Des. Codes Cryptogr. (2008) 48:269–286
DOI 10.1007/s10623-008-9206-6

Attacks on the RC4 stream cipher

Andreas Klein

Received: 13 March 2007 / Revised: 5 March 2008 / Accepted: 24 March 2008 /
Published online: 16 April 2008

Abstract In this article we present some weaknesses in the RC4 cipher and their cryptographic applications. Especially we improve the attack described by Fluhrer, Mantin, Shamir (In: Selected Areas in Cryptography, 2001) in such a way, that it will work, if the weak keys described in that paper are avoided. A further attack will work even if the first 256 Byte of the output remain unused. Finally we show that variants of the RC4 algorithm like NGG and RC4A are also vulnerable by these techniques.

Keywords Cryptanalysis · Stream cipher · RC4

AMS Classifications 68P25 · 94A60

1 Introduction

RC4 is probably the most popular stream cipher that is not based on a feedback shift register. It was developed in 1987 by Ron Rivest, but the algorithm was kept secret until 1994. The first publication of the algorithm was an anonymous posting at the mailing list cipherpunks. After this publication several weaknesses were discovered in the pseudo random sequence generated by RC4.

The most successful attack so far on RC4 was presented by Fluhrer et al. [2] (FMS-Attack) and uses a weakness in the key scheduling phase. The main idea is that RC4 is commonly used with keys of the form

$$\text{session key} = \text{initialization vector} \| \text{main key}.$$

Communicated by P. Wild.

A. Klein (✉)
Department of Pure Mathematics and Computer Algebra, Ghent University,
Krijgslaan 281-S22, 9000 Ghent, Belgium
e-mail: klein@cage.ugent.be

If the initialization vectors are suitably chosen the first byte of the pseudo random sequence is with high probability ($\approx \frac{1}{e}$) identical to a predefined byte of the main key.

In this paper we want to present a new attack on RC4. The work is organized as follows: In Sect. 2 we describe the RC4 algorithm and give an brief overview of previous analysis.

Then (Sect. 3) we prove a correlation between the publicly known output and the internal state. This previously unknown weakness will be the basis for our attacks.

Our first attack (Sect. 4) will examine the first bytes of the pseudo random sequence, like the FMS-Attack, but requires no special from of the initialization vector. If the attacker can not manipulate the initialization vector directly he is forced by the FMS-Attack to wait until a suitable number of weak initialization vectors occurred by chance. Approximately this will take between 1,000,000 and 4,000,000 sessions. The new attack will need only 25,000 sessions.

The second attack (Sect. 5) uses, like the FMS-Attack, chosen initialization vectors. The difference is that it does not need the first byte of the pseudo random sequence. It is still successful if the first 256 bytes of the RC4 pseudo random sequence are not observable. This is important in practice since many authors (see for example [10]) suggest not to use the first bytes of the RC4 pseudo random sequence to avoid known vulnerabilities.

In the Sect. 6 we describe an extension of our attacks that reduces the number of required sessions but increases the computation time. We will also discuss how the known ciphertext attacks from the previous sections can be transformed into a cipher text only attacks.

Finally (Sect. 7) we look at variants of the RC4 algorithm and discuss how to adapt the attacks to such variants.

2 The RC4 algorithm

In this part we will describe the RC4 algorithm and give an overview of previous work.

2.1 Description of RC4

RC4 consists of two parts. The key scheduling phase will generate the initial permutation from a (random) key of length l bytes. Typically l will be in the range between 5 and 64. The maximal key length is $l = 256$. The main part of the algorithm is a pseudo random generator that produces one byte output in each step. The encryption will be an XOR of the pseudo random sequence with the message, as usual for stream ciphers.

For the analysis of RC4 it is convenient to replace the original algorithm that works on bytes ($\mathbb{Z}/256\mathbb{Z}$) by a generalization that works on $\mathbb{Z}/n\mathbb{Z}$ for some $n \in \mathbb{N}$. For $n = 256$ we obtain the original algorithm.

We will call n successive outputs of the RC4 pseudo random generator a round, i.e. the first round will produce the output bytes 1 to n, the second round the bytes $n + 1$ to $2n$ and so on. If an attack only uses bytes from the i-th round or later we will call it an i-th round attack. All previously known attacks were 1-round attacks. In this paper we will present the first practical 2-round attack.

2.2 Correlations in the RC4 pseudo random generator

It is known that the RC4 pseudo random sequence differs in some aspects from a true random sequence.

Algorithm 1 RC4 key scheduling

```
{initialization}
for $i$ from 0 to $n - 1$ do
    $S[i] := i$
end for
$j := 0$
{generate a random permutation}
for $i$ from 0 to $n - 1$ do
    $j := (j + S[i] + K[i \mod l]) \mod n$
    Swap $S[i]$ and $S[j]$
end for
```

Algorithm 2 RC4 pseudo random generator

```
{initialization}
$i := 0$
$j := 0$
{generate pseudo random sequence}
loop
    $i := (i + 1) \mod n$
    $j := (j + S[i]) \mod n$
    Swap $S[i]$ and $S[j]$
    $k := (S[i] + S[j]) \mod n$
    print $S[k]$
end loop
```

The first published difference is due to Golić [4,5]. He proves that the sum of the last bits at time step t and $t + 2$ is correlated to 1. The correlation coefficient is $\approx 15 \cdot 2^{-24}$. He concludes that 2^{40} bytes of the RC4 pseudo random sequence are distinguishable from a true random sequence.

Fluhrer and McGrew [3] investigate the joint probability of two consecutive output bytes. They prove that the joint distribution of two successive bytes differers significantly from the uniform distribution. For example they prove that for $i \neq 1, n - 1$ the probability of two successive 0 as output is $\approx n^{-2}(1 + n^{-1})$ instead of n^{-2}. From this differences they conclude that even 2^{30} bytes of the RC4 pseudo random sequence are sufficient to distinguish it from a true pseudo random sequence. The most recent continuation of their work is due to Mantin [8], who lowers this bound to $\approx 2^{26}$ bytes.

2.3 Weaknesses in the key scheduling phase

The key scheduling phase of RC4 has also known weaknesses. In the ideal case that the key consists of n independent, identically and uniformly distributed elements of $\mathbb{Z}/n\mathbb{Z}$ the key scheduling phase can produce n^n results all of equal probability. But $n!$ is not a divisor of n^n and therefore the distribution of the initial permutation must differ from the uniform distribution. A detailed study of this differences is due to Mironov [10]. He shows among other things, that the identity is the most likely initial permutation and the cycle $(1, 2, \ldots, n-1, 0)$ is the most unlikely initial permutation. Mironov suggest not to use the first $12 \cdot 256$ bytes of the RC4 pseudo random sequence to avoid a possible exploitation of this weakness. As a minimal precautionary measure he suggests not to use the first $2 \cdot 256$ to $3 \cdot 256$ Bytes of the RC4 pseudo random sequence.

A weakness detected early was that the first byte of the pseudo random sequence is not very random (see [9,11]). The strongest know attack of this kind is due to Fluhrer et al. [2].

Their attack assumes, that the initialization vector precedes the main key and that the two first bytes are of the form $(b, n-1)$, where b is the byte of the main key that shall be reconstructed. If an attacker is not able to manipulate the initialization vector (which is the usual case) he has to wait until the initialization vector takes the desired form by chance, i.e. he can use approximately only 1 out of n^2 sessions. The authors show that their attack can be applied successfully against the WEP-Protocol. The weakness can be used even in situations in which the initialization vector follows the main key, but it is more complicated in this situation.

3 A correlation in the RC4 pseudo random generator

In this section we prove a strong correlation between the observable values i, $S[k]$ and the internal states j, $S[j]$ and $S[i]$. This correlation will be the base for our attacks. (The referee pointed me to [6] where the correlation of Theorem 1 previously mentioned but not proved. The proof of Theorem 1 gives us an deeper understanding and allows the transfer to RC4 variants, see Sect. 7.)

In this section we look only on the steps 9 and 10 (computation of k and output of $S[k]$) of the RC4 pseudo random generator (Algorithm 2).

Theorem 1 *Assume that the internal states are uniformly distributed. Then for a fixed public pointer i we have*:

$$P(S[j] + S[k] \equiv i \mod n) = \frac{2}{n} \tag{1}$$

$$P(S[j] + S[k] \equiv i \mod n \mid S[j] = x) = \frac{2}{n} \quad \textit{for all } x \in \{0, \dots, n-1\} \tag{2}$$

For $c \not\equiv i \mod n$ we have:

$$P(S[j] + S[k] \equiv c \mod n) = \frac{n-2}{n(n-1)} \tag{3}$$

$$P(S[j] + S[k] \equiv c \mod n \mid S[j] = x) = \frac{n-2}{n(n-1)} \quad \textit{for all } x \in \{0, \dots, n-1\} \tag{4}$$

Proof First note that (1) follows from (2) and (3) follows from (4).

Since we assume that the internal states are uniformly distributed we have Laplace probabilities. There are $n!$ different internal states with $S[j] = x$. So main problem is to count the number of permutations which satisfies the congruence.

To prove (2) we count all internal states with $S[j] + S[k] \equiv i \mod n$ and $S[j] = x$. First we use $k \equiv S[j] + S[i] \mod n$ (line 9 of Algorithm 2) to write $S[j] + S[k] \equiv i \mod n$ as $k + S[k] \equiv i + S[i] \mod n$.

We have to distinguish two cases

1. $i = k$.
   Then $S[i] = S[k]$ and the equivalence is satisfied trivially. In this case $S[i] = S[k] \equiv i - S[j] \equiv i - x \mod n$ and there are $(n-1)!$ possibilities to choose the remaining $n-1$ entries of the S-box.
2. $i \neq k$.
   Then we have to set $S[k] \equiv i - x \mod n$ and $S[i] = k + S[k] - i \mod n$. We may still choose k ($n-1$ possibilities) and the remaining $n-2$ entries of the S-box ($(n-2)!$ possibilities).

Hence there are all together $(n-1)!+(n-1)(n-2)! = 2(n-1)!$ possibilities in which we have $S[j]+S[k] \equiv i \mod n$ and $S[j] = x$ but there exists $n!$ possible internal states where $S[j] = x$ which proves (2).

The prove of (4) is quite similar.

Here we have to distinguish three cases:

1. $i = k$.
   In this case $S[i] = S[k]$ and $k + S[k] \equiv c + S[i] \mod n$ can not be satisfied, since $c \neq i = k$.
2. $c = k$.
   In this case $k + S[k] \equiv c + S[i] \mod n$ implies $S[k] \equiv S[i] \mod n$. But this is impossible, since $k \neq i$ and therefore $S[i] \neq S[k]$.
3. $i \neq k$ and $c \neq k$
   In this case we have to set $S[k] \equiv i - x \mod n$ and $S[i] = k + S[k] - c \mod n$. We may still choose k ($n-2$ possibilities) and the remaining $n-2$ entries of the S-box ($(n-2)!$ possibilities).

Therefore only the third case yields a contribution to the number of possible internal states. Thus

$$P(S[j]+S[k] \equiv c \mod n \mid S[j] = x) = \frac{(n-2)(n-2)!}{n!} = \frac{n-2}{n(n-1)} .$$

This proves the theorem. □

Of course we know that in a regular RC4 instantisation the internal states are not uniformly distributed, but the difference between the real distribution and the uniform distribution is only small, in other words, Theorem 1 is a good approximation.

4 Attack in the first round

Now we want to apply Theorem 1 in an 1-round attack. First we describe the basic version of the attack where the session keys have the form

main key||initialization vector

and the attack determines the sum of the first two key bytes. Afterwards we will show how to modify the attack to recover other key bytes or how to deal with a initialization vector preceding the main key.

4.1 The basic version of the attack

We investigate the first two steps of the key scheduling phase. In the first step j takes the value $K[0]$ and $S[0]$ is swapped with $S[K[0]]$. In the second step j is increased by $S[1]+K[1]$ and the entry of $S[j]$ is move to $S[1]$.

Since we start with $S[j] = j$ for all j, we deduce that after the second step of the key schedule the value of $S[1]$ is $t = K[0]+K[1]+1$, except the following cases:

(a) $K[0] = 1$, $K[1] = 0$. In this case we have $t = 0$.
(b) $K[0] = 1$, $K[1] \neq 0, n-1$. In this case we have $t = K[0]+K[1]$, since in the first step $S[1]$ is set to 0 and in the second step j is increased only by $K[1]$.

(c) $K[0] \neq 1$, $K[1] = n - 1$. In this case j is not changed in the second step and thus $S[1]$ gets the value $t = 0$.
(d) $K[0] \neq 1$ and $K[0] + K[1] = n - 1$. In this cases $j = 0$ after the second step and therefore $S[1]$ gets the value $t = K[1]$ (which is the value of $S[0]$ after the first step.)

The many special cases may look confusing, but the only thing we need to know is the following: For fixed $K[0]$ the value t of $S[1]$ after the second step is an easily computable function of $K[1]$.

In the remaining steps of the key scheduling phase $S[1]$ will never be changed except if j takes the value 1. The probability that this happens in one step is $1 - \frac{1}{n}$. If the session key has length n and if all key bytes are independent, we may conclude that $S[1]$ will not be changed after the second step with probability $(1 - \frac{1}{n})^{n-2} \approx \frac{1}{e}$. Of course, for shorter keys the independence assumption is false, this leads to some minor problems. We address these issues later in Subsect. 4.3. However, we may take $\frac{1}{e}$ as a good approximation for the probability that $S[1]$ is not changed after the second step of the key scheduling phase.

At this point we have achieved the following: We know that the value of $S[1]$ at the start of the RC4 pseudo random generator will be t, which depends only on $K[0]$ and $K[1]$, with a high probability ($\approx \frac{1}{e}$).

Now we want to use the correlation proven in Theorem 1 to obtain t from observation of the RC4 pseudo random sequence. To that goal we look at the generation of the first pseudo random byte. First i is set to 1, and than $S[1]$ and $S[j]$ are swapped. Now $S[j]$ contains the interesting value t. Furthermore we can observe the output $S[k]$ of the RC4 pseudo random generator. But by Theorem 1 we know that $S[j] \equiv 1 - S[k] \mod n$ with probability $\frac{2}{n}$. (At this point we mention again, that in the real RC4 instance neither the S-box nor the secret counter j are truly random. Therefore Theorem 1 gives only an approximation, but the small error has no affect to the attack.)

All together we have

$$P(t \equiv 1 - S[k] \mod n) \approx \frac{1}{e} \cdot \frac{2}{n} + \left(1 - \frac{1}{e}\right) \cdot \frac{n-2}{n(n-1)} \approx \frac{1.36}{n}. \tag{5}$$

(The second summand express the probability that $S[1] \neq t$ at the end of the key scheduling phase, but $t \equiv 1 - S[k] \mod n$ anyway.)

Our attack against RC4 takes to following form:

For a number of different initialization vectors we observe the first byte x_i of the RC4 pseudo random generator and compute $t_i = 1 - x_i$. The fraction of the t_i that have the correct value t is about $\frac{1.36}{n}$. All other values will have a relative frequency less than $\frac{1}{n}$. If the number of sessions is large enough we can be sure that the right value is the most frequent value.

We use methods from information theory to obtain an estimation for the number of necessary sessions. Without any observation the uncertainty about t is $\log_2(n)$. The redundancy of a distribution in which one value has probability $p = \frac{1.36}{n}$ and the remaining $n - 1$ values have probability $q = \frac{1-p}{n-1}$ is

$$r = \log_2(n) + p \log_2(p) + (n-1) q \log_2(q).$$

In each step the amount of obtained information is approximately r, thus we need about $\log_2(n)/r$ steps to obtain enough information to estimate r. Simplifying this formula (with a computer algebra system) we find that $\approx 17 n \ln(n)$ sessions are sufficient to reconstruct t. (For $n = 256$ this are only 25,000 sessions, where, in comparison, the FMS-attack needs approximately 1,000,000 sessions.) Experiments with different values of n indicate that the heuristic bound described above is a very accurate estimator for the number of necessary sessions.

In most applications we will not use precise bounds. A typical scenario will look like the following. After 15,000 sessions we will start a first attempt to reconstruct the key. Most likely it will fail. Then we will wait an other 5,000 sessions and start a next attempt. In most cases we will succeed after at most 30,000 sessions. The way how often we start an new attempt depends of course on the circumstance manly how difficult it is to observe a session. In some application we can use techniques like ARP-spoofing to generate very fast extra sessions. In such an application we will generate 35,000 or even more session before starting the first attempt. On the other extreme the are cases were it is near to impossible to observe even 20,000 session, in such cases we can use the technique described in Sect. 6.1 to make our attack still successful.

In the cases the heuristic based one information theory is not precise enough I recommend a sequential test. (See for example [13] for a introduction to sequential tests.) In our case we can do the following:

We suppose that all possible values of t have the same a priori probability (Bayes method). This is always the case if the key is uniformly distributed. The Bayes method is therefore very natural for our problem, in contrast to many other statistical problems. The a posterior probabilities can be computed from the absolute frequencies f_i (number of sessions where $t = i$) by the following formula:

$$P_i = P(t = i \mid \text{the absolute frequencies are } f_i) = \frac{\prod_{j=1}^{n} p_{i,j}^{f_j}}{\sum_{k=1}^{n} \prod_{j=1}^{n} p_{k,j}^{f_j}} \tag{6}$$

(Here we denote by $p_{i,j} = p = \frac{1.36}{n}$ for $i = j$ and $p_{i,j} = q = \frac{1-p}{n-1}$ for $i \neq j$.)

If there exists an i such that $P_i \geq 1 - \alpha$ for a prescribed error probability α, we stop and decide that $t = i$. Otherwise we observe a new session and repeat the test. This test guarantees an error probability less that α and uses a minimal number of sessions.

Since in cryptography we can check the reconstructed key simply by testing it, I think precise error bounds are not necessary and the sequential test is not worth the additional effort, but the method with the a posterior probabilities allows us to make a tradeoff between the computational effort of the attack and the number of observed sessions (see Sect. 6.1).

4.2 Attack on other key bytes

So far we can only reconstruct the first two key bytes ($K[0]$ has to be guessed and $K[1]$ is found by the test). Now we want to reconstruct $K[2]$. We note that in the third step of the key scheduling phase $S[2]$ is set to the value $t = f(K[0], K[1], K[2])$. The function f is efficiently computable and for fixed t, $K[0]$ and $K[1]$ there exists a unique $K[2]$ with $t = f(K[0], K[1], K[2])$. With probability $\approx \frac{1}{e}$ the value of $S[2]$ is not changed in the remaining $n - 3$ steps of the key scheduling phase and in the first step of the pseudo random generator. In the second step of the pseudo random generator $S[2]$ is swapped with $S[j]$, i.e. $S[j]$ is set to t. Now we apply Theorem 1 to estimate t from the second output byte generated by the RC4 pseudo random generator. From t we can compute $K[2]$.

The test needed for this has the same complexity as the test described in the previous subsection. Thus we may use the same $17n \ln(n)$ sessions.

The key bytes $K[3]$, $K[4]$ and so on can be obtained by the same way observing the third, forth and so on byte of the RC4 pseudo random sequence. The effort needed to break the RC4 algorithm is therefore reduced from $O(n^k)$ (brute force testing of all possible keys) to $O((k-1)n^2\ln(n))$ (guess the first key byte n possibilities and apply the test (afford $O(n\ln(n)))$ to obtain the remaining $k-1$ key bytes).

Note that a wrong guess of the first key byte is easily detected by the simple fact that the attack fails to produce the right key.

4.3 Implementation problems

In Subsect. 4.1 we have estimated that the probability that $S[1]$ is not changed after the first step of the key scheduling is approximately $\frac{1}{e}$. This is correct if the session key consists of n independent bytes. In a real attack scenario the session key is shorter than n bytes, i.e. the probability $\frac{1}{e}$ is not quite correct. In this subsection we want to discuss the problems that arises from this fact.

The problem is not that the session key is shorter than n and therefore the different values of j during the key scheduling are not independent. If the length session key is not a to small fraction of n (a session key of length 8 would be enough for $n = 256$) the correlation between the different values of j is small enough. $\frac{1}{e}$ is still a good approximation for the probability, that j does not take the value 1.

The problem is the following: Since the session key has the form main key‖initialization vector, the first l steps of the key scheduling phase will be the same in all sessions (l is the length of the main key). If we are unlucky, during the first l steps the pointer j will be set to 1. Hence the assumption $S[1] = t = f(K_0, K_1)$ fails and the basic variant of our attack will fail to reconstruct the key. Therefore we have to find a way to deal with such ugly main keys.

Lets have a closer look at the attack. The tests described in the previous subsections reconstruct the values $S[1], S[2], \ldots, S[l-1]$ of the S-box after the first l steps of the key scheduling phase. (In the $l+1$ step of the key scheduling phase we process the first byte of the initialization vector.) Our task is now to reconstruct the main key from this information.

Since there are only $n(n-1)\cdot\cdots\cdot(n-l+2)$ possible values of $(S[1], \ldots, S[l-1])$ but n^l possible keys, we expect $\frac{n^l}{n!/(n-l+1)!}$ keys resulting in the same values $S[1], \ldots, S[l-1]$. This means that for $n = 265$ and $l = 16$ we must do a search over ≈ 389 keys and for $n = 256$ and $l = 32$ we must search in a set of ≈ 1700 keys. This effort is of course negligible.

We describe now how to find the set of admissible keys.

We start by guessing $K[0]$. Now we can simulate the first step of the key scheduling phase. We call the values of $S[1], \ldots, S[l-1]$ after this first step the *old values*, every other number we call a *new value*.

We call a swap between $S[i]$ and $S[j]$ with $i, j \in \{1, \ldots, l-1\}$ a *problematic swap*. A problematic swap is *ugly* if $j < i$. The equalities $S[1] = f_1(K[0], K[1])$, $S[2] = f_2(K[0], K[1], K[2]), \ldots, S[l-1] = f_2(K[0], \ldots, K[l-1])$ as described in the previous subsection work only, if there are no ugly swaps occurred during the first l steps of the key scheduling phase. In this case we can compute the key by guessing $K[0]$ and inverting the known functions $f_1, \ldots, f_l$. (For $l = 16$ and $n = 256$ the fraction of nice keys for which this works is about 0.62.)

If we know exactly which ugly swaps occurs we can adapt our reconstruction. Thus we have to identify the ugly swaps. We do a little bit more—we identify the problematic swaps.

Since a problematic swap does not change the set $\{S[1], \ldots, S[l-1]\}$ it can not replace an old value by a new one. We search the old values among the know values of $S[1], \ldots, S[l-1]$ after the lth step of the key scheduling phase. The expected number of old values is $\frac{(l-1)^2}{n}$. (For $l = 16$ and $n = 265$ this is 0.87 for $l = 32$ and $n = 256$ this is 3.75.) Each old value indicates a possible problematic swap.

But the problematic steps must move the old values to their proper places. From this information we can reconstruct the possible combinations of problematic steps.

Example Let $l = 5$ and we guess $K[0] = 0$. The old values are 1, 2, 3 and 4. After the first 5 steps of the key scheduling phase we have $S[1] = 10$, $S[2] = 4$, $S[3] = 5$ and $S[4] = 17$. There is only one old value and therefore at most one possible problematic swap. Since we start with $S[4] = 4$ the problematic swap must swap $S[2]$ and $S[4]$ and it must occur either at time step $i = 2$ or $i = 4$.

If the number of old values in $S[1], \dots, S[l-1]$ is small (and this is the case for all realistic combinations of l and n), the number of possible problematic swaps is also small.

We compute for each combination of problematic swaps the corresponding key and test if that key is correct.

The computational effort of this algorithm is of course larger than the simple inversion of $S[1] = f_1(K[0], K[1])$, $S[2] = f_2(K[0], K[1], K[2]), \dots, S[l-1] = f_2(K[0], \dots, K[l-1])$ but it is still feasible.

4.4 Initialization vector precedes the main key

Now suppose that the initialization vector precedes the main key. If b is the length of the initialization vector, we know the bytes $K[0], \dots, K[b-1]$ of the session key. An attacker is therefore able to compute the first b steps of the key scheduling phase. He can therefore express the value t of $S[b]$ after the $(b+1)$th step as a function of the first unknown key byte $K[b]$. (This function depends of course also on the initialisation vector. So we must update it for every session.) With probability $\frac{1}{e}$ this value is not changed in the remaining steps of the key scheduling phase and the first $b-1$ steps of the pseudo random generator. We can use Theorem 1 to estimate t and therefore $K[b]$. This case is even simpler than the case in which the main key precedes the initialization vector since here we need not to guess the first key byte. Furthermore, the initialization vector at the beginning adds enough randomness to the system to avoid the problem discussed in Subsect. 4.3.

At this point we want to mention that it makes no difference if the initialization vector is a random number or a counter. For the purpose of our attack counters are an good approximation to random numbers.

5 Attack in the second round

Our attack in the second round uses a concept of weak initialization vectors, similar to the FMS-attack. But the FMS-attack must prescribe the first two bytes, our attack needs only that the sum of the first two bytes have a prescribed value. Thus we can use every nth session instead of every n^2th session.

5.1 The basic variant of the attack

For the basic variant of the attack we assume that the initialization vector precedes the main key. The first byte of the main key gets the number b.

We assume that in the first b steps of the key scheduling phase $S[1]$ is set to b. For random initialization vectors this will happen with probability $\frac{1}{n}$. Since the initialization vector is know to the attacker he can compute the first b steps of the key scheduling phase and check if $S[1] = b$. We will only analyse sessions that satisfies this assumption.

The next step of the key scheduling sets $S[b]$ to a value $f(K[b])$. The attacker knows the function f and he wants to find the unknown key value $K[b]$. In the remaining steps of the key scheduling phase each of the values of $S[1] = b$ and $S[k] = f(K[b])$ will not be changed with a high probability of $\frac{1}{e}$.

The first step of the pseudo random generator will set j to $S[1] = b$ and interchange $S[1]$ with $S[j] = S[b] = f(K[b])$. Thus we know that after the first step of the pseudo random generator $S[1] = f(K[b])$ with probability $\frac{1}{e^2}$. In the next $n - 1$ step of the pseudo random generator the pointer j will be different from 1 with probability $\approx \frac{1}{e}$, i.e. with at the beginning of the second round $S[1]$ will have the value $f(K[b])$ with probability $\approx \frac{1}{e^3}$.

Now we analyze the output of the first step of the second round. The pointer i will be equal 1 and j has a value that is not known to us. After we have swapped $S[i]$ and $S[j]$, we know that $S[j] = f(K[b])$ with probability $\frac{1}{e^3}$. Theorem 1 says that $S[j] = 1 - S[k]$ with probability $\frac{2}{n}$. Similar to our 1-round attack we obtain

$$P(f(K[b]) = 1 - S[k]) \approx \frac{1}{e^3} \cdot \frac{2}{n} + \left(1 - \frac{1}{e^3}\right) \cdot \frac{n-2}{n(n-1)} \approx \frac{1.05}{n} \tag{7}$$

Difference from the uniform distribution is smaller than the one found for the 1-round attack but still significant.

The we obtain a estimator for $f(K[b])$ by observing $S[k]$. Inverting the known function f we obtain $K[b]$. Since the success probability is smaller than the one used for the 1-round attack we have to study more sessions (approximately $813n \ln(n)$ session that satisfies $S[1] = b$).

After we have obtained $K[b]$ we can tread it as part of the initialization vector and apply the algorithm described above to obtain $K[b + 1]$, $K[b + 2]$ and so on.

Since we can not choose the initialization vectors we have to wait long enough to get the right number of initialization vectors (with $S[1] = b$, $S[1] = b + 1$, etc) for our attacks. For random initialization vectors the estimated number of needed session is therefore $n(813n \ln(n))$.

We want now to discuss the case in which the initialization vector is not random but generated by a counter.

5.1.1 Big endian counter

If the initialization vector is generated by a big endian counter the assumption $S[1] = b$ is satisfied in every nth session. Thus in this case we need $n(813n \ln(n))$ sessions for a successful attack.

5.1.2 Little endian counter

If the initialization vector is generated by a little endian counter the things are a bit different. If $S[1] = b$ is satisfied once it will stay b for many sessions, but it will take a long time until the assumption $S[1] = b$ is satisfied the first time.

If the initialization is b bytes long and we want attack the first byte $K[b]$ of the main key, we have to wait until $K[0] + K[1] = b - 1$ to obtain $S[1] = b$. This happens first for $K[0] = 0$ and $K[1] = b - 1$. The counter value is at this time $(b - 1)n^{b-2}$. Thus we have to wait such long before we can start our attack. For typical values like $n = 256$ and $b = 16$ this will be unacceptable.

The following modification of the 2-round attack will help in this case:

Instead of the investigation of initialization vectors that will result in $S[1] = f(K[b])$ after the first step of the pseudo random generator, we will look at initialization vectors that will result in $S[j] = f(K[b])$ after the jth step of the pseudo random generator. This is possible but now we must assume that the first j bytes of the S-box are not changed during the last $n - j$ steps of the key scheduling phase. The probability for this is $(1 - \frac{j}{n})^{n-j} \approx \frac{1}{e^j}$. This means that the success probability will be smaller, i.e. we have to observe more sessions. But if we choose j near at b we can use every nth session for the attack. We certain values of b, j and n this can be faster than the basic variant of the 2-round attack.

5.2 Initialization vector follows the main key

If the initialization vector follows the main key the attack becomes more difficult. The key length l should be small in comparison with n (like $l = 16$ and $n = 256$). In the first l steps of the key scheduling phase j will be set to a value j_l which depends only from the main key. The permutation S will be similar to the identity permutation.

If the key scheduling would start with $i = l$ and $j = j_l$ (S initialized with the identity), we would be able to apply the attack described in Sect. 5.1. Only minor modifications are necessary for the new starting values of i and j.

We do not know that value j_l, i.e. we have to guess it. Since after l steps the permutation S differers form the identity we must further assume that the difference do not matter, i.e. the success probability for the attack becomes smaller.

6 Further improvements

In this section we want to present two further enhancements of the attacks.

6.1 Reducing the number of sessions

In Sects. 4 and 5 we assumed that we can observe enough sessions. In this case the tests described in these sections can reconstruct the right key almost certainly. But if we cannot observe a sufficient number of sesions, this is not longer true. In that case we have to combine the results of the statistical analysis with a search in the complete key space.

For example we assume that we want to use the 1-round attack of Sect. 4 to attack RC4 with a 128-Bit key. But instead of 25,000 sessions we can observe only 13,000 sessions. Since the number of sessions is to small we can not longer assume that the test will find the correct key bytes. Bur our estimation of the information obtained per session suggest that the uncertainly about the key should drop from 128 to about 64. This means that we should be able to find the right key with only $\approx 2^{64}$ operations. To do this we proceed as follows.

The one 1-round attacks should find a value t from which can compute a key byte. We observe the absolute frequencies

$$t_i = \text{number of sessions with suggest } t = i.$$

The simple 1-round attack which analysis 25,000 sessions, searches the maximal value t_i and assumes that $t = i$ with very high probability. For less than 25,000 session we can not assume that the right t_i is the maximal with a high enough probability. But we can compute the a posterior probabilities for t with Eq. 6. But this time we stop before we have observed enough sessions to get an a posterior probability $\geq 1 - \alpha$.

The a posterior probabilities for t implies a posterior probabilities for the key K. This means that the entropy of the key becomes smaller. For 13,000 sessions the a posterior entropy will be ≈ 64 instead of 128.

We now begin testing all 2^{128} possible keys, but we start with the keys with the highest a posterior probability. If we do this, we may expected that we find the right key after only 2^{64} tests.

The tradeoff described above can applied to every attack described in this article. Depending on the number of observed sessions the computational effort of the attack varies between a full key search (and no observed session) and the right key is found without search (number of session as given in Sects. 4 and 5).

6.2 Cipher text only attacks

In all our previous analysis we have assumed that the attacker is able to observe the pseudo random bytes generated by RC4, i.e. we assumed that the attacker knows the enciphered plain text. Now we want to show how to convert our known plain text attacks to cipher text only attacks.

Let have a look at the basic variant of the 1-round attack. We can not obverse the first pseudo random byte x, but we can observe the encrypted message $c = m + x \mod n$.

The starting point for the 1-round attack was the observation

$$P(t \equiv 1 - x \mod n) \approx \frac{1}{e} \cdot \frac{2}{n} + \left(1 - \frac{1}{e}\right) \cdot \frac{n-2}{n(n-1)} \approx \frac{1.36}{n}.$$

Let now assume that the message m is equal m' with probability $\frac{2}{n}$ and that all other message have the same probability. Then we may conclude

$$\begin{aligned} P(t \equiv 1 - (c - m') \mod n) \approx{} & \frac{1}{e} \cdot \frac{2}{n} \cdot \frac{2}{n} + \frac{1}{e} \cdot \left(1 - \frac{2}{n}\right) \cdot \frac{n-2}{n(n-1)} \\ & + \left(1 - \frac{1}{e}\right) \cdot \frac{n-2}{n(n-1)} \cdot \frac{2}{n} + \left(1 - \frac{1}{e}\right) \cdot \left(1 - \frac{n-2}{n(n-1)}\right) \\ & \cdot \frac{n-2}{n(n-1)} \approx \frac{1}{n} + \frac{0.36}{n^2}. \end{aligned}$$

The difference from $\frac{1}{n}$ is smaller but still significant.

We may use the estimation of the information obtained per step to estimate the number of sessions needed in this scenario is about $15.4 \ln(n) n^3$. (For $n = 256$ this would be 130,000,000 session. But the redundancy of the message is only $\approx 2 \cdot 10^{-4}$. Many real messages have a redundancy of 0.5 or higher. For such a message we would need a smaller number of sessions, i.e. we will succeed with only 1,000,000 session in many cases.)

We see that, if we have less information about the encrypted message, we need more cipher test for a successful attack. But even then the number of needed sessions stays very small.

7 RC4 variants

In this section we want to discuss variants of the RC4 algorithm with respect to attacks described in the preceding sections.

7.1 Modification of the key scheduling algorithm

It is notable that even small variants of the RC4 algorithm can improve it strength against existing attacks.

If we change line 7 of the key scheduling algorithm (Algorithm 1) from

for i **from** 0 **to** n−1 **do**

to

for i **from** n−1 **downto** 0 **do**

we obtain a stronger algorithm.

Attacks that uses chosen initialization vectors (FMS-attack and our 2-round attack) needs a manipulation of the first S-Box bytes, that enforces a certain action in the first steps of the pseudo random generator. The change of the key scheduling describe above, allow only manipulations that affects the last bytes of the S-box. But such manipulations are useless for the FMS-attack and our 2-round attack.

Our 1-round attack can be still applied to the modified RC4 algorithm. But this time we must require that $S[n-1]$ is not changed in the last step of the key scheduling phase and in the first steps of the pseudo random generator. The probability that this happens is only $\approx \frac{1}{e^2}$. This means that the success probability of the attacks drops to

$$P = \frac{1}{e^2}\frac{2}{n} + \left(1 - \frac{1}{e^2}\right) \cdot \frac{n-2}{n(n-1)} \approx \frac{1.14}{n}. \tag{8}$$

With other words we need more sessions for a successful attack.

Further more we must investigate the $(n-1)$th pseudo random byte instead of the first pseudo random byte. Since in most applications the first bytes are highly regular (addresses, protocol information, etc.) it is easier to obtain information about the first bytes.

7.2 Modification of the pseudo random generator

An other interesting observation is that $S[k]$ is uncorrelated to $S[i]$. This leads to a interesting modification of the RC4 pseudo random generator (Algorithm 2). We move line 8 behind line 10. For our 1-round attack this means, Theorem 1 can still be used to guess the value $S[j]$. But now $S[j]$ is no longer a function from $K[0]$ and $K[1]$ and it seems that we have no good method to find a relation between $S[j]$ and the key K. Even worse j varies between all sessions and we do not know the value j. In other words this simple modification of the pseudo random generator can block our attack. Other weaknesses of the RC4 algorithm (like the FMS-attack, fortuitous states, correlation between different outputs) are not affected by this modification. To my knowledge the only difference between the original and modified variant of RC4 is that that modified variant is stronger against attacks based on Theorem 1.

7.3 The NGG generator

An interesting proposal for a generalization of the RC4 algorithm is due to Nawaz et al. [7]. They want do solve the following problem:

If we want the RC4 algorithm to deal with 32 or 64 bit words instead of bytes, the direct generalization would need a S-box with 2^{32} or 2^{64} entries, respectively. This is not acceptable, because the large amount of memory need for the S-box. The three authors suppose the following solution to that problem.

Fix two numbers $n, m \in \mathbb{N}$ with $n|m$ (typical values are $n = 256$ and $m = 2^{32}$ or $m = 2^{64}$). The RC4(n,m) algorithm works on a S-box of n values in $\mathbb{Z}/m\mathbb{Z}$ and goes like this:

Algorithm 3 NGG key scheduling phase

```
1. {Initialization}
2. for $i$ from 0 to $n-1$ do
3.   $S[i] := a_i$ {the values $a_i$ are fixed parameters}
4. end for
5. $j := 0$
6. {Generate a random S-box}
7. for $i$ from 0 to $n-1$ do
8.   $j := (j + S[i] + K[i \mod l]) \mod n$
9.   Swap $S[i]$ and $S[j]$
10.   $S[i] := S[i] + S[j] \mod m$
11. end for
```

Algorithm 4 NGG pseudo random generator

```
1. {Initialization}
2. $i := 0$
3. $j := 0$
4. {generate pseudo random sequence}
5. loop
6.   $i := (i + 1) \mod n$
7.   $j := (j + S[i]) \mod n$
8.   Swap $S[i]$ and $S[j]$
9.   $k := (S[i] + S[j]) \mod n$
10.   print $S[k]$
11.   $S[k] := S[i] + S[j] \mod m$
12. end loop
```

The difference to the RC4 algorithm lies mostly in the line 10 in Algorithm 3 and line 11 in Algorithm 4, respectively. The S-Box contains no longer a permutation of all values in $\mathbb{Z}/m\mathbb{Z}$ but only a n-tuple of not necessary distinct values.

Since Theorem 1 uses only the lines 8 and 9 of the pseudo random generator, we can read the output modulo n and get:

Theorem 2 *Suppose all $S[i]$ independent and are uniform distributed in $\mathbb{Z}/m\mathbb{Z}$. Then*

$$P(S[j] + S[k] \equiv i \mod n) = \frac{2}{n} + O\left(\frac{1}{n^2}\right) \tag{9}$$

$$P(S[j]+S[k] \equiv i \mod n \mid S[j] \equiv x \mod n) = \frac{2}{n} + O\left(\frac{1}{n^2}\right) \textit{ for all } x \in \{0, \ldots, n-1\} \tag{10}$$

For $c \not\equiv i \mod n$ we have:

$$P(S[j] + S[k] \equiv c \mod n) = \frac{n-2}{n(n-1)} + O\left(\frac{1}{n^2}\right) \tag{11}$$

$$P(S[j] + S[k] \equiv c \mod n \mid S[j] \equiv x \mod n) = \frac{n-2}{n(n-1)} + O\left(\frac{1}{n^2}\right) \textit{ for all } x \in \{0, \ldots, n-1\} \tag{12}$$

Proof The proof follows the idea of the proof of Theorem 1. The only problem is, that for a given S-box S and a fixed value x the value of j with $S[j] \equiv x \mod n$ is no longer unique. There may be more than one possible j or there may be no possible j. That make the counting more difficult and is the reason for the $O(\frac{1}{n^2})$ terms.

We show now how to compute the probability

$$P(S[j]+S[k]\equiv i \mod n \mid S[j]\equiv x \mod n)$$

for the case $i\not\equiv 2x \mod n$.

First we count the number of possible internal (reduced modulo n). We may choose j (n possibilities) which determines $S[j]\equiv x \mod n$ and then we may choose the remaining $n-1$ states of the S-box (n^{n-1} possibilities).

Now we count the number of states with $S[j]+S[k]\equiv i \mod n$ as in Theorem 1 we write the equivalently as $S[k]+k\equiv S[i]+i \mod n$.

We distinguish three cases:

1. $i=k$
   Then $S[i]\equiv k-S[j]\equiv i-x\not\equiv x \mod n$ (remember that we assumed $i\not\equiv 2x \mod n$). We may choose now $j\neq i$ ($n-1$ possibilities) which determines $S[j]\equiv x \mod n$. For the remaining $n-2$ entries of the S-box we have n^{n-2} possibilities.
2. $i\neq k$ and $k\not\equiv 2x \mod n$
   There are $n-2$ possible values for k. After we have chosen k the values $S[i]\equiv k-x\not\equiv x \mod n$ and $S[k]=S[i]+i-k=i-x\neq x$ are determined.
   We may choose now $j\neq i,k$ ($n-2$ possibilities) and the remaining $n-3$ S-box entries (n^{n-3} possibilities).
3. $i\neq k$ and $k\equiv 2x \mod n$
   Then $S[i]\equiv k-x\equiv x \mod n$ and $S[k]\equiv x+i-k\not\equiv x \mod n$.
   We may now chose either $j=i$, which leaves n^{n-2} possibilities for the remaining S-box or we choose $j\neq i,k$ ($n-2$ possibilities) which leaves n^{n-3} possibilities for the remaining S-box.

Altogether we have

$$(n-1)n^{n-2}+(n-2)(n-2)n^{n-3}+n^{n-2}+(n-2)n^{n-3}=2n^{n-1}-3n^{n-2}+2n^{n-3}$$

internal states with $S[j]+S[k]\equiv i \mod n$ and $S[j]\equiv x \mod n$.

This proves

$$P(S[j]+S[k]\equiv i \mod n \mid S[j]\equiv x \mod n)=\frac{2}{n}-\frac{3}{n^2}+\frac{2}{n^3}$$

for $i\not\equiv 2x \mod n$.

The other probabilities can computed the same way. Since the attack of the NGG generator is not the primary goal of this article we omit the details. □

That the probability is now $\frac{2}{n}+O(\frac{1}{n^2})$ instead of $\frac{2}{n}$ is irrelevant for our attack. We can therefore study the outputs of the NGG-Generator modulo n and apply any of attacks described in the previous sections. This helps us to reconstruct the values of the key modulo n.

But if all key values are known modulo n, we know always the exact value of i, j and k. This means we are able to identify the output of the NGG pseudo random generator as S-box value. This reconstructs the S-box and with the S-box we know the key.

Similarly the FMS-attack can be adapted to the NGG generator.

Our study shows that NGG generator fails against the same attacks as the RC4 algorithm, but it introduces potentially additionally weaknesses. Personally I see a problem with line 11 of Algorithm 4 that enforces $S[k]\equiv k \mod n$. For that reasons I can not recommend the NGG generator.

7.4 RC4A

An other interesting variant of the RC4 algorithm is the RC4A algorithm described in [12]. The idea is to use two S-boxes in “parallel” to obtain a stronger algorithm.

Algorithm 5 RC4A pseudo random generator

```
{initialization}
$i := 0$
$j_1 := 0 \quad j_2 := 0$
{generate pseudo random sequence}
loop
  $i := (i + 1) \mod n$
  $j_1 := (j_1 + S_1[i]) \mod n$
  Swap $S_1[i]$ and $S_1[j]$
  $k_2 := (S_1[i] + S_1[j]) \mod n$
  print $S_2[k_2]$
  $j_2 := (j_2 + S_2[i]) \mod n$
  Swap $S_2[i]$ and $S_2[j]$
  $k_1 := (S_2[i] + S_2[j]) \mod n$
  print $S_1[k_1]$
end loop
```

In each step we produce two output bytes. Since k_2 is computed from the values of the first S-box and both S-boxes are independent there cannot be a correlation between $S_1[j_1]$ and $S_2[k_2]$. But there is still an analogon to Theorem 1.

Theorem 3 *Assume that a permutations have the same probability and S_1 and S_2 are independent. Then*:

$$P(S_1[j_1] + S_1[k_1] + S_2[j_2] + S_2[k_2] \equiv 2i \mod n) = \frac{1}{n-1}. \tag{13}$$

Proof The proof is very similar to the proof of Theorem 1.

We use $k_2 \equiv S_1[i] + S_1[j_1] \mod n$ and $k_1 \equiv S_2[i] + S_2[j_2] \mod n$ to write

$$S_1[j_1] + S_1[k_1] + S_2[j_2] + S_2[k_2] \equiv 2i \mod n$$

as

$$(k_1 + S_1[k_1]) + (k_2 + S_2[k_2]) \equiv (i + S_1[i]) + (i + S_2[i]) \mod n.$$

Now we count the states that satisfies this condition. For this we have to distinguish several cases:

1. $k_1 = k_2 = i$
   In this case we may choose S_1 and S_2 with out any restriction. Thus we have $(n!)^2$ possible combinations.
2. $k_1 = i, k_2 \neq i$
   In this case we may choose S_1 as we want ($n!$ possibilities). Next we may choose k_2 ($n-1$ possibilities), than we have to choose $S_2[i]$ (n possibilities) which determines $S_2[k_2]$. After all we may choose the remaining part of S_2 ($(n-2)!$ possibilities). All together we have $(n!)((n-1)n(n-2)!) = (n!)^2$ possibilities in this case. (Compare this with Theorem 1).

3. $k_1 \neq i, k_2 = i$
   This is analog to the previous case $(n!)^2$ possibilities.
4. $k_1 \neq i, k_2 \neq i$
   We distinguish two subcases:

   (a) $k_1 + S[k_1] \equiv i + S_1[i] \mod n$
   This is equivalent to $k_2 + S[k_2] \equiv i + S_2[i] \mod n$.
   We may count the number of possibilities for both S-boxes separately. As in Theorem 1 or in the two cases above we find $(n!)^2$ possibilities in this case.

   (b) $k_1 + S[k_1] \not\equiv i + S_1[i] \mod n$
   In this case we may first choose k_1 (n possibilities) and $S_1[i]$ (n possibilities). Now we choose $S_1[k_1]$ ($n-2$ possibilities, since $S_1[k_1] \neq S_1[i]$ and $k_1+S[k_1] \not\equiv i+S_1[i] \mod n$ by assumption). There are $(n-2)!$ possibilities left to choose the remaining part of S_1.
   By assumption $k_2 \neq i$. Further more $(k_1 + S[k_1]) + k_2 \not\equiv (i + S_1[i]) + i \mod n$, since $S_2[k_2] \neq S_2[i]$. Thus we have $n - 2$ possibilities to choose k_2. Now we may choose $S_2[i]$ (n possibilities), which determines $S_2[k_2]$. For the remaining part of S_2 we have $(n - 2)!$ possibilities.
   This makes $[(n-1)n(n-2)(n-2)!][(n-2)n(n-2)!]$ possibilities in this case.

Altogether there are $n \cdot n!(n-2)!$ internal states with

$$(k_1 + S_1[k_1]) + (k_2 + S_2[k_2]) \equiv (i + S_1[i]) + (i + S_2[i]) \mod n.$$

Since the total number of possible internal states is $(n \cdot n!)^2$ this proves the theorem. □

There are also analogons to the Eqs. 2–4 of Theorem 1. But since this paper has not the primary goal to analyze RC4A we omit these details.

The correlation of Theorem 3 is weaker than the one proved in Theorem 1, but it is still strong enough to be exploited in an attack.

Theorem 3 allows us to estimate $S_1[j_1] + S_2[j_2] \mod n$. If we are able to use this information to obtain the main key, depends on the key scheduling algorithm.

Suppose we have two independent sub keys K_1 and K_2 and use these sub key with normal RC4 key scheduling algorithm (Algorithm 1) to initialize S_1 and S_2 respectively. Then we can use Theorem 3 and the technique of our 1-round attack to compute a relation between K_1 and K_2. This reduces the effective key length by $\frac{1}{2}$.

If we generate K_2 form K_1 by a simple RC4 algorithm as suggested in [12]. We can mount the following attack. Assume that the initialization vector precedes the main key. With a chosen initialization vector we may enforce that first two bytes of K_2 depend in a simple way from the first two bytes of the main key. (Use initialization vectors similar to the one used in the FMS-attack.) Now we analyze the first two out-put bytes of the RC4A pseudo random generator with Theorem 3. $S_1[j]$ depends only on the first two bytes of K_1. These bytes are part of the initialization vector which are know to the attack. Thus the attack may compute $S_2[j_2]$ from the observed values. This byte depends only on the first two bytes of main key. The attack follows now the pattern of 1-round attack described in Sect. 4.

For the attack described above we need more information about the initialization vector than for the FMS-attack, further more the correlation is rather weak. This means that this attack needs an unrealistic high number of observed sessions.

As a conclusion of this subsection we may say: RC4A has weaknesses that are very similar to the weaknesses of RC4, but the consequences are not so dramatic.

8 Conclusion

The 1-round attack described in this article needs in opposite to the FMS-attack no weak initialization vectors. It is therefore very difficult to avoid that kind of attack and the attack succeeds with a fewer number of observed sessions. In the cases the initialization vector precedes the main key the FMS-attack still remains a very attractive variant, because its simple structure. Especially one needs only the first pseudo random byte to do the FMS-attack. If the initialization vector follows the main key the new attack is clearly better.

The 2-round attack proves that it is possible exploit the weakness of the key scheduling, even after many steps of the pseudo random generator. The attack needs more work than the 1-round attack but remains still in a practicable region.

If one wants to use RC4 he should follow the advice given by Mironov [10] and discard the output of the first 12 rounds.

An other interesting idea is to compute the session key from the main key and the initialization vector via a hash function [1]. The hashing would avoid all attacks similar to the FMS-attack or the attacks described in this work. On the other hand we must implement a cryptographic hash function with would eliminate one of the main advantages of RC4, the quick and simple implementation.

Acknowledgments The research of the author takes place within the project "Linear codes and cryptography" of the Fund for Scientific Research Flanders (FWO-Vlaanderen) (Project nr. G.0317.06), and is supported by the Interuniversitary Attraction Poles Programme-Belgian State-Belgian Science Policy: project P6/26-Bcrypt.

References

1. Ferguson N., Schneier B.: Practical Cryptography. Wiley Publishing, Inc. (2003).
2. Fluhrer S., Mantin I., Shamir A.: Weakness in the Key Scheduling Algorithm of RC4. In: Selected Areas in Cryptography, vol. 2259 of LNCS, pp. 1–24. Springer, Berlin (2001).
3. Fluhrer S.R., McGrew D.A.: Statistical analysis of the alleged RC4 keystream generator. In: Proceedings of the 7th International Workshop on Fast Software Encryption, vol. 1978 of LNCS, pp. 19–20. Springer, Berlin (2000).
4. Golić J.Dj.: Linear statistical weakness of alleged RC4 keystream generator. In: Advances in Cryptology – EUROCRYPT '97, vol. 1233 of LNCS, pp. 226–238. Springer, Berlin (1997).
5. Golić J.Dj.: Linear models for a time-variant-permutation generator. IEEE Trans. Inform. Theory **45**(7), 2374–2382 (1999).
6. Golic J.Dj.: Iterative probabilistic cryptanalysis of rc4 keystream generator. In: ACISP 2000, pp. 220–233 (2000).
7. Gupta K., Nawaz Y., Gong G.: A 32-bit RC4-like keystream generator. Technical Report CACR 2005-21, Center for Applied Cryptographic Research, University of Waterloo, 2005. http://www.cacr.math.uwaterloo.ca/tech_reports.html (2005).
8. Mantin I.: Predicting and distinguishing attacks on RC4 keystream generator. In: Cramer R. (ed.), Advances in Cryptology – EUROCRYPT 2005, vol. 3494 of LNCS, pp. 491–506. Springer, Berlin (2000).
9. Mantin I., Shamir A.: A practical attack on broadcast RC4. In: Matsui M. (ed.), Revised Papers from the 8th International Workshop on Fast Software Encryption, vol. 2355 of LNCS, pp. 152–164. Springer, London (2001).
10. Mironov I.: (Not so) random shuffles of RC4. In: Advances in Cryptology – CRYPTO 2002, vol. 2442 of LNCS, pp. 304–319. Springer, Berlin (2002).
11. Mister S., Tavares S.E.: Cryptanalysis of RC4-like ciphers. In: Selected Areas in Cryptography (Kingston, ON, 1998), vol. 1556 of LNCS, pp. 121–143. Springer, Berlin (1999).
12. Paul S., Preneel B.: A new weakness in the RC4 keystream generator and an approach to improve the security of the cipher. In: Fast Software Encryption 2004, vol. 3017 of LNCS, pp. 245–259 (2004).
13. Wald A.: Sequential Analysis. Wiley and Sons, New York (1947).