Pseudo-Random Number Generator RC4 Period Improvement

A. G. Chefranov
Eastern Mediterranean University, Famagusta, North Cyprus and Taganrog State University of Radio Engineering, Taganrog, Russia, Alexander.chefranov@emu.edu.tr

and T. A. Mazurova
Taganrog State University of Radio Engineering, Taganrog, Russia, mta777@rambler.ru

***Abstract*- A new pseudo-random number generator algorithm RC4E, having period significantly greater than RC4 has, is proposed. It is an extension of RC4. It uses RC4 as a main engine and Heap's algorithm as a second engine, providing enumeration of all possible permutations. Second engine is invoked periodically and it resets the current state of the main engine. Experiments, conducted on the both algorithms, showed their good statistical properties and practically same performance.**

I. Introduction

Pseudo-random number generators (PRNG) are widely used [1-4] nowadays in computer systems for data encryption and simulation. They provide sequences of numbers statistical properties of which are similar to the random numbers with a uniform distribution. Contrary to really random numbers, pseudo-random numbers can be repeated if initial state for PRNG will be the same. Thus, sending and receiving sides use the same initial state and can produce the same sequence of the numbers. An encryption is made by, for example, the byte wise XOR operation on the plaintext bytes and the bytes of a generated sequence, while decryption uses the same operation but on the cipher text and the generated sequence of the numbers.

Algorithm RC4 is one of the most widely used PRNG [5-8] (review of PRNG can be found in [2, 7, 8]). Security strength of PRNG depends on actual characteristics of generated sequences. A period of a generated sequence is one of the most important characteristics of PRNG. A period is a number of generated numbers after generation of which, the sequence of the numbers will repeat. Formally, if $S_i, i=\overline{1,N}$, is a sequence of the generated numbers, a period P is a minimal positive number such that $(\forall i=\overline{1,N-P})(S_{i+P}=S_i)$. It is claimed in [7, 8], that the period of RC4 is about 10^{100}, but proofs are not provided. The algorithm RC4 uses pseudo-random permutations [4] to generate pseudo-random numbers. The maximal number of not repeated permutations of n numbers is known to be n!. The algorithm RC4 uses permutations of n=256 numbers, hence a period of RC4 may be about 256!, which is significantly greater than 10^{100} since due to Stirling's formula [9]

$$n! \approx n^n e^{-n}\sqrt{2\pi n}, \text{ when } n \to \infty,$$

one can write

$$256! \approx 256^{256} e^{-256}\sqrt{2\pi \cdot 256} \approx 10^{500}. \quad (1)$$

In this paper, we present an extension of the RC4 algorithm which guarantees period of generated sequences of the order of (1) and preserves good randomness. The paper is organized as follows. Section II presents details of the RC4 generator. Section III introduces an extension of RC4, an algorithm RC4E. Section IV presents the results of the experimental evaluation of the statistical characteristics of the both algorithms. Section V contains conclusion.

II. Details of RC4 Generator

We follow [8] in description of RC4. A variable-length key of from 1 to 256 bytes is used to initialize a 256-byte state vector S, with elements S[0], S[1],.., S[255]. The vector S contains a permutation of all 8-bit numbers from 0 to 255 at an each time instant. For encryption and decryption, a byte k is generated from S by selecting one of the 256 entries in a systematic fashion. After the next value of k is generated, entries in S are again permuted.

Initialization of S: Initialization of S is made as follows:

```
/*Initialization part*/
1 for (i=0, 255){
2  S[i]=i;
3  T[i]=K[i mod keylen];
4 }
```

where a temporary array *T* is filled using a key *K* of the length *keylen* (line 3 of Initialization part). After that, the array *T* is used to produce the initial permutation *S*:

```
/*Initial permutation part*/
1 j=0;
2 for (i=0,255){
3  j=(j+S[i]+T[i]) mod 256;
4  swap(S[i], S[j]);
5 }
```

Because of 1) the only operation on *S* is a swap (line 4 of Initial permutation part), and 2) we start with permutation (line 2 of Initialization part), after the termination of Initial permutation part, vector *S* still contains all the numbers from 0 to 255.

Stream Generation: Stream generation is given by the following code:

```
/* RC4 Stream generation part*/
1 i=j=0;
2 while (true){
3   i=(i+1) mod 256;
4   j=(j+S[i]) mod 256;
5   Swap(S[i], S[j]);
6   t=(S[i]+S[j]) mod 256;
7   k=S[t];// output of RC4
8 }
```

III. Algorithm RC4 Extension

An idea of the RC4 extension (algorithm RC4E), providing a guaranteed period of the order of the value given by (1), is the following. We incorporate a mechanism for all possible permutations enumeration into RC4 algorithm (which is actually used as a generator of random permutations obtained by swapping of two elements (line 5 of RC4 Stream generation part)). The mechanism will be launched periodically, after generating of *RC4Counter* (*RC4Counter* is a user defined parameter of RC4E algorithm*)* number of permutations, and it will output the next permutation out of possible *256!* permutations. This permutation will be used as the next one in the stream generation part instead of the previous permutation, obtained in the line 5 of RC4 Stream generation part. The first permutation P for this mechanism is the same as used for the RC4 algorithm, i.e. it is obtained by use of Initialization and Initial permutation parts of RC4. To generate all possible permutations, we may use, for example, non recursive Heap's algorithm [10-12] (algorithm below is taken from [11], it provides all permutations of P[1],..,P[N]):

```
/*Heap's algorithm*/
1  for(i=N;i>1;i--)c[i]=1; i=2;
2  process;
3  do{
4     if (c[i]<i){
5      if (odd(i)) k=1;
6      else k=c[i];
7      swap(P[i],P[k]);
8      c[i]=c[i]+1; i=2;
9      process;
10    }
11    else {c[i]=1; i=i+1};
12 }while (i<=N);
```

In the Heap's algorithm above, *process* (lines 2, 9) is a macro which is to be invoked each time when next permutation is ready; *odd(i)* is *true* if *i* is odd, otherwise it returns *false*. Next permutation in the Heap's algorithm is obtained by lines 3-12 and terminates in line 9 (normal termination). Heap's algorithm terminates in line 12, if all N! permutations are already generated. Hence, RC4E algorithm can be represented as follows.

RC4E algorithm.

1. RC4 Initialization part
2. RC4 Initial permutation part
3. Initialize permutation *P=S* (first permutation for change mechanism is the same as an initial permutation for RC4). Initialize *RC4Counter.*
4. Generate next *RC4Counter* permutations by RC4 Stream generation part
5. Find next after *P* permutation by Heap's algorithm; it will be the new *P*.
6. *S=P*
7. Go to line 4.

Consideration of RC4E algorithm shows that its period is not less than the product *RC4Counter*256!* because for each of *256!* possible permutations, produced by Heap's algorithm (second engine), RC4 Stream generation part (main engine) produces *RC4Counter* permutations. If *RC4Counter >>1* then Heap's algorithm will be launched rarely, and performance of RC4E should be slightly worse than that of RC4. Statistical quality of outputs should be close for both algorithms.

IV. Experimental Results

The following tests [13, 14] were used for the comparison:

1) a number of 1-s in a generated bit sequence (length>=20000 bit); its statistics is $F_1 = \frac{1}{n}(\sum_{s=1}^{K}\frac{Q_s^2}{p_s}) - n$, where *K=2*, p_s is an estimated probability of appearance of the s-th value, Q_s is a really counted number of appearances of the s-th value in the sequence of n bits, $s = \overline{0,1}$; it must have an asymptotic distribution as χ^2 with *K-1* levels of freedom;

2) a number of an m-bit vectors; its statistics is given by: $F_2 = \frac{2^m}{K}(\sum_{i=0}^{2^m-1} N_i^2) - K$, where N_i is the frequency of appearance of an *i*-th vector in their sequence of *n* vectors; it must have an asymptotic distribution as χ^2 with $2^m - 1$ levels of freedom;

3) a number of sequences of 1-s and 0-s; its statistics is $F_3 = \sum_{i=1}^{K}\frac{(B_i - E_i)^2}{E_i} + \sum_{i=1}^{K}\frac{(G_i - E_i)^2}{E_i}$, where B_i is a number of series of 0-s of the length *i*, G_i is a number of series of 1-s of the length *i*, $E_i=(n-i+3)/(2^{i+2})$ is an expected number of

series, K is a maximal integer i such that $E_i > 5$; it must have an asymptotic distribution as χ^2 with $2K-2$ levels of freedom;

4) a coefficient of a sequential correlation showing dependency of the next symbol on the previous one, calculated as $F_4 = \frac{n(\sum_{i=0}^{n-2} U_i U_{i+1} + U_{n-1}U_0) - (\sum_{i=0}^{n-1} U_i)^2}{n\sum_{i=0}^{n-1} U_i^2 - (\sum_{i=0}^{n-1} U_i)^2}$. The value of this statistics must be in the range $[\mu_n - 2\sigma_n, \mu_n + 2\sigma_n]$ in 95% of occasions, where $\mu_n = -\frac{1}{(n-1)}$, $\sigma_n = \frac{1}{(n-1)}\sqrt{\frac{n(n-3)}{n+1}}$, n>2;

5) a size Sz after compression of the sequence by the archive utility WinRar as a ratio of the resulting size to the initial one, measured in percents. It shows a level of predictability of a sequence. A good sequence must not be compressed by such the utilities.

The results, presented in Table 1, were obtained for *RC4Counter=3500* on IBM PC with Pentium 1.7GHz, 256 Mb RAM. These results show that the statistical characteristics of the both algorithms are close to each other and satisfy conditions imposed on a good PRNG. The performance of the algorithms is also practically same.

TABLE I
RESULTS OF TESTING OF ALGORITHMS RC4E AND RC4 ON SEQUENCES OF 25 000 AND 250 000 BYTES

	RC4E		RC4	
Number of Bytes	25 000	250 000	25 000	250 000
Number of 1-s; F_1; probability of such value appearance	100 071; 0.1; [0.05; 0.25]	1 000 184; 0.07; [0.05; 0.25]	100 094; 0.18; [0.25;0.50]	999 888; 0.025; [0.05; 0.25]
F_2; value of m; probabilities	258.09; 8; [0.50; 0.75]	236.15; 8; [0.05; 0.25]	253.62; 8; [0.25; 0.50]	248.56; 8; [0.25; 0.50]
F_3; value of k; probabilities	18.56; 14; [0.05; 0.25]	52.32; 17; [0.95; 0.99]	30.62; 14; [0.75; 0.95]	35.2; 17; [0.50; 0.75]
F_4; acceptable range	-0.0065; [-0.013; 0.013]	1.1e-4; [-0.004; 0.004]	-0.003; [-0.013; 0.013]	1e-4; [-0.004; 0.004]
Sz, %	100	100	100	100
Duration, sec	0.45	4.8	0.45	4.71

V. CONCLUSION

In this paper, we present a new algorithm RC4E of the pseudo-random numbers generating, which is obtained as an extension of the RC4 algorithm. Algorithm RC4E uses incorporated in RC4 simple Heap's algorithm for periodic change of RC4 state. This extra job is performed not frequently and does not produce substantial overhead in respect to RC4 that is shown in the experiments. Algorithm RC4E requires extra memory for keeping one more permutation in comparison with RC4. The algorithm RC4E shows practically the same statistical and space/time characteristics as the algorithm RC4, but contrary to RC4, the algorithm RC4E has the guaranteed period of the order of 256! that is significantly greater than the guessed period of RC4, estimated as 10^{100}. Area of applicability of the both algorithms seems to be the same.

REFERENCES

[1] M. Alioto, S. Bernardi, A. Fort, S. Rocchi, and V. Vignoli, "On the stability of digital maps for integrated pseudo-RNGs," in *ECCTD-03 – European Conf. on Curcuit Theory and Design*, Sept. 1-4, 2003, Cracow: Poland, pp. III-349-III-352, 2003.

[2] B.M. Gammel, "Hurst's rescaled range statistical analysis for pseudo-random number generators used in physical simulations," *Phys. Rev. E*, vol. 58 , No. 2, pp. 2586-2597, 1998.

[3] J. Yang, L. Gao, and Y Zhang, "Improving memory encryption performance in secure processors," *IEEE Trans. Comp.,* vol. 54, No. 5, pp. 630-640, 2005.

[4] M. Naor and O. Reingold, "Constructing pseudo-random permutations with a prescribed structure," *J. Cryptology*, vol. 15, pp. 97-102, 2002.

[5] P.D. Kundarewich, S.J.E. Wilton, and A.J. Hu, "A CPLD-based RC4 cracking system," in *Proc. 1999 Canadian Conf. Electr. Comp. Eng.*, May 9-12, 1999, Shaw Conf. Center, Edmonton, Alberta: Canada, pp. 397-402, 1999.

[6] J. Dg. Golic, "Linear models for a time-variant permutation generator," *IEEE Trans. Inform. Theory*, vol. 45, No. 7, pp. 2374-2382, 1999.

[7] M.J.B. Robshow, *Stream Ciphers*. RSA Lab. Tech. Rep. TR-701, Version 2.0 RSA Laboratories, Redwood City (July 25, 1995)

[8] W. Stallings, *Cryptography and Network Security. Principles and Practice*. 3rd ed., Upper Saddle River: Prentice Hall, Pearson Education Int., 2003.

[9] G.A. Korn, T.M. Korn, *Mathematical Handbook for Scientists and Engineers*. N.Y.: McGraw-Hill Book Company, 1968; Moscow: Nauka, 1977.

[10] D.E. Knuth, *The Art of Computer Programming*. Pre-fascicle 2B. A Draft on Section 7.2.1.2: Generating All Permutations. www-cs-faculty.stanford.edu/~knuth/fasc2b.ps.gz

[11] R. Sedgewick, "Permutation generation methods," *Computing Surveys*, vol. 9, No. 2, pp. 137-164, 1977.

[12] B.R. Heap, "Permutations by interchanges," *Computer J.*, vol. 6, pp. 293-294, 1963.

[13] A.A. Varfolomeyev, A.E. Zukov, M.A. Pudovkina, *Stream cryptosystems. The basic properties and methods of cryptanalysis resistance*. Moscow : PAIMS, 2000 (in Russian).

[14] A.G. Chefranov, T.A. Mazurova, I.D. Sidorov, T.S. Letia, "Orthogonal arrays application to pseudorandom numbers generators and optimization problems," *Proc. 14th Int. Conf. Control Syst. and Computer Sci.*, 2-5 July, 2003, Bucharest: Romania, Politechnica Press, vol. 1, pp. 254-259, 2003.