A 32-bit RC4-like Keystream Generator

Yassir Nawaz[1], Kishan Chand Gupta[2] and Guang Gong[1]

[1]Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, ON, N2L 3G1, CANADA
[2]Centre for Applied Cryptographic Research
University of Waterloo
Waterloo, ON, N2L 3G1, CANADA
ynawaz@engmail.uwaterloo.ca, kgupta@math.uwaterloo.ca,
G.Gong@ece.uwaterloo.ca

Abstract. In this paper we propose a new 32-bit RC4 like keystream generator. The proposed generator produces 32 bits in each iteration and can be implemented in software with reasonable memory requirements. Our experiments show that this generator is 3.2 times faster than original 8-bit RC4. It has a huge internal state and offers higher resistance against state recovery attacks than the original 8-bit RC4. We analyze the randomness properties of the generator using a probabilistic approach. The generator is suitable for high speed software encryption.

Keywords: RC4, stream ciphers, random shuffle, keystream generator.

1 Introduction

RC4 was designed by Ron Rivest in 1987 and kept as a trade secret until it leaked in 1994. In the open literatures, there is very small number of proposed keystream generator that are not based on shift registers. An interesting design approach of RC4 which have originated from exchange-shuffle paradigm [10], is to use a relatively big table/array that slowly changes with time under the control of itself. As discussed by Golić in [5], for such a generator a few general statistical properties of the keystream can be measured by statistical tests and these criteria are hard to establish theoretically. Two RC4 like stream ciphers are VMPC [23] and RC4A [18]. RC4 consists of a table of all the $N = 2^n$ possible n-bit words and two n-bit pointers. In original RC4 n is 8, and thus has a huge state of $log_2(2^8! \times (2^8)^2) \approx 1700$ bits. It is thus impossible to guess even a small part of this state and almost all the techniques developed to attack stream ciphers based on linear feedback shift registers (LFSR) fail on RC4. For weakness of RC4 and most of the known attacks on it see [1–7, 9, 11–22].

The most important parameter in RC4 is n, which was typically chosen as 8 by Ron Rivest in 1987. At that time 8-bit/16-bit processors were available for commercial use. Increase in the word size also results in an exponential increase in memory required to store it and N number of swap operation in Key Scheduling Algorithm. But now we have 32-bit/64-bit very fast processors available for commercial use and the cost of internal internal memory has reduced drastically. So can not we easily take n to be 32?. The immediate problem in original RC4 is $N = 2^{32}$ a huge table size, and the number of swap operations in Key Scheduling Algorithm is N, which is impractical. Over the past 10 years, numerous papers have been written on various aspects of RC4, but to the best of our knowledge none of them has tried to solve this.

In this paper we propose some modifications to the RC4 algorithm so that it can exploit the 32-bit and 64-bit processor architectures without increasing the size of the table significantly. The proposed algorithm is general enough to incorporate different word as well as table sizes. For example with 32-bit word size a table of lenght 256 words can be used. We try to keep the original structure of RC4 as much as possible, however the proposed changes affect some underlying design principles on which the security of RC4 is based. Therefore we analyze the security of the modified RC4 and compare it to the original RC4. The rest of the paper is organized as follows. In section 2 we give a brief description of original RC4. In section 3 we propose a modified RC4 keystream generator. The security of the proposed generator is analyzed in section 4 followed by a performance analysis in section 5. We conclude in section 6.

2 Description of RC4

In this section we give a description of the original RC4. The RC4 algorithm consists of two parts: The key scheduling algorithm (KSA) and the pseudo-random generation algorithm(PRGA). The algorithms are shown in Figure 1 where l is the length of the secret key in bytes, and N is the the size of the array S or the S-box in words. In most applications RC4 is used with a word size $n = 8$ and array size $N = 2^8$. In the first phase of RC4 operation an identity permutation $(0, 1, ..., N-1)$ is loaded in the array S. A secret key K is then used to initialize S to a random permutation by shuffling the words in S. During the second phase of the operation PRGA produce random words from the permutation in S. Each iteration of the PRGA loop produces one output word which constitutes the running keystream. The keystream is bit-wise XORed with the plaintext to obtain the ciphertext. All the operations described in Figure 1 are byte operations ($n = 8$). Most modern processors however operate on 32-bit or 64-bit words. If the word size in RC4 is increased to $n = 32$ or $n = 64$, to increase its performance, the size of array S becomes 2^{32} or 2^{64} bytes which is not practical. Note that these are the array sizes to store all the 32-bit or 64-bit permutations respectively.

```
KSA(K, S)
for i=0 to N − 1
    S[i] = i
j = 0

for i = 0 to N − 1
    j = (j + S[i] + K[i mod l]) mod N
    Swap(S[i], S[j])
-----------------------------------
PRGA(S)
i=0
j = 0
Output Generation Loop
    i = (i + 1) mod N
    j = (j + S[i]) mod N
    Swap(S[i], S[j])
    Output=S[(S[i] + S[j])mod N]
```

Fig. 1. The Key Scheduling Algorithm (KSA) and Pseudo-Random Generation Algorithm (PRGA)

3 Proposed Modification to RC4

We now propose a modification to the original RC4 algorithm which enables us to release 32 bits or 64 bits in each iteration of the PRGA loop. This is done by increasing the word size to 32 or 64 while keeping the array size S much smaller than 2^{32} or 2^{64}. We will denote the new algorithm as RC4(n, m) where $N = 2^n$ is the size of the array S in words, m is the word size in bits, and $n \leq m$. For example RC4(8,32) means that the size of the array S is 256 and each element of S holds 32-bit words. Also we will use the term Z_{2^r} to represent the integer ring modulo 2^r.

If we choose n to be much smaller than m ($m = 32$ or 64) in RC4(n, m), then this results in reasonable memory requirements for the array S. However now the contents of the array S do not constitute a complete permutation of 32 bit or 64 bit words. To maintain the randomness of the output keystream we add an integer addition modulo 2^{32} (2^{64} for $n = 64$) to the state update part of the PRGA. The state update now consists of a swap operation and a single word update operation through an integer addition. The index value selected for the update in array S is the same from where the output is taken. This is to ensure that the keystream word just produced is no longer in the array S. This is important because the array is not a permutation and the size of the array is only a small fraction of all the possible numbers M. Therefore knowledge of any value in the array can reveal important state information to the attacker.

KSA(K, S)
for i=0 to $N-1$
    $S[i] = a_i$
$j = 0$
for $i = 0$ to $N-1$
    $j = (j + S[i] + K[i \bmod l]) \bmod N$
    Swap$(S[i], S[j])$
    $S[i] = S[i] + S[j] \bmod M$

PRGA(S)
$i = 0$
$j = 0$
Output Generation Loop
    $i = (i + 1) \bmod N$
    $j = (j + S[i]) \bmod N$
    Swap$(S[i], S[j])$
    Output=$S[(S[i] + S[j] \bmod M) \bmod N]$
    $S[(S[i] + S[j] \bmod M) \bmod N]$=$S[i] + S[j] \bmod M$

Fig. 2. The Modified Key Scheduling Algorithm (KSA) and Pseudo-Random Generation Algorithm (PRGA) for RC4(n, m)

We are still investigating the potential attacks on the proposed algorithm. For example the integer addition modulo M provides the randomness in the update operation however it also satisfies the relation $S[t \bmod N] = t$ where $t = S[i]+S[j] \bmod M$. To avoid this we are currently experimenting with a shift operation which provides a mixing from higher bits to lower bits. The update operation with the shift is given by $S[t \bmod N]$=$t \ll c$, where c is an odd integer.

A similar modification is made to the KSA. In addition to a swap operation each word is updated through an integer addition. Moreover the initial values of the array S can be selected randomly. These values are precomputed and can be stored in $'a'$ beforehand. We give some randomly selected initial values for RC4$(8, 32)$ in Appendix A. The modified KSA and PRGA are given in Figure 2 where $N = 2^n$, $M = 2^m$ and l is the length of the key in bytes.

4 Security Analysis of RC4(n, m)

In this section we analyze the security of RC4(n, m). We consider the resistance of the generator against state recovery attacks and the randomness properties of the keystream.

4.1 Internal State of RC4(n, m)

Like the original RC4, the security of RC4(n, m) comes from its huge internal state. The size of the internal state of original RC4 is approximately 1700 bits. In case of RC4(n, m) the internal state does not consists of a permutation and it may have repetitions of words. Number of putting 2^m elements into N cells where repetitions are allowed is $(2^m)^N$. Therefore the size of the internal state is simply given by $N^2 \times (2^m)^N$. For example for RC4$(8, 32)$ this number is 8208 bits which is much larger than original RC4. Recovering the internal state of RC4(n, m) is therefore much harder than recovering the internal state of RC4.

4.2 Forward Secrecy in RC4(n, m)

Like most of the key stream generator RC4(n, m) keystream generator can also be represented as a finite state machine. Suppose $N = 2^n$, $M = 2^m$ and $R = \mathrm{Z}_N^2 \times \mathrm{Z}_M^N$. The next state function is $f : R \to R$. Let $(i, j, x_0, x_1, \cdots, x_{N-1}) \in R$ be any state, and $(e, d, y_0, y_1, \cdots, y_{N-1}) \in R$ be the next state of the function f. Then we have $e = i + 1 \bmod N$, $d = j + x_e \bmod N$, $y_d = x_e$, $y_e = x_d$, $y_t = x_t, t \notin \{e, d\}$, $k = x_e + x_d \bmod M$, $y_{k \bmod N} = k$. Output of the cipher is $x_{k \bmod N}$. As seen above we can deterministically write down value of each parameter of the next state. So given a state $(e, d, y_0, y_1, \cdots, y_{N-1})$, we can recover $(i, j, x_0, x_1, \cdots, x_{N-1})$ except x_k because x_k has been replaced. Therefore Without the knowledge of x_k the state function is non invertible.

In original RC4 the state function is invertible. Non invertible state functions are known to cause a significantly shorter average cycle length. If the size of the internal state is s and the next state function is randomly chosen then the average cycle length is about 2^{fracs2}. For a randomly chosen invertible next state function the average cycle length is 2^{s-1}. As s in RC4$(8,32)$ is huge (i.e., 8208) the reduction in cycle length is not a problem.

4.3 Randomness of the Keystream

To analyze the keystream of RC4(n,m) we first state the security principles underlying the design of original RC4. The KSA intends to turn an identity permutation S into a pseudorandom permutation of elements and PRGA generates one output byte from a pseudorandom location of S in every round. At every round the secret internal state S is changed by the swapping of elements, one in a known location and other pointed to by a random index. Therefore we can say that the security of original RC4 depends on the following three factors.

- Uniform distribution of the initial permutation of elements in S.
- Uniform distribution of the value of index pointer j.
- Uniform distribution of the index pointer from which the output is taken (i.e., $(S[i]+S[j]) \bmod N$).

The above three conditions are necessary but not sufficient. The KSA uses a secret key to provide a uniformly distributed initial permutation of the elements in S. The value of the index pointer j is updated by the statement $j = (j + S[i]) \bmod N$. Since the elements in S are uniformly distributed the value of j is also uniformly distributed. By the same argument $(S[i]+S[j]) \bmod N$ is also uniformly distributed. Note that the internal state of RC4 consists of the contents of array S and the index pointer j. The state update function consists of an update of the value of j and the update of the permutation in S through a swap operation given by the statement Swap$(S[i], S[j])$. Since j is updated in a uniformly distributed way, the selection of the locations to be swapped is also uniformly distributed. This ensures that the internal state of RC4 evolves in a uniformly distributed way.

We now consider RC4(n,m). The first difference from original RC4 is that whereas the array S in original RC4 is a permutation of all the 256 elements in Z_{2^8}, the array S in RC4(n,m) only contains 2^n m-bit words out of 2^m possible words in $Z_{2^{\mathrm{m}}}$. Consider the PRGA and assume that the initial permutation of 2^n elements in S is uniformly distributed over $Z_{2^{\mathrm{m}}}$. Then the index pointer j is update by the statement

$$j = j + S[i] \bmod N$$

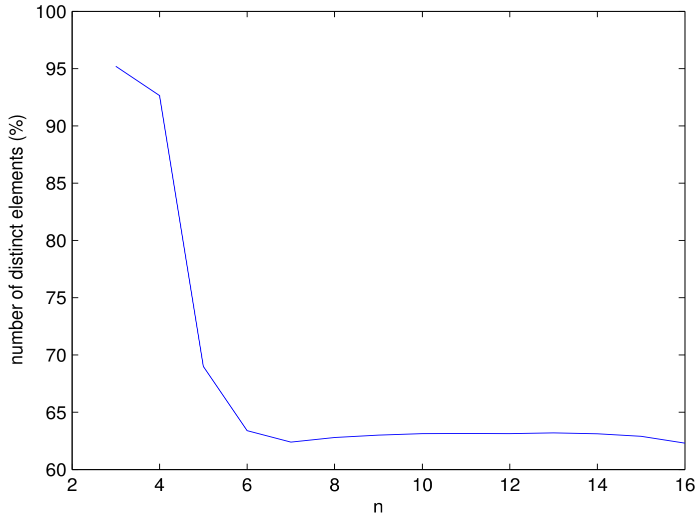

Fig. 3. The effect of n on the percentage of distinct elements observed in 2^{16} word long keystream

where $j \in \mathrm{Z}_{2^\mathrm{n}}$ and $S[i] \in \mathrm{Z}_{2^\mathrm{m}}$. If the value of $S[i]$ is uniformly distributed over $\mathrm{Z}_{2^\mathrm{m}}$, the value of index pointer j is also uniformly distributed over $\mathrm{Z}_{2^\mathrm{n}}$. This implies that the value of the index pointer from which the output is taken (i.e.,$(S[i] + S[j] \bmod M) \bmod N$) is uniformly distributed over $\mathrm{Z}_{2^\mathrm{n}}$. For the above properties to hold during PRGA phase it is essential that the internal state of the RC4(n, m) evolves in a uniformly distributed manner. Recall that in original RC4 the uniform distribution of pointer j was the reason for the state to evolve uniformly since all the 256 elements in Z_{2^8} were present in the state. However in modified RC4 this is not the case and the uniform distribution of j over $\mathrm{Z}_{2^\mathrm{n}}$ is not sufficient. The state update function also consists of the update of an element in S by integer addition modulo M given by the statement $S[(S[i] + S[j] \bmod M) \bmod N] = S[i] + S[j] \bmod M$. Since both $S[i]$ and $S[j]$ are uniformly distributed, the updated element in the state is also uniformly distributed. The internal state of $RC4(m, n)$ evolves in a uniformly distributed manner and therefore the output of the cipher is also uniformly distributed, i.e., all the elements from $\mathrm{Z}_{2^\mathrm{m}}$ occur with equal probability.

Next we analyze the randomness of a smaller segment of the RC4(n, m) keystream and compare it to original RC4. The output of the original RC4 can be considered as drawing cards randomly from a deck of 256 distinct cards with replacement. In this model if we draw $\sqrt{256}$ cards randomly then due to birthday paradox we can get a collision, i.e., at least one card is drawn twice with high probabilty. Similarly it can be shown that if we draw 256 cards randomly with replacement the number of distinct cards that will be selected is approximately $.632 \times 256 \approx 162$. Therefore we can expect to observe 162 distinct values in a 256 bytes segment of original RC4. The $RC4(n, m)$ case is slightly different. Here we can view the output as coming from a deck of 2^n cards, without replacement, which has been drawn from a larger deck of 2^m distinct cards. After each draw the smaller deck is replenished by taking a new card from larger deck with replacement. To study the behavior of this model we consider a reduced $RC4(n, m)$ example. We fix $m = 16$ and vary n to see the effect of the size of the smaller deck on the number of distinct cards chosen when 2^{16} cards are drawn. In other words this is the number of distinct 16-bit words observed in a 2^{16} words long $RC4(n, 16)$ keystream. The results of our experiment are shown in Figure 3. The graph shows the number of distinct 16-bit words observed as a percentage of the total words in the keystream, i.e., 2^{16}. For each value of n we ran the algorithm 10,000 times and the number of distinct words observed were averaged over 10,000 streams. The results show that when n is small the number of distinct elements observed is very high. However as n increases this number comes down and seems to be approaching towards 63.2 (original RC4 case) as n approaches m. However for $n = 16$ the percentage of distinct values observed was 62.1 which is slightly lower than original RC4. This difference is because of the fact that the array S in RC4(n, n) can have duplicate elements whereas the array S in original RC4 can not since it is a permutation. Therefore the number of collisions in RC4$(n, n)$is slightly higher than original RC4.

Another way of studying this behavior is to look at the frequency distribution graph of RC4 and RC4(n, m). For this purpose we used RC4$(n, 16)$ to generate 2^{24} words long keystreams. The frequency of each 16-bit word in the keystream was computed. If each word occurs equal number of times in the keystream, then all words occur with a frequency of $2^{24-16} = 256$. We plot the number of elements observed for each frequency in Figure 4. For each value of n, 100 different keystreams were generated. The graph suggests that if n is small compared to m, the output of RC4(n, m) can be distinguished form original RC4 by looking at the frequency distribution of the words in the smaller segments of the keystream. The curves for RC4(4,16) and RC4(7,16) can be easily distinguished from RC4 curve in Figure 4. However as n increases the frequency distribution resembles more with the original RC4 distribution. This behavior is demonstrated more clearly in Figure 5 where the curves

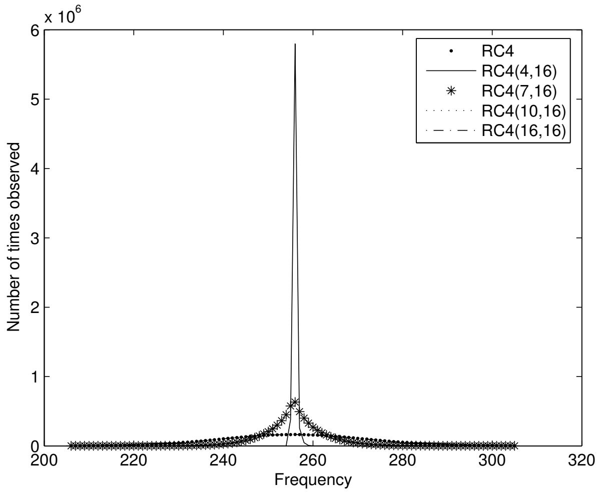

Fig. 4. Frequency distribution of RC4(n, m)

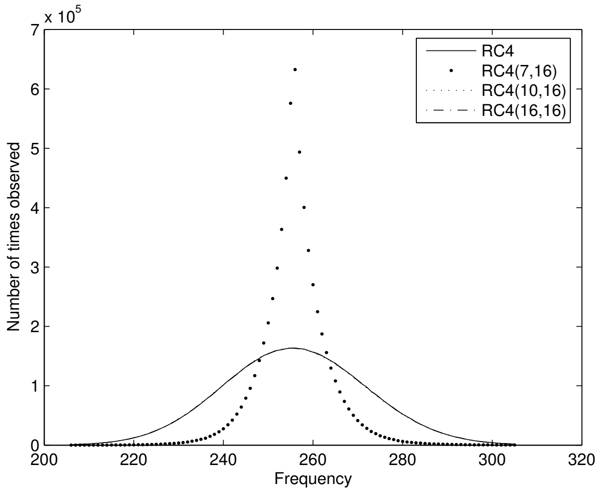

Fig. 5. Frequency distribution of RC4(n, m)

for RC4(10,16) and RC4(16,16) are very similar to original RC4. To increase the complexity of distinguishing RC4(n, m) from RC4, based on frequency distribution analysis, a large n can be used. However even if we use a smaller n, distinguishing RC4(n, m) output does not help in recovering the internal state of RC4(n, m). There are many existing pseudorandom generators which can be distinguished from a random process by such analysis. For example if a generator generates a complete cycle or permutation, i.e., 2^r r bit distinct words in a 2^r words long keystream, it can be distinguished from a truly random stream by observing $2^{r/2}$ words. AES used in counter mode, Klimov-Shamir pseudorandom sequence generaotr [8] and LFSRs with primitive feedback polynomials are examples of such generators.

5 Performance of RC4(n, m)

RC4$(n, 32)$ has been designed to exploit the 32-bit architecture of the current processors. If n is chosen such that the corresponding memory requirements are reasonable, RC4$(n, 32)$ can give much higher throughputs than the original 8-bit RC4. We implemented both 8-bit RC4 and RC4(8,32) on a PC and computed the ratio of the throughputs obtained from both. Our results show that RC4(8,32) is approximately 3.2 times faster than the original 8-bit RC4. This speedup will be much higher for RC4$(n, 64)$ on a 64-bit machine.

6 Conclusions and Future Work

In this paper we have proposed a new 32-bit RC4 like keystream generator. The proposed generator is 3.2 times faster than 8-bit RC4 on a 32-bit machine. The internal state of the proposed generator is much larger than the internal state of original RC4. Moreover given the current internal state of the generator it is not possible to retrieve the previous state in the absence of the keystream. The keystream produced by the proposed generator has good randomness properties and is uniformly distributed. However the frequency distribution of the words in a smaller segment of the keystream is different from original RC4 for smaller table sizes.

In future we would like to find the relation between the values of n and m which gives optimal frequency distribution and other randomness properties of the keystream. Like RC4 it would be interesting to investigate the existence of certain states that can not occur or have a negligible probability of occurance in the proposed RC4(n,m). Similarly it will be nice to see what is the probability of occurance of a particular state. Another problem is to compute the functional form of the RC4(n,m) state update function.

We are currently working on the security aspect of the 32/64 bit RC4 like stream cipher and trying to make it robust.

References

1. E. Biham, L. Granboulan, and P. Nguyen. Impossible and Differential Fault Analysis of RC4. *Fast Software Encryption 2005.*
2. H. Finney, An RC4 cycle that can't happen, *Post in sci.crypt, September 1994.*
3. S. Fluhrer, I. Mantin, and A. Shamir. Weaknesses in the Key Scheduling Algorithm of RC4. *SAC 2001 , vol. 2259 of LNCS, pp. 1-24, Springer-Verlag, 2001.*
4. S. Fluhrer and D. McGrew. Statistical Analysis of the Alleged RC4 Keystream Generator. *Fast Software Encryption 2000. vol. 1978 of LNCS, pp. 19-30, Springer-Verlag, 2000.*
5. J. Golić. "Linear Statistical Weakness of Alleged RC4 Keystream Generator," Eurocrypt '97 (W. Fumy, ed.), vol. 1233 of LNCS, pp. 226-238, Springer-Verlag, 1997.
6. A. Grosul and D. Wallach. A related key cryptanalysis of RC4. *Department of Computer Science, Rice University, Technical Report TR-00-358, June 2000.*
7. R. Jenkins. Isaac and RC4. *Published on the Internet at http://burtleburtle.net/bob/rand/isaac.html.*
8. A. Klimov and A. Shamir, A New Class of Invertible Mappings, *CHES 2002, Vol. 942 of LNCS, pp. 470-483, Springer-Verlag, 2002.*
9. L. Knudsen, W. Meier, B. Preneel, V. Rijmen, and S. Verdoolaege. Analysis Methods for (Alleged) RC4. *Asiacrypt '98, vol. 1514 of LNCS, pp. 327-341, Springer-Verlag, 1998.*
10. M. D. MacLaren and G Marsaglia. Uniform random number generation. *J. ACM, vol. 15, pp. 83–89, 1965.*
11. I. Mantin. Predicting and Distinguishing Attacks on RC4 Keystream Generator. *Eurocrypt Vol. 3494 of LNCS, pp. 491-506, Springer-Verlag, 2005.*
12. I. Mantin and A. Shamir. A Practical Attack on Broadcast RC4. Fast Software Encryption 2001. *Vol. 2355 of LNCS, pp. 152-164, Springer-Verlag, 2001.*
13. I. Mantin. The Security of the Stream Cipher RC4. *Master Thesis (2001) The Weizmann Institute of Science*
14. A. Maximov. Two Linear Distinguishing Attacks on VMPC and RC4A and Weakness of the RC4 Family of Stream Ciphers. *Fast Software Encryption 2005.*
15. I. Mironov. Not (So) Random Shuffle of RC4. *Crypto Vol. 2442 of LNCS, pp. 304-319, Springer-Verlag, 2002.*
16. S. Mister and S. Tavares. Cryptanalysis of RC4-like Ciphers. *SAC '98, vol. 1556 of LNCS, pp. 131-143, Springer-Verlag, 1999.*
17. S. Paul and B. Preneel. Analysis of Non-fortuitous Predictive States of the RC4 Keystream Generator. *Indocrypt 2003, vol. 2904 of LNCS, pp. 52-67, Springer-Verlag, 2003.*
18. S. Paul and B. Preneel. A New Weakness in the RC4 Keystremeam Generator and an Approach to Improve the Security of thr Cipher. Fast Software Encryption 2004. *Vol. 3017 of LNCS, pp. 245-259, Springer-Verlag, 2004.*

19. M. Pudovkina. Statistical Weaknesses in the Alleged RC4 keystream generator. *Cryptology ePrint Archive 2002-171, IACR, 2002.*
20. A. Roos. Class of weak keys in the RC4 stream cipher. *Post in sci.crypt, September 1995.*
21. A. Stubblefield, J. Ioannidis, and A. Rubin. Using the Fluhrer, Mantin and Shamir attack to break WEP. *Proceedings of the 2002 Network and Distributed Systems Security Symposium, pp. 17-22, 2002.*
22. Y. Tsunoo, T. Saito, H. Kubo, M. Shigeri, T. Suzaki, and T. Kawabata. The Most Efficient Distinguishing Attack on VMPC and RC4A. *SKEW 2005.*
23. B. Zoltak. VMPC One-Way Function and Stream Cipher. *Fast Software Encryption, vol. 3017 of LNCS, pp. 210-225, Springer-Verlag, 2004.*

Appendix A

Initial values for RC4(8,32) in hexadecimal format

a_{0}=AC1C1485	a_{1}=93FBFC7A	a_{2}=EF6E4DCF	a_{3}=5840EDE3
a_{4}=93A55E19	a_{5}=2A61EB6E	a_{6}=76DAACEC	a_{7}=F998D38B
a_{8}=7E857AB1	a_{9}=3AC3B17D	a_{10}=C5941CD9	a_{11}=19A50BE5
a_{12}=25C0DE69	a_{13}=079DF648	a_{14}=2E1AEB67	a_{15}=17CEC440
a_{16}=189EC261	a_{17}=CF544912	a_{18}=2C2E2485	a_{19}=0C28B1AC
a_{20}=6A2C2144	a_{21}=C6B4FD97	a_{22}=5A8DEC93	a_{23}=447391E7
a_{24}=FDE63D36	a_{25}=BE7F23AD	a_{26}=186F96CD	a_{27}=92B5E8AD
a_{28}=880D1003	a_{29}=29E97FE8	a_{30}=204FAD86	a_{31}=0E6DD8B3
a_{32}=D4805387	a_{33}=536B9CCC	a_{34}=63D6C749	a_{35}=38B83DCE
a_{36}=CC0A04A6	a_{37}=025B7563	a_{38}=D9E5E723	a_{39}=9AE27819
a_{40}=44848A51	a_{41}=E4294F27	a_{42}=3401AD9E	a_{43}=592F8A17
a_{44}=B042F066	a_{45}=7233B29F	a_{46}=92B9B132	a_{47}=62EB7323
a_{48}=1B97CA70	a_{49}=3089A026	a_{50}=A0BFAC39	a_{51}=05FCF2AA
a_{52}=2081C18D	a_{53}=711B88B3	a_{54}=D1C669AE	a_{55}=428B1206
a_{56}=B5B8DE0E	a_{57}=082B7A97	a_{58}=9165923C	a_{59}=207F2DF4
a_{60}=FFB20384	a_{61}=90F1AD8F	a_{62}=9B90D15F	a_{63}=B4E14AF6
a_{64}=9A5B3C5A	a_{65}=2FD7569F	a_{66}=E564DFB6	a_{67}=CD630423
a_{68}=E795C72C	a_{69}=FDBB5E8B	a_{70}=B45977AB	a_{71}=5A9B5D37
a_{72}=67BA087B	a_{73}=7D107FEF	a_{74}=D91B4819	a_{75}=0BB71EDA
a_{76}=30B4C371	a_{77}=C6DCC43F	a_{78}=289159A8	a_{79}=BCADC277
a_{80}=F16A54C6	a_{81}=4AE5923A	a_{82}=A7E87CE7	a_{83}=9F3FFB22
a_{84}=3292BA1A	a_{85}=61879694	a_{86}=098DC774	a_{87}=51424859
a_{88}=A62BAC2E	a_{89}=08094AE5	a_{90}=6C8327AE	a_{91}=330A3FA2
a_{92}=CD241DE5	a_{93}=6A72FF1A	a_{94}=787B8AC7	a_{95}=C1F17854
a_{96}=3603D1FB	a_{97}=30257CFF	a_{98}=4E8E3735	a_{99}=BE463311
a_{100}=2784F0D1	a_{101}=ADCEC09B	a_{102}=E89445C8	a_{103}=6B323ABC
a_{104}=718CC2BE	a_{105}=606B6072	a_{106}=DD665CF6	a_{107}=265288EA
a_{108}=B6E2F1BD	a_{109}=657597EE	a_{110}=A1911B1E	a_{111}=2411BA20
a_{112}=728B0382	a_{113}=C46E1179	a_{114}=B93ED4CD	a_{115}=23135C61
a_{116}=80BDF530	a_{117}=C564E110	a_{118}=82619049	a_{119}=41248BD5
a_{120}=FBC41845	a_{121}=B52F5554	a_{122}=736E67F0	a_{123}=648A8194
a_{124}=A1529479	a_{125}=B76A7D6F	a_{126}=091BB331	a_{127}=D6B90BD2
a_{128}=220C51A8	a_{129}=9E799F39	a_{130}=B706AF21	a_{131}=F7A6FB04
a_{132}=0C5308FA	a_{133}=E220F287	a_{134}=96317BFE	a_{135}=3E5308BD
a_{136}=2C9ECCE4	a_{137}=2571548D	a_{138}=AAA99C21	a_{139}=E455C945
a_{140}=94DB51BF	a_{141}=F01BD3B2	a_{142}=EDE57029	a_{143}=DD55A344
a_{144}=B7B1B6FF	a_{145}=89674C52	a_{146}=8F973667	a_{147}=19DFE7F3

a_{148}=2C9F9C55	a_{149}=105F613A	a_{150}=FDFF8DD2	a_{151}=23FF74AF
a_{152}=1D48D23F	a_{153}=48F19AE2	a_{154}=AF0AA311	a_{155}=8D7692E7
a_{156}=6D68E4D7	a_{157}=B81571B2	a_{158}=DEBE6453	a_{159}=D398EDB5
a_{160}=164B3FDD	a_{161}=04398FDC	a_{162}=79C46A18	a_{163}=9E07CDA9
a_{164}=57C0D84B	a_{165}=AE7F8F86	a_{166}=1F0E8114	a_{167}=84EB028D
a_{168}=9ED574FD	a_{169}=A594FB5D	a_{170}=E8E7F7C9	a_{171}=BD562227
a_{172}=5BEAFE2E	a_{173}=482A62CE	a_{174}=3FFA129F	a_{175}=67F60747
a_{176}=19CF8EDC	a_{177}=C156B571	a_{178}=23076173	a_{179}=4C48ACA6
a_{180}=9716174C	a_{181}=F1655069	a_{182}=7B63C7F6	a_{183}=49DFDC5B
a_{184}=B4BCCD49	a_{185}=3BE45CCB	a_{186}=697C6A5C	a_{187}=C58154F7
a_{188}=62D3C4D4	a_{189}=EEF06449	a_{190}=C9F0F522	a_{191}=DA294682
a_{192}=0CD55B58	a_{193}=29095DB3	a_{194}=2EFCC7F8	a_{195}=1E3059ED
a_{196}=0AC768AA	a_{197}=61F86F0B	a_{198}=2348010E	a_{199}=2E279EC3
a_{200}=13584D54	a_{201}=FA8C373E	a_{202}=8067A05B	a_{203}=6B28150D
a_{204}=DB9C4FFD	a_{205}=8D416CC9	a_{206}=077DFA94	a_{207}=83AFB0CF
a_{208}=311BEEF4	a_{209}=D601FDC4	a_{210}=AEE358ED	a_{211}=C332FD86
a_{212}=613D0F1B	a_{213}=5AA32732	a_{214}=3DCA86D7	a_{215}=4312DC90
a_{216}=A96639D0	a_{217}=47DD1B8D	a_{218}=2A2A905E	a_{219}=67A1E863
a_{220}=F1402D83	a_{221}=1BB91FD9	a_{222}=DA8001D2	a_{223}=4C4A7FFD
a_{224}=3D00A615	a_{225}=17CF5BCB	a_{226}=AD2267DC	a_{227}=1F2592FD
a_{228}=64A5033B	a_{229}=8DD7ABCB	a_{230}=843BE330	a_{231}=6D749833
a_{232}=13892876	a_{233}=7B4D29CF	a_{234}=9BAD0682	a_{235}=E6E362B5
a_{236}=25BFDF62	a_{237}=213B5A66	a_{238}=F65B605D	a_{239}=3280A5C0
a_{240}=D80A7565	a_{241}=706B1576	a_{242}=A8C05B12	a_{243}=C8C36AE2
a_{244}=6D716E08	a_{245}=7D869E68	a_{246}=C5B255DC	a_{247}=E325CD41
a_{248}=25C98682	a_{249}=64F298E3	a_{250}=BBB923E5	a_{251}=0A0AAE06
a_{252}=CB9D5A3B	a_{253}=3E6CAB45	a_{254}=627860DA	a_{255}=C919422D