. Refined RC4; modified KSA; multiple *S*-boxes; key-mixing; dynamic *S*-box.

1. Introduction

The RC4 stream cipher has a simple but robust structure, which, after going through significant analysis by researchers, proves to be robust enough on different platforms. The core of this cipher is only one internal state table of size N (generally 256), which eventually acts as an *S*-box to perform the main jumbling of bytes. There are two components in RC4: the KSA, i.e. the Key-Scheduling Algorithm, and the PRGA, i.e. the Pseudo-Random Generation Algorithm. The first one is usually employed for *S*-box initialisation and the second one generates the actual key-stream. In both components, the *swap* function plays an important role in exchanging the positions of two bytes. Researchers argued that the swap function introduces initialisation weakness in the *S*-box, as well as strong biases in the key-stream.

In this paper, the authors have first attempted to find out the optimal number of bytes that should be discarded to remove the biases from RC4. To study whether the security of RC4 really increases if more and more initial bytes are discarded from the key-stream, here N (256), $2N$ (512), $4N$ (1024) and lastly $8N$ (2048) initial bytes are discarded and then the outputs are analysed along with the original algorithm [1], following the guidelines of NIST (National Institute of Standards and Technology), USA, in their Statistical Test Suite, coded by the authors. It has been found that there is a certain optimal level of discarding the output bytes—it is not that discarding as many bytes as possible helps continuously in increasing the security.

Also, different logics have been introduced to modify the PRGA to handle at least two *S*-boxes to generate a more complicated key-stream, which has been analysed along with the model proposed by Paul and Preneel [2]. The authors have also introduced a key-expansion logic, which gives a stronger initialisation to the *S*-box and calculates the initial value of j in the PRGA from the key values, not as 0, as given in the original algorithm [1]—thus giving a more dynamic value to j. RC4 is described as follows:

KSA

```
for i = 0, …,N − 1
S[i] = i;
  next i
j = 0;
for i = 0, …, N − 1
 { j = j + S[i] + K[i]
   swap(S[i], S[j]); }
  next i
```

PRGA

```
  i = 0; j = 0;
  while TRUE
{ i = i + 1
j = j + S[i]
  swap(S[i], S[j]);
z = S[S[i] + S[j]];}
```

2. Existing articles and motivation

Maitra and Paul [1] revealed the non-uniformity in KSA and presented a three-layer architecture in a scrambling phase to remove the weaknesses of the KSA and the PRGA.

*For correspondence
Published online: 22 October 2019

Their modified algorithm, named as RC4+, resolves most of the existing weaknesses of RC4. Paul and Preneel [2] described a new statistical bias in the distribution of the first two output bytes of RC4 and suggested some more random variables in the PRGA to reduce correlation between states, forming a new variant, RC4A. They proposed dropping $2N$ initial bytes. Rivest and Schuldt [3] used extensive simulations to search biases in RC4 and suggested that about 281 output bytes are required to distinguish a new variant. They named the improved variant as 'Spritz'. Mironov [4] identified a weakness in RC4, stemming from an imperfect shuffling algorithm used in the KSA and the PRGA. He argued that discarding the initial $12N$ (or at least $3N$) bytes from the output stream eliminates the possibility of strong attacks. Klein [5] argued that even after many steps of the PRGA, it is possible to exploit the weakness of the KSA. They also analysed RC4A and found that it also has weaknesses, similar to that of RC4—a new attack and a correlation between the output and the internal state of RC4 have been presented.

Roos [6] mentioned some limitations in the KSA by identifying several classes of weak keys for RC4 with some important technical results. He has defined strong correlations between the secret key bytes and the final key-stream generated. Akgün *et al* [7] showed that the KSA leaks information about the secret key if the initial state table is known. A new bias in the KSA has been detected by them, and they proposed the framework of a new algorithm to retrieve the RC4 key in a faster way. Mantin and Shamir [8] noted down the major non-uniformity in the output bytes at the second round of RC4, whose probability is twice the expected value. They pointed out that a cipher-text-only attack can exploit the weakness in broadcast applications.

SenGupta *et al* [9] studied the RC4 designing problem of throughput. An architecture to generate two key-stream bytes per clock using the idea of loop unrolling and pipelining has been implemented. Nawaz *et al* [10] introduced a new 32-bit RC4-like faster key-stream generator with a huge internal state and offered higher resistance against state recovery attacks. Tomašević and Bojanić [11] proposed a new technique to improve cryptanalytic attack on RC4 based on a tree representation of RC4. The strategy has been used to favour more promising values that should be assigned to unknown entries in the RC4 table. Grosul and Wallach [12] observed that by increasing key length, the strength of the RC4 key did not grow linearly and advised discarding the first 256 bytes of the key-stream. They showed that for each 2048-bit key, a family of related keys generates similar key-streams.

Al Fardan *et al* [13] recommended that RC4 should be avoided in TLS (Transport Layer Security) and WPA (Wi-Fi Protected Access). They have shown how plaintext recovery for RC4 is possible from arbitrary positions in the plaintext. Zoltak [14] reported that among 20 RC4-like algorithms, only VMPC-R produces strong pseudo-random output. They concentrated mainly on the PRGA, keeping the KSA same for all. They found statistical weaknesses in almost all of them. Fluhrer and McGrew [15] proved that the joint distribution of two successive bytes differs significantly from the uniform distribution. They computed the joint probability of two consecutive output bytes, which requires much less key-stream bytes. Sepehrdad *et al* [16] presented a tool that may be used as an application of automated discovery of weaknesses in ciphers—this may suggest a new kind of tool for cryptanalysts. They presented several weaknesses in RC4 by this technique.

Church [17] gave a complete list of irreducible polynomials for prime moduli (2, 3, 5, 7 and 11) of each degree. The determination of the exponents provided very satisfactory control of the irreducibility of the polynomials. Daemen and Rijmen [18] defined a process of creating *S*-boxes by a mathematical process in GF(2^8) (Galois Field). The process of calculating the multiplicative inverse of a byte with an irreducible polynomial as modulus has been explained. The publication SP 800-90A of NIST [19] contains specifications for cryptographically secured PRNGs (Pseudo-Random Number Generators), providing some methods based on hash functions, block cipher algorithms or number theoretic problems.

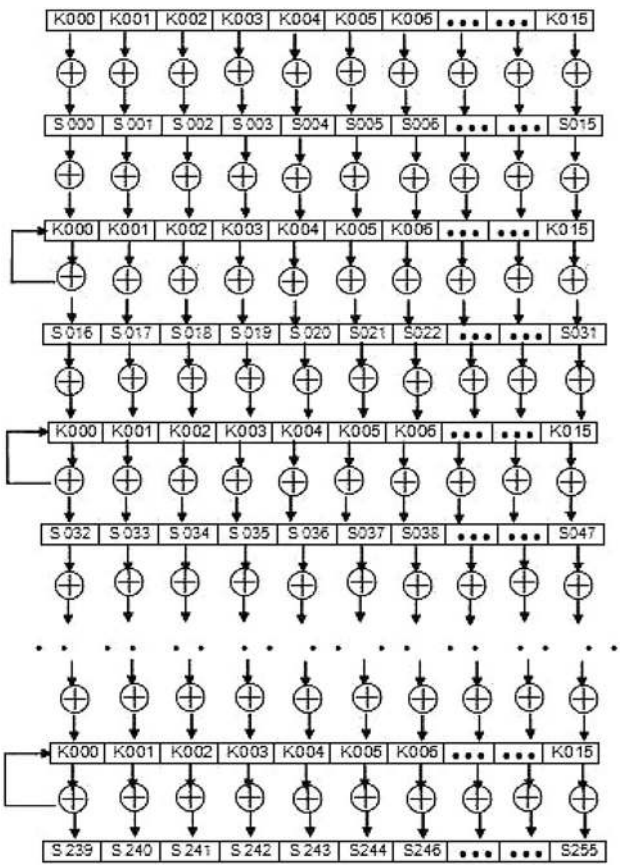

Figure 1. Block diagram of the key-mixing procedure to initialise the RC4 *S*-box (represents addition mod 256).

Table 1. Comparison of (a) POP status and (b) UD status generated by 15 NIST tests for discarded bytes.

Test ↓	RC4	RC4_*N*	RC4_2*N*	RC4_4*N*	RC4_8*N*
(a) POP Status for NIST tests					
1	0.988000	0.988000	*0.996000* ***1***	0.990000	0.992000
2	*0.992000* ***1***	0.984000	0.988000	0.988000	0.986000
3	0.992000	0.992000	*0.996000* ***.33***	*0.996000* ***.33***	*0.996000* ***.33***
4	0.982000	0.982000	0.982000	*0.988000* ***.5***	*0.988000* ***.5***
5	0.984000	0.980000	0.980000	0.982000	*0.986000* ***1***
6	0.980000	0.994000	*0.996000* ***.5***	0.982000	*0.996000,****5***
7	0.990000	0.992000	0.990000	*0.996000* ***1***	0.995000
8	*0.992000* ***.5***	0.986000	0.990000	*0.992000* ***.5***	0.988000
9	0.982000	*0.996000* ***1***	0.992000	0.992000	0.990000
10	0.992000	0.982000	*0.996000* ***1***	0.988000	0.986000
11	0.982000	*0.994000* ***1***	0.990000	0.991000	0.990000
12	*0.992000* ***.5***	*0.992000* ***.5***	0.990000	0.990000	0.988000
13	0.995000	0.988000	0.980000	*0.998000* ***1***	0.995000
14	0.983500	0.986000	*0.987500* ***.5***	*0.987500* ***.5***	0.986000
15	0.985889	*0.988000* ***1***	0.987000	0.987667	0.986111
Total:	**2**	**3.5**	**3.33**	***3.85***	**2.33**
(b) Uniformity Distribution for NIST tests					
1	4.154218^{-01}	9.626880^{-01}***1***	9.540148^{-01}	8.831714^{-01}	7.981391^{-01}
2	4.904834^{-01}**1**	4.446914^{-01}	9.891987^{-02}	1.087909^{-01}	6.952004^{-01}***1***
3	8.920363^{-01}	4.635119^{-01}	8.891176^{-01}	5.181061^{-01}	3.537331^{-01}
4	5.790211^{-01}	6.454484^{-01}	9.288566^{-01}***1***	2.343734^{-01}	5.831447^{-01}
5	2.492839^{-01}	8.343083^{-01}	8.708559^{-01}***1***	2.596162^{-01}	4.788391^{-02}
6	4.170881^{-02}	1.372823^{-01}***1***	1.146836^{-03}	1.509358^{-03}	4.901567^{-05}
7	8.272794^{-01}***1***	8.055687^{-01}	4.635119^{-01}	4.749856^{-01}	4.446914^{-01}
8	2.224804^{-01}	6.662452^{-01}	7.359083^{-01}	9.673823^{-01}***1***	9.093595^{-02}
9	3.856456^{-02}	5.872742^{-01}	4.446915^{-01}	3.976884^{-01}	8.343083^{-01}***1***
10	5.462832^{-01}	3.907208^{-01}	6.828233^{-01}	7.034170^{-01}	7.981391^{-01}***1***
11	1.699807^{-01}	6.972574^{-02}	5.558536^{-01}	5.707923^{-01}***1***	1.916867^{-01}
12	2.953907^{-01}	4.226379^{-01}	7.675818^{-01}	7.981391^{-01}***1***	6.204653^{-01}
13	8.201435^{-01}	7.981391^{-01}	1.494948^{-01}	9.705978^{-01}***1***	5.030520^{-02}
14	6.729885^{-02}	8.314506^{-02}	2.822770^{-01}	6.204653^{-01}***1***	6.291943^{-02}
15	8.386675^{-02}	4.515471^{-01}	7.002254^{-01}***1***	5.328562^{-01}	1.503405^{-02}
Total:	**2**	**2**	**3**	***5***	**3**

3. Proposed modifications to RC4

Roos [6] observed a strong correlation in RC4 between the *S*-box and the key. He pointed out the *swap* function of KSA as responsible for this while initialising the *S*-box. The line $j = j + S[i] + K[i]$, preceding the swap function, calculates the indices i and j, where i is deterministic and j is pseudo-random. He argued that by this way, a particular element may change its position once or more, or may not relocate at all,—which becomes a major weakness in the cipher. According to his analysis, there is a high probability of about 37% that an element is swapped not even once. To get rid of this weakness, he proposed to discard at least N number of initial bytes from the key-stream.

Mironov [4] also identified the same kind of weaknesses in RC4 key-stream due to *improper swap function*, as he mentioned, as the prime cause of bias. He observed that the bias appears up to the first $2N$ or $3N$ bytes. Using an abstract model, he calculated a maximum number of $12N$ initial bytes to be dumped, to get a safe key-stream.

In this paper, an attempt has been made to identify the optimum number of bytes to be discarded from RC4 key-stream before starting the actual encryption. Undoubtedly, dumping more and more bytes from the output stream, as a result, does not keep on increasing the security. Here, a total of five key-streams have been analysed, generated by discarding 0, N, $2N$, $4N$ and $8N$ ($N = 256$) bytes, in separation; the first one is the original cipher, and others are identified as RC4_*N*, RC4_2*N*, RC4_4*N* and RC4_8*N*, respectively.

Moreover, a new method of initialising the RC4 *S*-box has been proposed through modular arithmetic. The 256 cells of the *S*-box have been filled up by the multiplicative inverses of the 256 bytes (0–255), where an irreducible polynomial [16, 17, 20] in GF(2^8) has been used as the modulus [21, 22]. By this way, the *S*-box can be initialised

Table 2. POP status and UDs for (a) RC4, (b) RC4_N, (c) RC4_2N, (d) RC4_4N and (e) RC4_8N.

Test ↓	Expected POP	Observed POP	Status	Uniformity Distribution	Status
(a)					
1	0.976651	0.988000	Successful	4.154218^{-01}	Uniform
2	0.976651	0.992000	Successful	4.904834^{-01}	Uniform
3	0.976651	0.992000	Successful	8.920363^{-01}	Uniform
4	0.976651	0.982000	Successful	5.790211^{-01}	Uniform
5	0.976651	0.984000	Successful	2.492839^{-01}	Uniform
6	0.976651	0.980000	Successful	4.170881^{-02}	Uniform
7	0.976651	0.990000	Successful	8.272794^{-01}	Uniform
8	0.976651	0.992000	Successful	2.224804^{-01}	Uniform
9	0.976651	0.982000	Successful	3.856456^{-02}	Uniform
10	0.976651	0.992000	Successful	5.462832^{-01}	Uniform
11	0.980561	0.982000	Successful	1.699807^{-01}	Uniform
12	0.976651	0.992000	Successful	2.953907^{-01}	Uniform
13	0.980561	0.995000	Successful	8.201435^{-01}	Uniform
14	0.985280	0.983500	Unsuccessful	6.729885^{-02}	Uniform
15	0.986854	0.985889	Unsuccessful	8.386675^{-02}	Uniform
(b)					
1	0.976651	0.988000	Successful	9.626880^{-01}	Uniform
2	0.976651	0.984000	Successful	4.446914^{-01}	Uniform
3	0.976651	0.992000	Successful	4.635119^{-01}	Uniform
4	0.976651	0.982000	Successful	6.454484^{-01}	Uniform
5	0.976651	0.980000	Successful	8.343083^{-01}	Uniform
6	0.976651	0.994000	Successful	1.372823^{-01}	Uniform
7	0.976651	0.992000	Successful	8.055687^{-01}	Uniform
8	0.976651	0.986000	Successful	6.662452^{-01}	Uniform
9	0.976651	0.996000	Successful	5.872742^{-01}	Uniform
10	0.976651	0.982000	Successful	3.907208^{-01}	Uniform
11	0.980561	0.994000	Successful	6.972574^{-02}	Uniform
12	0.976651	0.992000	Successful	4.226379^{-01}	Uniform
13	0.980561	0.988000	Successful	7.981391^{-01}	Uniform
14	0.985280	0.986000	Successful	8.314506^{-02}	Uniform
15	0.986854	0.988000	Successful	4.515471^{-01}	Uniform
(c)					
1	0.976651	0.996000	Successful	9.540148^{-01}	Uniform
2	0.976651	0.988000	Successful	9.891987^{-02}	Uniform
3	0.976651	0.996000	Successful	8.891176^{-01}	Uniform
4	0.976651	0.982000	Successful	9.288566^{-01}	Uniform
5	0.976651	0.980000	Successful	8.708559^{-01}	Uniform
6	0.976651	0.996000	Successful	1.146836^{-03}	Uniform
7	0.976651	0.990000	Successful	4.635119^{-01}	Uniform
8	0.976651	0.990000	Successful	7.359083^{-01}	Uniform
9	0.976651	0.992000	Successful	4.446915^{-01}	Uniform
10	0.976651	0.996000	Successful	6.828233^{-01}	Uniform
11	0.980561	0.990000	Successful	5.558536^{-01}	Uniform
12	0.976651	0.990000	Successful	7.675818^{-01}	Uniform
13	0.980561	0.980000	Unsuccessful	1.494948^{-01}	Uniform
14	0.985280	0.987500	Successful	2.822770^{-01}	Uniform
15	0.986854	0.987000	Successful	7.002254^{-01}	Uniform
(d)					
1	0.976651	0.990000	Successful	8.831714^{-01}	Uniform
2	0.976651	0.988000	Successful	1.087909^{-01}	Uniform
3	0.976651	0.996000	Successful	5.181061^{-01}	Uniform
4	0.976651	0.988000	Successful	2.343734^{-01}	Uniform
5	0.976651	0.982000	Successful	2.596162^{-01}	Uniform
6	0.976651	0.982000	Successful	1.509358^{-03}	Uniform
7	0.976651	0.996000	Successful	4.749856^{-01}	Uniform

Table 2 continued

Test ↓	Expected POP	Observed POP	Status	Uniformity Distribution	Status
8	0.976651	0.992000	Successful	9.673823^{-01}	Uniform
9	0.976651	0.992000	Successful	3.976884^{-01}	Uniform
10	0.976651	0.988000	Successful	7.034170^{-01}	Uniform
11	0.980561	0.991000	Successful	5.707923^{-01}	Uniform
12	0.976651	0.990000	Successful	7.981391^{-01}	Uniform
13	0.980561	0.998000	Successful	9.705978^{-01}	Uniform
14	0.985280	0.987500	Successful	6.204653^{-01}	Uniform
15	0.986854	0.987667	Successful	5.328562^{-01}	Uniform
(e)					
1	0.976651	0.992000	Successful	7.981391^{-01}	Uniform
2	0.976651	0.986000	Successful	6.952004^{-01}	Uniform
3	0.976651	0.996000	Successful	3.537331^{-01}	Uniform
4	0.976651	0.988000	Successful	5.831447^{-01}	Uniform
5	0.976651	0.986000	Successful	4.788391^{-02}	Uniform
6	0.976651	0.996000	Successful	4.901567^{-05}	Non-uniform
7	0.976651	0.995000	Successful	4.446914^{-01}	Uniform
8	0.976651	0.988000	Successful	9.093595^{-02}	Uniform
9	0.976651	0.990000	Successful	8.343083^{-01}	Uniform
10	0.976651	0.986000	Successful	7.981391^{-01}	Uniform
11	0.980561	0.990000	Successful	1.916867^{-01}	Uniform
12	0.976651	0.988000	Successful	6.204653^{-01}	Uniform
13	0.980561	0.995000	Successful	5.030520^{-02}	Uniform
14	0.985280	0.986000	Successful	6.291943^{-02}	Uniform
15	0.986854	0.986111	Unsuccessful	1.503405^{-02}	Uniform

with 256 random-like values, which is the primary aim of the KSA. This is how one can take care of the controversial swap function. The list of these polynomials is given as follows:

```
11B, 11D, 12B, 12D, 139, 13F, 14D, 15F, 163, 165,
169, 171, 177, 17B, 187, 18B, 18D, 19F, 1A3, 1A9,
1B1, 1BD, 1C3, 1CF, 1D7, 1DD, 1E7, 1F3, 1F5, 1F9.
```

To ensure that the *S*-box is more unpredictable, these initial values have then been merged with the key bytes, by introducing a key-mixing logic, described in the following pseudocode. The logical block diagram of the same is shown in figure 1.

```
for(i=0; i<256; i++)
 if(i<16)S[i]=(S[i]+K[i])%256;
 else S[i]=(S[i]+S[i-16]+K[i%16])%256.
```

By this way, each bit of the key is diffused into several stages, and sufficient nonlinearity is introduced against reconstructing the state array, to make it more resistant to known cryptanalytic attacks. This key-mixing process is simple to implement on any platform, removing biases and weak keys inserted by the *swap* function of KSA.

In the original RC4, values of the *S*-box are only swapped, not updated. Researchers argue on this point that many a value for the *S*-box may not at all move in this process, and thus serious bias occurs in the output of this cipher. Keeping the increase in security and robustness in mind as the main points, the authors tried to minimise the bias by removing the conventional swap from the KSA portion of RC4 and achieving the key-mixing logic.

While updating values with key-mixing, first the newly generated value is stored in a binary tree. Then, a binary search is performed on the tree from 0 to $i - 1$ to search for a duplicate, with time complexity O(log i). If the newly generated value is found to be a duplicate one, the same is incremented by 1 (mod 256), and again a new search is performed, from the very beginning. Once it is confirmed that the newly generated value is unique from $S[0]$ to $S[i]$ up to the Nth value ($N = 256$, here), then only the value is taken as granted. After sufficient experimental and implementational observations, it can be concluded that the aforementioned algorithm is fast and efficient enough, and there is no scope of generating duplicate values once it completes its operation. Future work might be interesting for the mathematicians and researchers in this regard.

Though the aforementioned procedure definitely increases the time complexity of the cipher, it can be mentioned that the key-mixing is used only in place of the KSA, i.e. the *S*-box initialisation only. As the PRGA remains unchanged for the rest of the encryption process, which generally uses data of very large size that may tend to

Table 3. *P*-value distribution for (a) RC4, (b) RC4_N, (c) RC4_2N, (d) RC4_4N and (e) RC4_8*N*.

Test ↓	1	2	3	4	5	6	7	8	9	10
(a)										
1	68	50	48	49	52	48	44	47	49	45
2	49	49	53	39	51	45	62	48	47	57
3	55	59	48	44	54	48	48	49	49	46
4	38	61	50	51	46	60	51	44	51	48
5	58	56	48	50	47	39	44	66	49	43
6	56	47	46	45	59	38	47	70	38	54
7	41	45	46	49	52	52	56	50	49	60
8	46	57	40	57	42	50	64	58	40	46
9	60	55	56	55	57	54	32	42	53	36
10	51	44	56	48	43	46	63	47	52	50
11	99	115	122	104	98	104	82	98	88	90
12	49	57	65	50	54	51	39	50	40	45
13	90	106	105	98	99	104	102	112	98	86
14	438	400	414	382	397	384	399	385	401	400
15	955	854	901	891	948	903	895	935	832	886
(b)										
1	45	48	51	50	51	54	51	54	41	57
2	56	43	41	58	44	58	57	42	49	52
3	49	47	55	53	58	39	46	60	50	43
4	53	43	56	42	52	45	54	55	54	46
5	51	48	53	56	50	51	38	46	49	56
6	68	50	57	50	47	45	51	39	41	50
7	36	50	53	55	49	46	51	52	52	56
8	53	52	60	46	54	46	40	50	53	46
9	53	50	53	47	49	62	47	43	55	41
10	56	47	42	43	42	61	45	54	59	53
11	93	86	97	105	103	92	109	105	97	113
12	45	36	55	56	52	44	56	53	46	57
13	87	102	115	99	106	104	97	101	91	98
14	406	389	365	372	402	433	428	403	432	370
15	913	928	948	897	908	891	893	854	912	856
(c)										
1	46	46	44	59	51	52	50	53	45	55
2	61	53	32	61	47	43	42	52	49	60
3	47	51	53	54	55	44	43	58	48	47
4	45	47	47	52	57	50	55	49	55	47
5	51	51	50	60	51	45	42	45	53	52
6	57	67	55	31	50	38	66	53	43	40
7	37	52	52	56	43	57	40	61	49	53
8	53	55	41	61	41	47	52	54	47	49
9	40	52	62	52	47	47	56	43	55	46
10	53	51	48	56	51	40	54	41	49	57
11	88	87	92	107	105	101	102	104	98	116
12	42	41	47	55	58	50	48	51	54	54
13	96	116	115	92	78	94	97	116	101	95
14	375	373	391	425	407	424	395	402	433	377
15	897	914	942	923	897	925	881	867	884	870
(d)										
1	47	46	43	57	49	53	58	47	46	54
2	60	57	38	52	44	45	40	48	66	50
3	50	48	54	52	53	42	42	59	58	42
4	49	61	37	55	47	63	50	55	38	45
5	51	51	50	55	58	48	40	36	61	50
6	47	61	54	27	47	42	61	69	40	52
7	41	45	58	46	56	41	56	54	55	50
8	50	52	58	51	47	44	47	49	52	52
9	42	60	57	39	50	53	52	41	53	55
10	52	48	47	50	57	40	50	44	54	59
11	87	85	96	106	105	96	99	113	103	112
12	40	43	48	54	56	47	50	57	53	54
13	114	85	90	117	92	110	85	83	101	127
14	390	417	423	415	414	394	389	392	403	365
15	901	909	972	895	879	881	906	872	831	956
(e)										
1	44	53	44	59	46	55	52	54	39	54
2	58	50	44	48	53	45	52	42	49	59
3	46	47	57	58	46	48	47	64	49	38
4	52	41	41	58	43	58	50	57	45	55
5	51	51	46	61	59	47	36	46	51	52
6	42	50	49	41	65	36	46	82	36	53
7	43	53	46	50	45	63	38	56	58	48
8	60	39	38	47	55	61	60	48	40	52
9	53	50	54	45	46	62	49	47	48	46
10	55	47	46	45	55	48	53	47	59	47
11	90	84	90	114	103	89	116	100	100	114
12	44	41	45	50	56	54	61	48	48	53
13	89	74	111	84	110	113	103	106	98	112
14	421	373	424	377	444	403	414	386	396	362
15	871	932	971	890	878	877	939	909	855	880

infinity, the increased time complexity of *S*-box initialisation will not eventually affect the overall time complexity of the cipher.

As a next phase, following the proposal of Paul and Preneel [2], multiple *S*-boxes have been used to make the output stream more randomised. Their algorithm of PRGA, marked as RC4_2A here, has been reformed by the authors, where two irreducible polynomials from GF(2^8) acted as moduli to initialise two *S*-boxes; the key-mixing logic has not been utilised in this part. Initial values of j_1 and j_2 have been calculated from key values, not from 0 as in RC4 or its other variants, thus, giving more dynamic values while starting the PRGA. The modified PRGA, marked as RC4_2B, is given as follows:

```
i++;
j1 = j1 + S1[i];
 swap(S1[i],S1[j1]);
z = S1[S1[i]+ S1[j1]];
```

```
j2 = j2 + S2[i];
 swap(S2[i],S2[j2]);
z = S2[S2[i]+S2[j2]].
```

In another attempt, PRGA has been reformed again, where two values of i: i_1 and i_2, for each *S*-box, have been introduced, initialised as 0, forming a new variant. Initial values of j_1 and j_2 have been calculated using the same logic as before. The basic aim is to create the key-stream more random to make it harder for the intruder to break. This variant, marked as RC4_2C, is as follows:

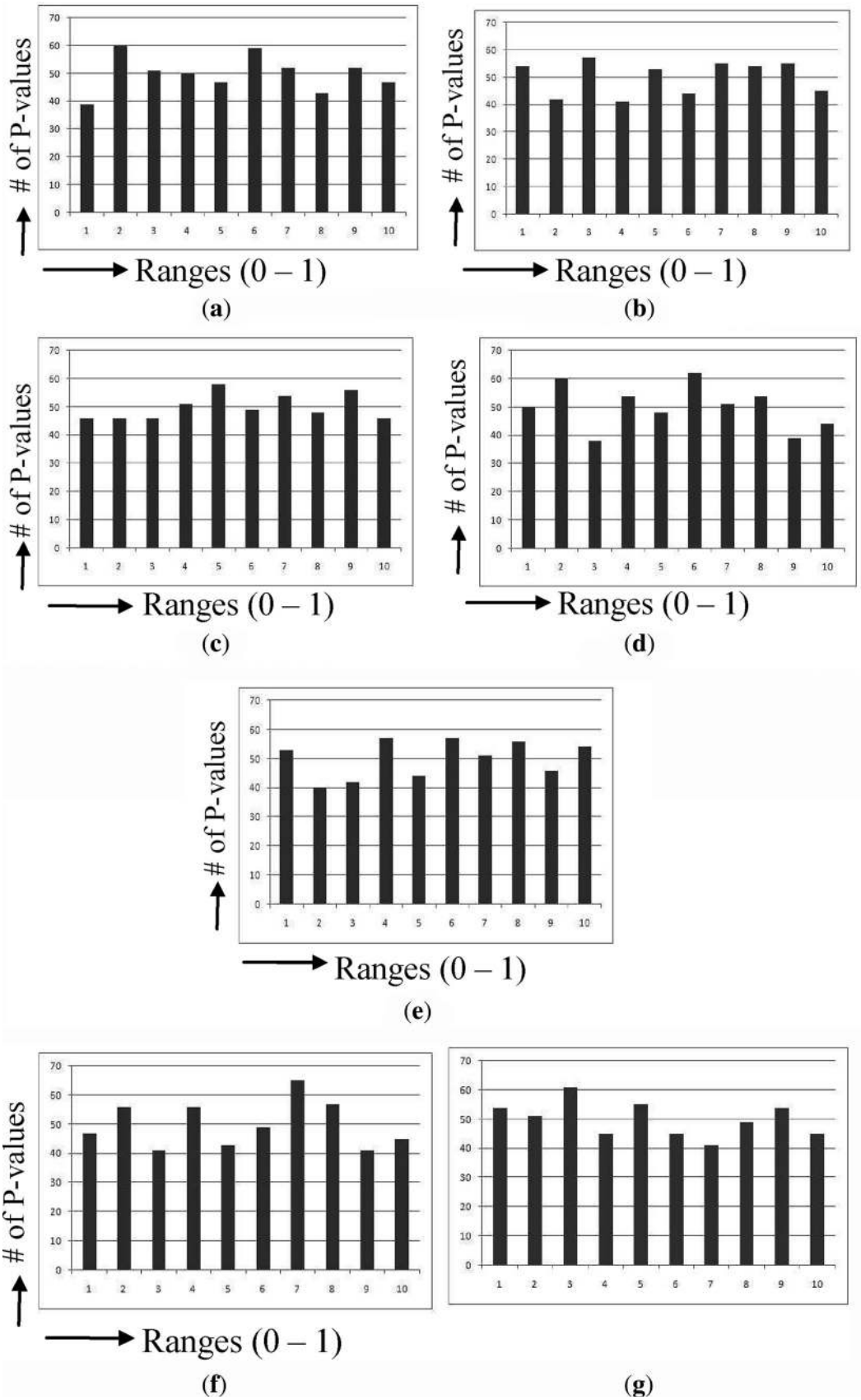

Figure 2. *P*-value distributions of test 4 for **(a)** RC4, **(b)** RC4_*N*, **(c)** RC4_2*N*, **(d)** RC4_4*N*, **(e)** RC4_8*N*. *P*-value distributions of Test 8 for **(f)** RC4, **(g)** RC4_N, **(h)** RC4_2*N*, **(i)** RC4_4*N*, **(j)** RC4_8*N*.

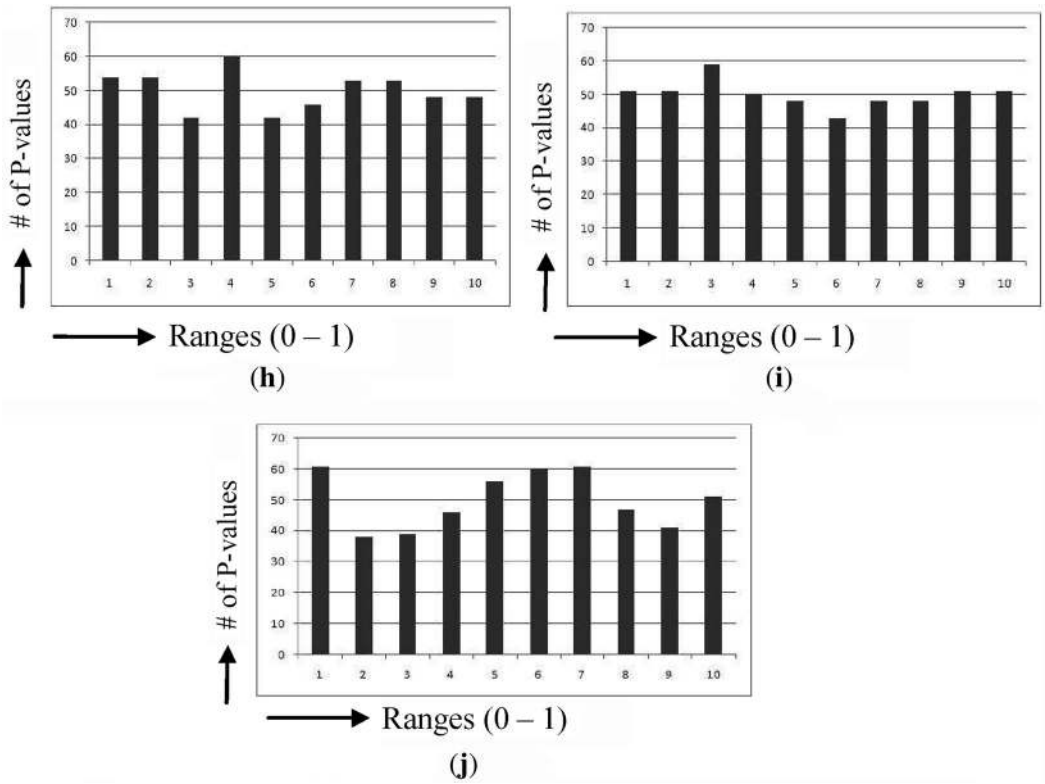

Figure 2. continued

```
i1++;
j1 = j1 + S1[i1];
 swap(S1[i1],S1[j1]);
z = S1[S1[i1]+ S1[j1]];
```

```
i2++;
j2 = j2 + S2[i2];
 swap(S2[i2],S2[j2]);
z = S2[S2[i2]+S2[j2]].
```

Finally, another variant, with a single *S*-box, has been formed by implementing the key-mixing logic, whose initialisation has been done by calculating the multiplicative inverses using modular arithmetic, as described before. This algorithm is much simpler, and its space complexity is also reasonably less. Furthermore, in this way, the KSA has been replaced by a new function. This one is also analysed along with the aforementioned variants to check efficiency.

All these variants of RC4, along with the original one, have been statistically analysed using the NIST Statistical Test Suite, containing 15 different statistical tests to check the robustness of a cipher-text. For each test, a *P*-value (probability value) will be generated, from which two parameters, POP (Proportion of Passing) and UD (Uniformity Distribution), will be calculated to compare the results. For all the algorithms, an example text file has been encrypted 500 times using 500 same encryption keys, generating 500 cipher-texts for each algorithm, each containing 1,342,500 bits, as has been recommended by NIST [23]. Results are then compared to find out how the security varies following the steps of these algorithms.

4. Results and discussion

The analysis proves that though RC4 itself is sufficiently secured, the new variants claim themselves even more efficient than RC4. It has been found that the reformed algorithms create a tweak in RC4 to increase security. First of all, the optimum number of initial key-stream bytes to be discarded is determined by analysing RC4, RC4_*N*, RC4_2*N*, RC4_4*N* and RC4_8*N*. It has been observed that discarding some initial bytes undoubtedly increases the randomness in outputs, but discarding more and more bytes eventually may not increase the security, and at some point, 'the beginning of RC4 ends' [4].

Tables 1(a) and (b) show the results of the analysis, where the POP status and UD of NIST tests [23–25] for these five variants, are compared. Percentages of the probable best results in a particular test for each algorithm have been calculated (shaded in rows), which are then summed at the bottom of the table. The highest count (here, for RC4_4*N*) gives the winner, showing that this one is more robust than the remaining.

POPs and UDs for the five algorithms, compared to the expected values, are shown in tables 2(a)–(e). The earlier two tables (table 1(a) and (b)) have been generated from these five tables. Distributions of *P*-values for the same data sets are displayed in table 3(a)–(e). The interval between 0

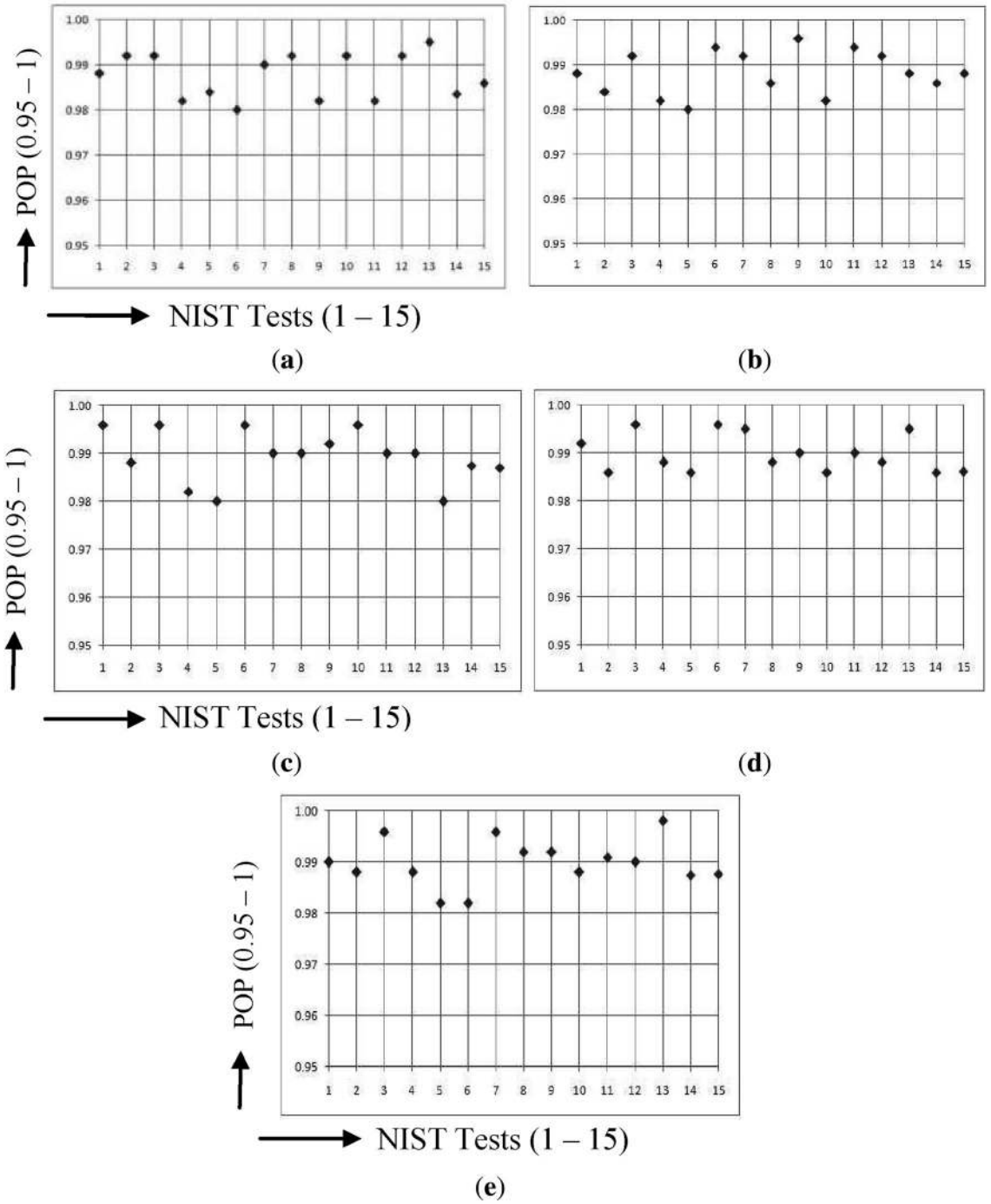

Figure 3. POP status of 15 NIST tests for **(a)** RC4, **(b)** RC4_*N*, **(c)** RC4_2*N*, **(d)** RC4_4*N*, **(e)** RC4_8*N*.

and 1 is divided into 10 sub-intervals, and *P*-values lying within each sub-interval are counted; distribution of *P*-values should be uniform in each sub-interval [23]. Figure 2 shows histograms on the distribution of *P*-values for two tests 4 and 8.

Figure 3 displays scattered graphs on the POP status for the 15 tests, which examine the proportion of sequences that pass a test. A threshold value (expected POP) has been calculated following the guidance given by NIST [23]. If the proportion falls outside of (i.e., less than or below) this expected value, then there is evidence that the data is not random [23]. If most of the values are greater than (or above) this expected value, then the data is considered to be random. For any algorithm, the more the number of POPs that tend to 1 for the 15 tests, the more the data set is assumed to be random.

Table 4. Comparison of (a) POP status and (b) UD status generated by 15 NIST tests for RC4 variants including key-mixing.

Test ↓	RC4	RC4_2A	RC4_2B	RC4_2C	Key-mixing
(a) POP status for NIST tests					
1	0.988000	0.984000	0.990000	*0.994000* ***.5***	*0.990000* ***.5***
2	*0.992000* ***1***	0.994000	0.986000	0.994000	0.998000
3	0.992000	0.990000	0.990000	0.984000	0.990000
4	0.982000	0.982000	*0.994000* ***.5***	*0.994000* ***.5***	0.990000
5	0.984000	*0.992000* ***.5***	0.974000	0.988000	*0.992000* ***.5***
6	0.980000	0.986000	0.984000	0.978000	*0.994000* ***1***
7	0.990000	0.984000	0.992000	0.990000	*0.994000* ***1***
8	0.992000	0.984000	0.990000	*0.994000* ***1***	0.988000
9	0.982000	0.984000	0.980000	*0.994000* ***1***	0.980000
10	0.992000	*0.994000* ***.5***	0.984000	0.986000	*0.994000* ***.5***
11	0.982000	0.990000	*0.995000* ***1***	0.994000	0.986000
12	0.992000	0.986000	0.996000	0.996000	*0.998000* ***1***
13	*0.995000* ***1***	0.989000	0.988000	0.989000	0.982000
14	0.983500	0.985500	0.987250	0.989250	*0.990000* ***1***
15	0.985889	0.983444	0.985444	*0.991556* ***1***	0.991444
Total:	**2**	**1**	**1.5**	**4**	***5.5***
(b) Uniformity Distribution for NIST tests					
1	4.154218^{-01}	8.092489^{-01}	1.223252^{-01}	8.480268^{-01}	8.708559^{-01}***1***
2	4.904834^{-01}	1.237553^{-01}	4.984243^{-01}	2.133093^{-01}	8.801447^{-01}***1***
3	8.920363^{-01}	9.087602^{-01}	9.114125^{-01}***1***	6.620908^{-01}	8.343083^{-01}
4	5.790211^{-01}***1***	3.282969^{-01}	1.855552^{-01}	1.357197^{-01}	5.625911^{-01}
5	2.492839^{-01}	1.311222^{-01}	5.261047^{-01}	2.201592^{-01}	7.636775^{-01}***1***
6	4.170881^{-02}	3.345382^{-01}	2.452120^{-02}	5.101528^{-01}***1***	2.096883^{-01}
7	8.272794^{-01}	5.381822^{-01}	1.835471^{-01}	3.314077^{-01}	9.908191^{-01}***1***
8	2.224804^{-01}	5.996926^{-01}***1***	2.248206^{-01}	7.665814^{-02}	1.087909^{-01}
9	3.856456^{-02}	9.421983^{-01}	2.291461^{-02}	9.442743^{-01}	9.502466^{-01}***1***
10	5.462832^{-01}	9.947196^{-01}***1***	1.062459^{-01}	2.417409^{-01}	1.866758^{-02}
11	1.699807^{-01}	3.537331^{-01}	8.816625^{-01}***1***	8.716150^{-02}	2.368098^{-01}
12	2.953907^{-01}	4.559373^{-01}	5.625912^{-01}	1.756906^{-01}	6.282074^{-01}***1***
13	8.201435^{-01}***1***	8.129051^{-01}	1.068773^{-01}	2.485492^{-02}	1.357197^{-01}
14	6.729885^{-01}	3.175654^{-01}	5.620795^{-01}	4.007594^{-01}	9.368231^{-01}***1***
15	8.386675^{-02}	2.137403^{-03}	5.594693^{-01}***1***	3.222618^{-06}	3.996299^{-02}
Total:	**2**	**2**	**3**	**1**	***7***

Table 5. POP status and UDs for (a) RC4, (b) RC4_2A, (c) RC4_2B, (d) RC4_2C and (e) key-mixing.

Test ↓	Expected POP	Observed POP	Status	Uniformity Distribution	Status
(a)					
1	0.976651	0.988000	Successful	4.154218^{-01}	Uniform
2	0.976651	0.992000	Successful	4.904834^{-01}	Uniform
3	0.976651	0.992000	Successful	8.920363^{-01}	Uniform
4	0.976651	0.982000	Successful	5.790211^{-01}	Uniform
5	0.976651	0.984000	Successful	2.492839^{-01}	Uniform
6	0.976651	0.980000	Successful	4.170881^{-02}	Uniform
7	0.976651	0.990000	Successful	8.272794^{-01}	Uniform
8	0.976651	0.992000	Successful	2.224804^{-01}	Uniform
9	0.976651	0.982000	Successful	3.856456^{-02}	Uniform
10	0.976651	0.992000	Successful	5.462832^{-01}	Uniform
11	0.980561	0.982000	Successful	1.699807^{-01}	Uniform
12	0.976651	0.992000	Successful	2.953907^{-01}	Uniform
13	0.980561	0.995000	Successful	8.201435^{-01}	Uniform

Table 5 continued

Test ↓	Expected POP	Observed POP	Status	Uniformity Distribution	Status
14	0.985280	0.983500	Unsuccessful	6.729885^{-02}	Uniform
15	0.986854	0.985889	Unsuccessful	8.386675^{-02}	Uniform
(b)					
1	0.976651	0.984000	Successful	8.092489^{-01}	Uniform
2	0.976651	0.994000	Successful	1.237553^{-01}	Uniform
3	0.976651	0.990000	Successful	9.087602^{-01}	Uniform
4	0.976651	0.982000	Successful	3.282969^{-01}	Uniform
5	0.976651	0.992000	Successful	1.311222^{-01}	Uniform
6	0.976651	0.986000	Successful	3.345382^{-01}	Uniform
7	0.976651	0.984000	Successful	5.381822^{-01}	Uniform
8	0.976651	0.984000	Successful	5.996926^{-01}	Uniform
9	0.976651	0.984000	Successful	9.421983^{-01}	Uniform
10	0.976651	0.994000	Successful	9.947196^{-01}	Uniform
11	0.980561	0.990000	Successful	3.537331^{-01}	Uniform
12	0.976651	0.986000	Successful	4.559373^{-01}	Uniform
13	0.980561	0.989000	Successful	8.129051^{-01}	Uniform
14	0.985280	0.985500	Successful	3.175654^{-01}	Uniform
15	0.986854	0.983444	Unsuccessful	2.137403^{-03}	Uniform
(c)					
1	0.976651	0.990000	Successful	1.223252^{-01}	Uniform
2	0.976651	0.986000	Successful	4.484243^{-01}	Uniform
3	0.976651	0.990000	Successful	9.114125^{-01}	Uniform
4	0.976651	0.994000	Successful	1.855552^{-01}	Uniform
5	0.976651	0.974000	Unsuccessful	5.261047^{-01}	Uniform
6	0.976651	0.984000	Successful	2.452120^{-02}	Uniform
7	0.976651	0.992000	Successful	1.835471^{-01}	Uniform
8	0.976651	0.990000	Successful	2.248206^{-01}	Uniform
9	0.976651	0.980000	Successful	2.291461^{-02}	Uniform
10	0.976651	0.984000	Successful	1.062459^{-01}	Uniform
11	0.980561	0.995000	Successful	8.816625^{-01}	Uniform
12	0.976651	0.996000	Successful	5.625912^{-01}	Uniform
13	0.980561	0.988000	Successful	1.068773^{-01}	Uniform
14	0.985280	0.987250	Successful	5.620795^{-01}	Uniform
15	0.986854	0.985444	Unsuccessful	5.594693^{-04}	Uniform
(d)					
1	0.976651	0.994000	Successful	8.480268^{-01}	Uniform
2	0.976651	0.994000	Successful	2.133093^{-01}	Uniform
3	0.976651	0.984000	Successful	6.620908^{-01}	Uniform
4	0.976651	0.994000	Successful	1.357197^{-01}	Uniform
5	0.976651	0.988000	Successful	2.201592^{-01}	Uniform
6	0.976651	0.978000	Successful	5.101528^{-01}	Uniform
7	0.976651	0.990000	Successful	3.314077^{-01}	Uniform
8	0.976651	0.994000	Successful	7.665814^{-02}	Uniform
9	0.976651	0.994000	Successful	9.442743^{-01}	Uniform
10	0.976651	0.986000	Successful	2.417409^{-01}	Uniform
11	0.980561	0.994000	Successful	8.716150^{-02}	Uniform
12	0.976651	0.996000	Successful	1.756906^{-01}	Uniform
13	0.980561	0.989000	Successful	2.485492^{-02}	Uniform
14	0.985280	0.989250	Successful	4.007594^{-01}	Uniform
15	0.986854	0.991556	Successful	3.222618^{-06}	Non-uniform
(e)					
1	0.976651	0.990000	Successful	8.708559^{-01}	Uniform
2	0.976651	0.998000	Successful	8.801447^{-01}	Uniform
3	0.976651	0.990000	Successful	8.343083^{-01}	Uniform
4	0.976651	0.990000	Successful	5.625911^{-01}	Uniform
5	0.976651	0.992000	Successful	7.636775^{-01}	Uniform

Table 5 continued

Test ↓	Expected POP	Observed POP	Status	Uniformity Distribution	Status
6	0.976651	0.994000	Successful	2.096883^{-01}	Uniform
7	0.976651	0.994000	Successful	9.908191^{-01}	Uniform
8	0.976651	0.988000	Successful	1.087909^{-01}	Uniform
9	0.976651	0.980000	Successful	9.502466^{-01}	Uniform
10	0.976651	0.994000	Successful	1.866758^{-02}	Uniform
11	0.980561	0.986000	Successful	2.368098^{-01}	Uniform
12	0.976651	0.998000	Successful	6.282074^{-01}	Uniform
13	0.980561	0.982000	Successful	1.357197^{-01}	Uniform
14	0.985280	0.990000	Successful	9.368231^{-01}	Uniform
15	0.986854	0.991444	Successful	3.996299^{-02}	Uniform

Table 6. *P*-value distribution for (a) RC4, (b) RC4_2A, (c) RC4_2B, (d) RC4_2C and (e) key-mixing.

Test ↓	1	2	3	4	5	6	7	8	9	10
(a)										
1	68	50	48	49	53	48	44	47	49	45
2	49	49	53	39	52	45	62	48	47	57
3	55	59	48	44	55	48	48	49	49	46
4	38	61	50	51	47	59	51	44	51	48
5	58	56	48	50	48	39	44	66	49	43
6	56	47	46	45	60	38	47	70	38	54
7	41	45	46	49	53	52	56	50	49	60
8	46	57	40	57	43	50	64	58	40	46
9	60	55	56	55	58	54	32	42	53	36
10	51	44	56	48	44	46	63	47	52	50
11	99	115	122	104	99	104	82	98	88	90
12	49	57	65	50	55	51	39	50	40	45
13	90	106	105	98	100	104	102	112	98	86
14	438	400	414	382	398	384	399	385	401	400
15	955	854	901	891	949	903	895	935	832	886
(b)										
1	59	55	50	43	44	49	51	45	52	54
2	41	61	36	58	54	58	51	59	46	38
3	52	45	58	49	44	55	53	50	44	52
4	59	41	55	56	51	45	53	43	40	59
5	46	57	47	52	68	52	50	36	50	44
6	52	56	46	51	57	39	60	54	36	51
7	42	51	54	41	49	52	64	46	54	49
8	53	56	57	46	42	50	58	48	52	40
9	53	55	42	53	51	47	51	44	51	55
10	47	48	51	49	47	50	57	52	46	55
11	101	107	98	107	105	94	116	93	103	78
12	52	60	44	54	51	42	62	50	46	41
13	101	115	100	91	93	106	90	105	96	105
14	441	406	383	400	369	377	412	423	394	397
15	932	965	871	906	863	987	874	910	813	881
(c)										
1	51	64	33	57	44	46	41	49	55	60
2	60	48	47	51	36	57	41	53	55	50
3	47	53	55	47	44	44	52	58	47	51
4	41	56	60	56	41	39	46	53	44	62
5	52	57	43	45	43	43	45	62	54	54
6	55	59	52	45	50	34	63	54	54	34

Table 6 continued

Test ↓	1	2	3	4	5	6	7	8	9	10
7	56	53	46	38	45	56	47	59	61	39
8	52	46	52	66	55	45	49	42	54	39
9	73	43	40	41	41	50	56	54	53	49
10	57	50	50	54	36	48	43	68	54	40
11	103	104	94	89	110	93	96	102	101	108
12	50	59	39	46	56	48	51	44	47	60
13	107	108	110	111	99	119	89	89	87	81
14	403	370	411	371	410	414	426	403	390	404
15	820	836	840	893	955	961	972	919	905	899
(d)										
1	48	41	55	45	46	56	50	51	54	54
2	56	59	66	48	42	44	48	46	41	50
3	55	55	42	42	52	44	51	53	58	48
4	63	48	50	54	59	54	43	39	36	54
5	55	69	43	56	40	47	46	44	46	54
6	58	51	54	43	43	42	43	58	55	53
7	60	47	60	57	41	59	45	42	41	58
8	50	57	66	44	52	41	54	57	41	38
9	49	58	55	53	49	44	44	52	49	49
10	47	60	52	40	39	52	42	48	58	62
11	108	74	94	114	109	100	92	91	115	103
12	57	44	40	59	51	45	36	51	61	56
13	92	105	101	115	80	99	114	123	93	78
14	387	382	375	403	410	406	435	388	428	386
15	752	871	947	925	858	932	870	932	927	986
(e)										
1	49	55	54	54	57	46	45	48	49	43
2	44	43	47	55	52	54	52	58	50	45
3	43	48	50	46	47	62	52	53	53	46
4	46	47	47	55	52	50	47	56	37	64
5	48	53	36	53	48	50	49	52	51	61
6	55	64	51	42	46	42	49	71	40	40
7	47	51	53	54	50	50	46	56	43	50
8	53	52	44	43	47	60	46	67	43	39
9	49	47	40	54	49	55	51	49	49	57
10	43	56	45	65	61	37	63	49	40	41
11	105	90	108	113	105	116	92	93	97	81
12	55	34	54	56	50	65	42	59	44	41
13	106	81	114	101	91	107	96	119	83	102
14	403	385	377	418	391	396	412	409	400	409
15	853	918	929	962	950	882	886	855	843	922

Thus, the observation is that discarding too many of the initial key-stream bytes does not help for increasing the security of RC4; it requires an optimum value in between to be maintained, which in the current data set has been found to be $4N$, i.e. here 1024.

The next analysis is based on multiple *S*-boxes in RC4 (RC4_2A, RC4_2B and RC4_2C), along with the newly proposed variant using modular arithmetic and key-mixing logic to initiate the *S*-box. The results are shown in tables 4(a) and (b).

Now it is clear from these two tables that obviously the new variant with key-mixing logic is able to establish itself as the best, even with a simpler logic and a single *S*-box among the other variants. Hence, it can now be decided that using only one *S*-box and an unchanged PRGA has a better time and space management than the variants with multiple *S*-boxes. POPs and UDs of this data set are depicted in table 5(a)–(e), and distributions of *P*-values generated by the same are included in tables 6(a)–(e) as per the rules stated earlier.

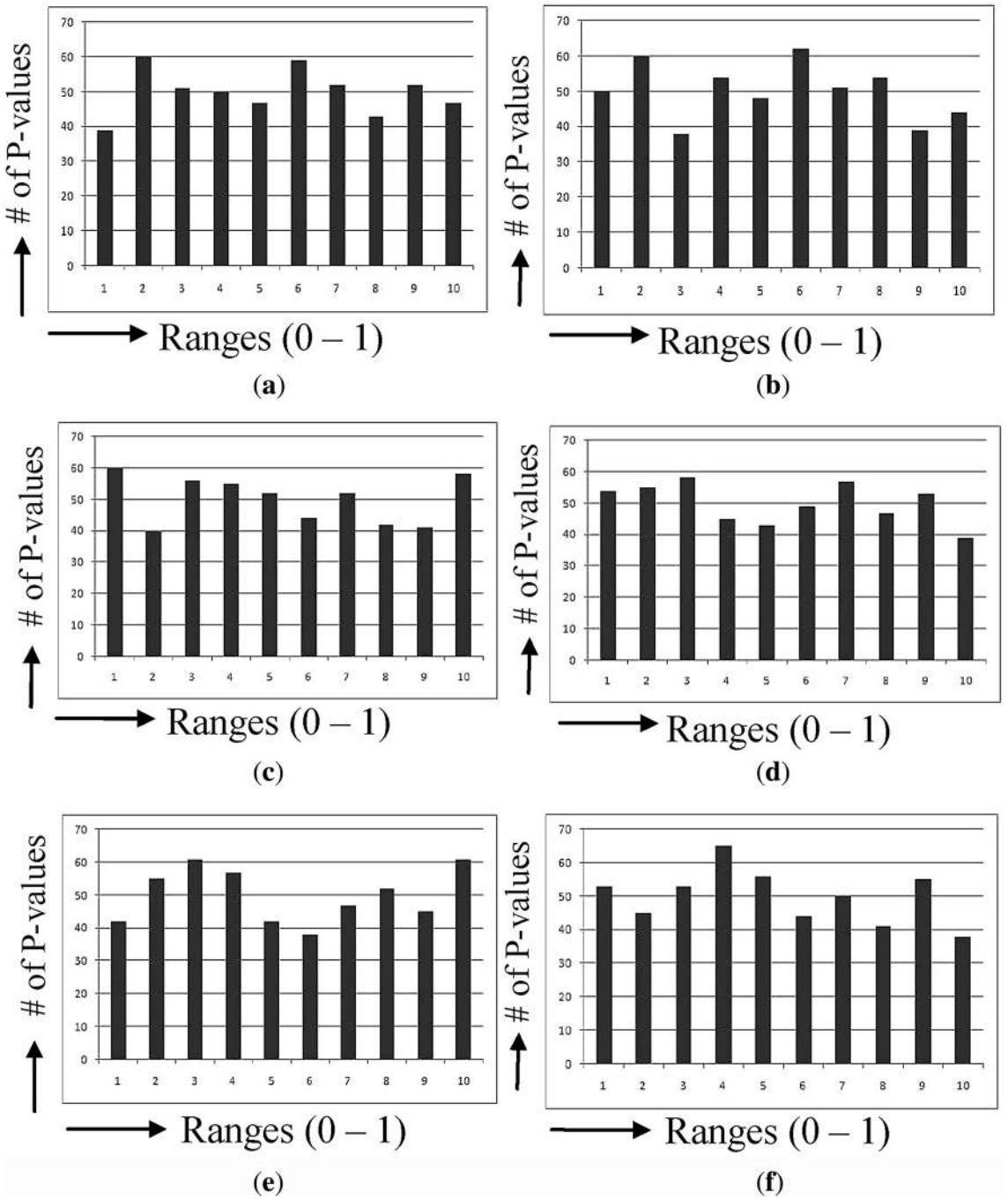

Figure 4. **(a)** and **(b)** *P*-value distributions of tests 4 and 8 for RC4. *P*-value distributions of tests **(c)** 4 and **(d)** 8 for RC4_*N*. **(e)** 4 and **(f)** 8 for RC4_2*N*. **(g)** 4 and **(h)** 8 for RC4_4*N*. **(i)** 4 and **(j)** 8 for RC4_8*N*.

Figure 4 shows histograms of the distribution of *P*-values for two tests 4 and 8 for four variants including original RC4. Scattered graphs on the POPs for the tests 4 and 8 for the same four variants are given in figure 5 to find the most secured variant following the logic as stated earlier.

These figures indicate that for key-mixing logic, more number of points tend to 1, compared with the remaining. Hence, graphically too, it is now evident that this particular variant gives more secured output among the other RC4 alternatives. It also points out that if an RC4 variant with

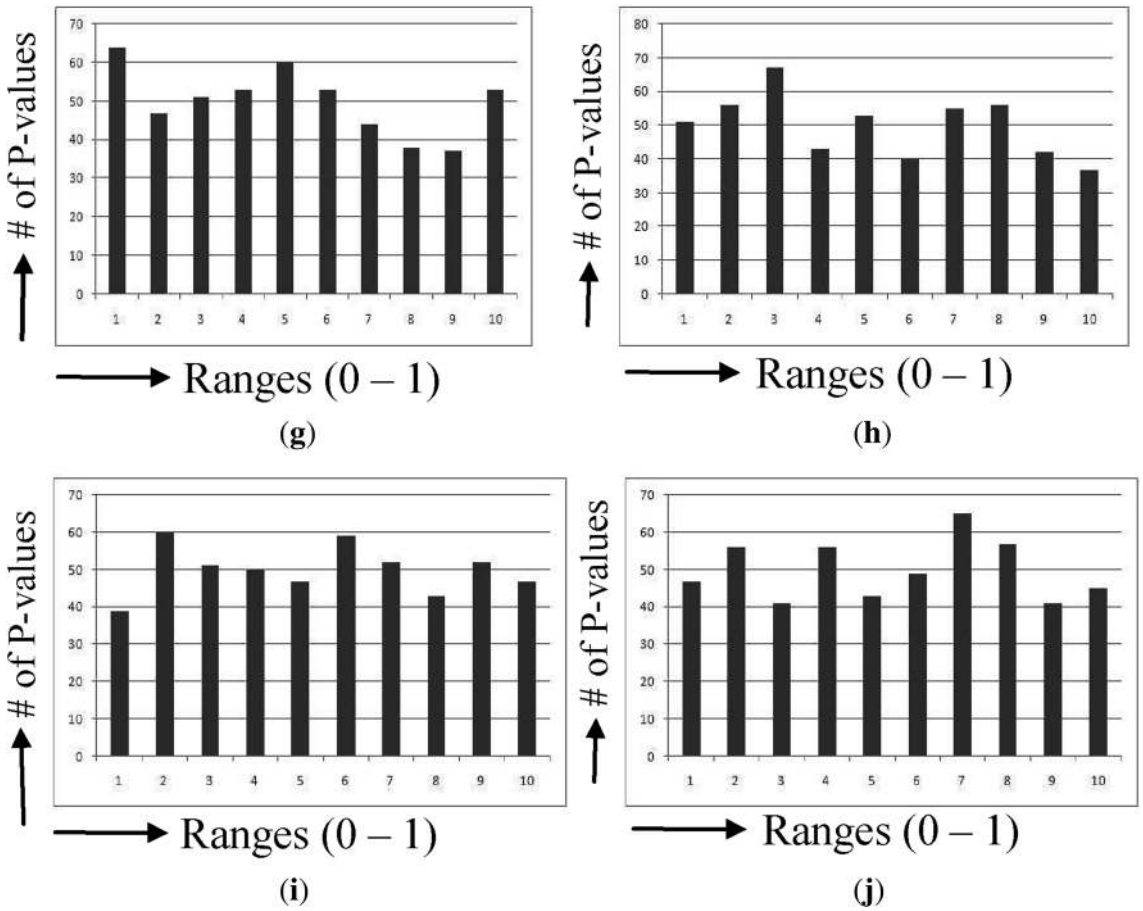

Figure 4. continued

single *S*-box and a simple key-mixing logic give better security than the ones with multiple *S*-boxes, it inevitably possesses much less operational complexity and better space management.

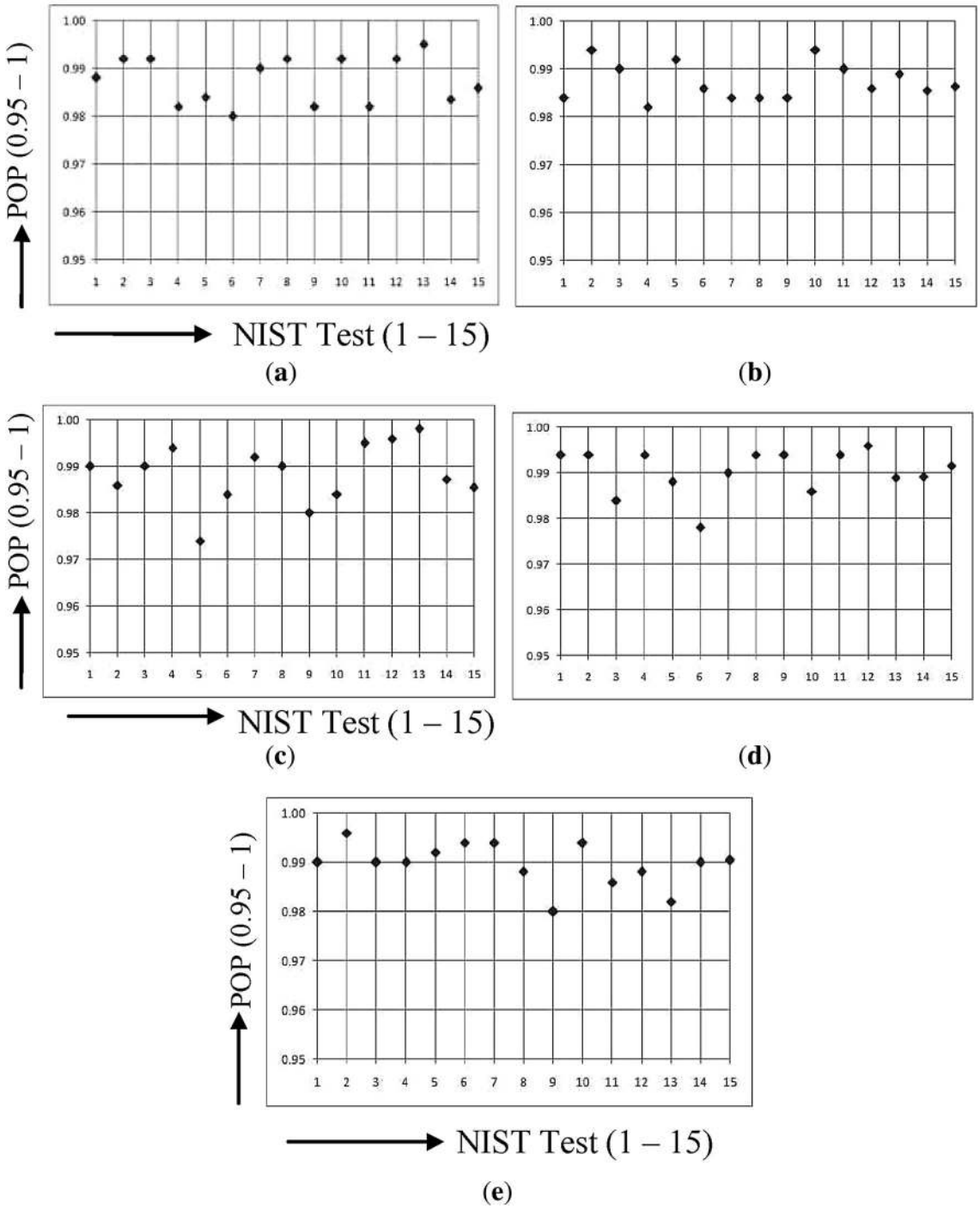

Figure 5. POP status of 15 NIST tests for **(a)** RC4 and **(b)** RC4_2A, **(c)** RC4_2B, **(d)** RC4_2C, **(e)** key-mixing.

5. Conclusion

Besides having enough robustness, one can envisage that the internal state array of RC4 has some limitations and internal biases due to which convenience may be available to the intruders, as argued by a number of researchers. The security of RC4 will be undoubtedly enhanced if the state array can be initialised using any intelligent technique, where it becomes stronger for different applications. Here, by discarding an optimum number of initial bytes and accompanying modified intelligent PRGAs with multiple *S*-boxes, the security of RC4 has been grossly enhanced.

Using the key-mixing logic, it has been found that the security of RC4 can be strongly established without using

multiple *S*-boxes and keeping the original PRGA unchanged. Other mathematical procedures, models, logic and modifications may also be employed to refine RC4 and studies on them are required to find better opportunities to generate more secured key-streams.

References

[1] Maitra S and Paul G 2008 Analysis of RC4 and proposal of additional layers for better security margin. In: *Proceedings of the International Conference on Cryptology in India (INDOCRYPT)*, Lecture Notes in Computer Science (LNCS), Leuven-Heverlee, Belgium: Springer. http://eprint.iacr.org/2008/396.pdf

[2] Paul S and Preneel N 2008 A new weakness in the RC4 keystream generation: an approach to improve the security of the cipher. In: *Proceedings of Fast Software Encryption (FSE)*, LNCS. Heidelberg: Springer. https://www.esat.kuleuven.be/cosic/publications/article-40.pdf

[3] Rivest R L and Schuldt J C N 2014 Spritz—a spongy RC4-like stream cipher and hash function. In: *Proceedings of Advances in Cryptology (CRYPTO) Rump Session*. https://people.csail.mit.edu/rivest/pubs/RS14.pdf

[4] Mironov I 2002 (Not So) Random shuffles of RC4. In: *Proceedings of Advances in Cryptology (CRYPTO)*, California. LNCS. https://eprint.iacr.org/2002/067.pdf

[5] Klein A 2006 *Attacks on the RC4 stream cipher*. Department of Pure Mathematics and Computer Algebra, Ghent University, Belgium. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.484.3279&rep=rep1&type=pdf

[6] Roos A 1995 A class of weak keys in the RC4 stream cipher. *Post in sci.crypt*. http://www.impic.org/papers/WeakKeys-report.pdf

[7] Akgün M, Kavak P and Demicri H 2008 New results on the key scheduling algorithm of RC4. In: *Proceedings of the International Conference on Cryptology in India (INDOCRYPT)*, LNCS. http://link.springer.com/content/pdf/10.1007/978-3-540-9754-5_4.pdf

[8] Mantin I and Shamir A 2002 A practical attack on broadcast RC4. In: *Proceedings of Fast Software Encryption (FSE)*, LNCS 2355, Berlin, Heidelberg: Springer-Verlag, pp. 152–164. http://ai2-s2-pdfs.s3.amazonaws.com/74c2/63425da9c375ab385d1e2954eb27137f9e6b.pdf

[9] Sen Gupta S, Chattopadhyay A, Sinha K, Maitra S and Sinha B P 2013 High-performance hardware implementation for RC4. *IEEE Transactions on Computers* 82: 4

[10] Nawaz Y, Gupta K C and Gong G 2005 A 32-bit RC4-like key-stream generator. *International Association for Cryptologic Research (IACR)*, e-print archive. https://eprint.iacr.org/2005/175.pdf

[11] Tomašević V and Bojanić S 2004 Reducing the state space of RC4. In: *Proceedings of the International Conference on Computational Science* (*ICCS*), LNCS 3036. Berlin–Heidelberg: Springer-Verlag, pp. 644–647. https://doi.org/10.1007/978-3-540-24685-5_110

[12] Grosul A L and Wallach D S 2000 A related-key cryptanalysis of RC4. In: *Proceedings of Defense Advanced Research Projects Agency (DARPA)*, Air Force Research Laboratory (USAFRL), Department of Computer Science, Rice University, F30602-97-2-298. http://pubs.cs.rice.edu/sites/pubs.cs.rice.edu/files/A%20Related-Key%20Cryptanalysis%20of%20RC4%2C.pdf

[13] Al Fardan N J, Bernstein D J, Paterson K G, Poettering B and Schuldt J C N 2013 *On the security of RC4 in TLS and WPA*. Information Security Group, Royal Holloway, University of London. http://www.isg.rhul.ac.uk/tls/RC4biases.pdf

[14] Zoltak B 2014 *Statistical weaknesses in 20 RC4-like algorithms*. https://eprint.iacr.org/2014/315.pdf

[15] Fluhrer S C and McGrew D A 2000 Statistical analysis of alleged RC4. In: *Proceedings of the 7th International Workshop on Fast Software Encryption*, LNCS. Berlin Springer. http://www.mindspring.com/∼dmcgrew/rc4-03.pdf

[16] Sepehrdad P, Vaudenay P and Vuagnoux M 2010 Discovery and exploitation of new biases in RC4. In: *Proceedings of the 17th International Workshop on Selected Areas in Cryptography*, Waterloo, Ontario, Canada, LNCS. Berlin, Heidelberg: Springer, vol. SAC10, pp. 74–91. https://doi.org/10.1007/978-3-642-19574-7_5

[17] Church R 1935 Tables of irreducible polynomials for first four prime moduli. *The Annals of Mathematics* (2nd Series) 36: 198–209. http://www.jstor.org/stable/1968675

[18] Daemen J and Rijmen V 1999 *AES proposal: Rijndael*, Version 2. National Institute of Standard and Technology (NIST), USA. http://csrc.nist.gov/encryption/aes

[19] National Institute of Standard and Technology (NIST), USA 2013 *Recommendation for random number generation using deterministic random bit generators*. http://csrc.nist.gov/publications/nistpubs/800-90A/SP800-0A.pdf

[20] Foruzan B 2007 *Cryptography and network security*. New Delhi: Tata McGraw-Hill

[21] The Federal Information Processing Standard (FIPS) 2001 *Announcing AES*. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[22] FIPS Publication 197 2010 *The official AES standard*. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[23] NIST, Technology Administration, U. S. Department of Commerce 2010 A statistical test suite for RNGs and PRNGs for cryptographic applications. http://csrc.nist.gov/publications/nistpubs800/22rec1SP800-22red1.pdf

[24] Kim, S J, Umeno K and Hasegawa A 2004 *Corrections of the NIST statistical test suite for randomness*. Communications Research Laboratory, Incorporated Administrative Agency, Tokyo, Japan. https://eprint.iacr.org/2004/018.pdf

[25] Gong G, Gupta K C, Hell M and Nawaz Y 2005 Towards a general RC4-like key-stream generator. In: *Proceedings of the First Sklois Conference*, Department of Electrical and Computer Engineering, University of Waterloo. http://avierfjard.com/PDFs/Cryptography/RC4%20Stream%20Cipher/Towards%20a%20General%20RC4like%20Key-stream%20Generator.pdf