UNIVERSITY OF LJUBLJANA
FACULTY OF COMPUTER AND INFORMATION SCIENCE

Jakob Makovac

RC4 Cipher

Project

Mentor: Prof. Aleksandar Jurišić, PhD

Ljubljana, 2017

Contents

1. Introduction

RC4 cipher is one of the most popular stream ciphers in the world. In 2015 it was used to protect around 30 percent of SSL traffic. Today, however, its popularity is falling due to security weaknesses. Our main goal is to study RC4 cipher from the security aspect.

There has been a lot of research done on RC4, since the structure of the algorithm has been made public. Most important for our work was a paper written by Tsunoo et al. It proposed a distinguisher for two variants of RC4 algorithm. I will describe both of them and the distinguisher later. In 2014 Rivest proposed an improved version of RC4. As we will see later, Subhadeep and Takanori described a distinguisher in their paper in 2016.

The goal of this project is to explain how RC4 works and to see how even very simple algorithms can be employed to provide quite decent pseudo-random generation. And more importantly the goal of this project is also to understand the weaknesses of said algorithm.

In the first part we present the historical aspect of RC4 cipher and its implementations. Second part is about the encryption procedure and its properties. Then we will move on to security, explaining strengths and weaknesses of RC4 and different classes of attacks that are possible. We will conclude with some proposed improvements to RC4 and evaluate them.

2. History

RC4 was designed within RSA Security company, by Ron Rivest to be specific in 1987. The name RC4 stands for "Rivest Cipher 4". Initially it meant to be a trade secret, however its design was leaked in 1994. It was anonymously posted online and spread quickly. RSA Security did not officially confirm nor deny the validity of leaked algorithm, but it produced the same output as licensed RC4 found in proprietary software. During the years it became part of numerous protocols and standards, for example: WEP in 1997, WPA in 2003, SSL in 1995 and TLS in 1999. Since the name RC4 is trademarked, it is often common to see the ARC4 name used instead. This stands for "Alleged RC4". For the sake of simplicity I will use RC4 throughout this paper.

As stated before RC4 was used to protect around 30 percent of all SSL traffic in 2015. It used to be even higher, around 50 percent in 2011 and then dropped ever since, due to different weaknesses that were being found. The high usage in 2011 is entirely due to BEAST attack that targeted block ciphers at that time. RC4, being a stream cipher, was not affected by BEAST attack and was consequently recommended as a workaround. Today its usage is at its lowest point, since its use in TLS was prohibited by IETF in 2015. Nowadays it is used in Mac OS as a random number generator and it is often used as comparison for new random number generators.

3. Encryption Process

RC4 encryption procedure is surprisingly simple considering its widespread use. Unlike the majority of stream ciphers it does not make use of linear feedback shift registers. It is instead optimized for efficient software implementation. It runs around two times faster than AES. It requires 256 bytes of memory for the state array S, 5 to 32 bytes of memory for the key and two integers for two indices i and j.

The RC4 encryption procedure is split into two parts. First is the Key Scheduling Algorithm (KSA) and the second one is the Pseudo-Random Generation Algorithm (PRGA). The KSA takes the key and scrambles the identity permutation of the state array. The final state of the state array is passed on to the PRGA. It takes the state array and preforms the desired amount of rounds of permutations. This is normally the length of the message, so it can be XORed with the output of the PRGA. I will describe both KSA and PRGA in detail below.

3.1 KSA

The KSA initializes the state array S with values 0 to 255. The key is expanded with repetition to match the length of the state array. Indices i and j are initialized to 0. KSA then runs for 256 rounds. Every round, index i is incremented by 1, so every element of the state array is affected at least once. Index j is calculated in the following way:

$$j = S[i] + k[i] \bmod 256 \tag{3.1}$$

The values at $S[i]$ and $S[j]$ are then swapped. The algorithm is also described below. After 256 rounds the state array is passed on to PRGA.

Algorithm 1 RC4 Key Scheduling Algorithm (KSA)

1: **for** $i = 0$ to $N - 1$ **do**
2: $\quad S[i] \leftarrow i$
3: **end for**
4: $j \leftarrow 0$
5: **for** $i = 0$ to $N - 1$ **do**
6: $\quad j \leftarrow j + S[i] + K[i \bmod \ell] \bmod N$
7: $\quad$ swap($S[i]$,$S[j]$)
8: **end for**

Figure 3.1: Key scheduling algorithm for RC4

3.2 PRGA

This is the main part of RC4 cipher. It takes the state array from the KSA and produces the amount of pseudo-random keystream needed. The procedure is similar to that of KSA but with minor differences. The state array is taken from the KSA. Indices i and j are again initialized to zero and index i is incremented by 1 every round. Index j is calculated as follows[1]:

$$j = S[i] + j \bmod 256 \tag{3.2}$$

The values $S[i]$ and $S[j]$ are swapped and output k is calculated in the following way:

$$k = S[i] + S[j] \bmod 256 \tag{3.3}$$

It is easier to visualize the algorithm with a figure, so we provide one below.

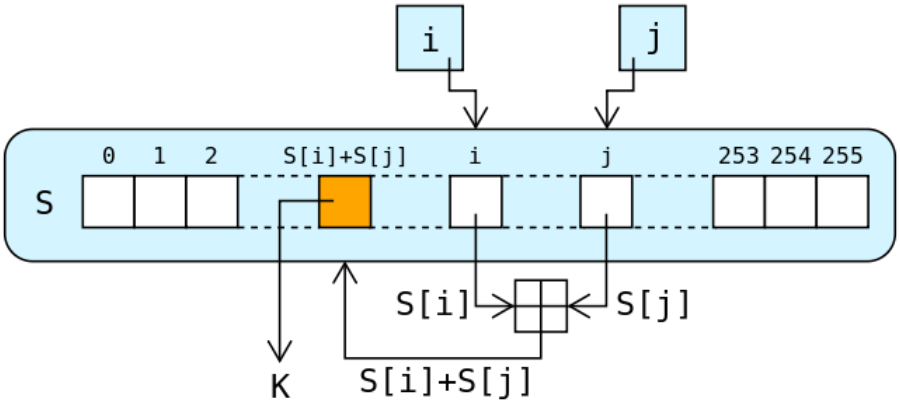

Figure 3.2: Pseudo-Random Generation Algorithm

4. Security

In this section we are going to see different types of attacks used against RC4 cipher. Being a stream cipher, the RC4 is vulnerable to general attacks on stream ciphers like reused key attack and bit-flipping attack if implemented incorrectly. Reused key attack is effective, when two messages are encrypted with the same key. An attacker can XOR the two ciphertexts and obtain first message XOR second message, because:

$$c_1 \oplus c_2 = m_1 \oplus k \oplus m_2 \oplus k = m_1 \oplus m_2 \tag{4.4}$$

where c_1 is the first ciphertext and c_2 is the second ciphertext. If both messages are random strings of characters it is not easy to decrypt $m_1 \oplus m_2$. Messages however are usually not random, which means the frequency of characters will be predictable. A modern computer can decipher such ciphertexts efficiently.

Next we describe the bit-flipping attack. This one is effective when the attacker knows the message or at least part of the message. The attacker can effectively change the known part of the message with his own without knowing the key. He just has to XOR the ciphertext with known message XOR new message, like this:

$$m_1 \oplus k \oplus m_1 \oplus m_2 = m_2 \oplus k \tag{4.5}$$

where m_1 is the original message and m_2 is the new message.

In the following section we present attacks specific to RC4. They can be classified into 3 classes.

4.1 Weak keys

There exists a class of weak keys that causes a part of the state array to stay the same during the KSA. The part that stays the same is the least significant bits. They in turn determine the least significant bits of the state array during PRGA and furthermore they determine the least significant bits in the output keystream. If the attacker can spot a weak key being used he can recover significant portion of plaintext bytes from the beginning of the message [2].

4.2 State recovery

Thanks to the massive state-space, state recovery attack for RC4 is quite challenging. Number of possible states is around 256!, which is roughly 2^{1600}. This means that even the best attacks in this category today are still of complexities around 2^{200}.

4.3 Biases and distinguishers

Most of the attacks in this category are focused on the heavily biased first 100 bytes of the output, but they are ineffective if we just drop the first n bytes, n being the number from 100 on. Riddhipratim et al. have shown that information about index j is always present in the keystream, no matter how many bytes of output we drop, so we will focus on this particular result [3].

We will not present the complete proof, because it would be too long. However it can be found in literature [3]. We will just describe the results and implications in this paper.

The idea is to check the number of all possible permutations of the state array given different values of indices i, j and output z. Number of all possible permutations is computed in a tree-like fashion. We condition on one of the variables on each level and we get a result in the end given certain conditions on those variables. We get three different probabilities that deviate from a random sequence:

1. $P(j = z \mid i \text{ odd}) \geq \frac{1}{N} + \frac{1}{N^2}$
2. $P(j = z \mid i \text{ even}) \leq \frac{1}{N} - \frac{1}{N^2}$
3. $P(j = z \mid 2z = i + 1) = \frac{2}{N} - \frac{1}{N(N-1)}$

Those results show that z is either positively or negatively biased to j, where N is the size of state array S. Furthermore, we can see that those results hold for z at any stage during algorithm, no matter how many bytes are dropped or how long the KSA part is. We can also see that the magnitude of this bias is $\frac{1}{N^2}$. The biases and the general idea behind the proof is presented later in our work when we examine the procedure more closely for each proposed improvement to RC4 algorithm.

5. Improvements

There have been some attempts to improve RC4 after several publications of its weaknesses and practical attacks exploiting them. I will describe 3 of them and go over their properties.

5.1 RC4A

It was proposed in 2004. Compared to RC4 it uses two state arrays S_1 and S_2 instead of just one, so it also need two indices j_1 and j_2 but only one index i. When i is incremented the first part of the procedure is identical to RC4, but the output is looked up in the second state array:

$$k = S_2[S_1[i] + S_1[j_1]], \tag{5.6}$$

where k is the output. Then the original RC4 procedure of adding $j_2 += S2[i]$ and swapping values happens without incrementing i and second output is looked up in the first state array:

$$k = S_1[S_2[i] + S_2[j_2]]. \tag{5.7}$$

The process continues with incrementation of i and cycles. Although the algorithm requires more steps than RC4, it takes similar amount of time if parallelised. It is also stronger in terms of security, but its output was distinguished from truly random sequence after 2^{23} output bytes by Tsunoo et al. [4].

They assume that the element at index 1 of the array S_1 equals 2. They go on to prove that if the assumption holds, the first and the third bytes of the output can never be the same. This is done by showing that given the assumption, the values at relevant places are never swapped. So it follows that they cannot be the same, since the array is initialized with unique elements. It can be calculated that the probability of such event differs from a truly random sequence by 2^{-16}. They confirm the calculation experimentally. The success probability of their attack after 100 independent trials is 53% with 2^{23} output bytes.

5.2 VMPC

VMPC or Variably Modified Permutation Composition is another attempt at making RC4 stronger. It tries to mitigate the bias of first 100 or so bytes of keystream by running the KSA for 768 rounds instead of 256. There is also a change in how we compute the index j and how the output is produced:

$$j = S[j + S[i]], \tag{5.8}$$

$$k = S[S[S[j]] + 1], \tag{5.9}$$

where k is the output. This procedure improves the security slightly, but it was attacked in the same paper as RC4A and its output was distinguished from a truly random sequence using 2^{38} output bytes by Tsunoo et al. [4]. They use a procedure similar to RC4A to prove the bias exists. They assume the first element of the state array S equals 0. Given this assumption they prove the first two output bytes of the keystream can never be the same. This happens because the algorithm fixes i to 0 at the start. By following the relevant places we can see that values are never swapped and since we are looking at two different elements at indices i and j they cannot be the same. The probability of such event is different from a truly random sequence by 2^{-24}.

5.3 Spritz

In 2014 Rivest and Schuldt proposed a modified version of RC4 named Spritz. Changes to the algorithm are the following. Another index w is used, which is relatively prime to the size of the state array. Instead of incrementing i by one, it is incremented by w:

$$i = i + w \bmod 256. \tag{5.10}$$

This still ensures we swap each value at least once, while i does not take predictable values. On top of w, another new index k is used. So generating output goes as follows:

$$j = k + S[j + S[i]], \tag{5.11}$$

$$k = k + i + S[j], \tag{5.12}$$

$$z = S[j + S[i + S[z + k]]], \tag{5.13}$$

where z is the output. This not only increases the security of the cipher but it also provides the possibility to employ it as a hash function or encryption that supports authentication thanks to it being constructed as a sponge function. Unfortunately, this version also does not improve sufficiently.

In 2016 Subhadeep and Takanori calculated the number of output bytes required to distinguish Spritz cipher output from a random sequence. They used a theorem which states that we need $\mathcal{O}(\frac{1}{pq^2})$ samples to distinguish an event happening in two different distributions, where p is the probability that the event e happens in the distribution X and $p(1 + q)$ is the probability that e happened in the distribution Y. X represents an ideal random stream and Y represents probability distribution that the first two bytes were produced by Spritz cipher. Event e denotes the first two bytes of output being the same. That happens with probability $\frac{1}{N^2}$ in X and $\frac{1}{N^2}(1 + \frac{3}{N^2})$ in Y, where N is the size of the state array S. p and q are then $\frac{1}{N^2}$ and $\frac{3}{N^2}$ respectively, which gives us $\mathcal{O}(2^{44.8})$.

That covers the bias in the first two bytes of the keystream. This bias can be mitigated by dropping a portion of keystream before producing output. Subhadeep and Takanori however also calculated the number of samples required for distinguishing Spritz cipher output from a random sequence utilising a bias that is present throughout the keystream. The algorithm is constructed so the index i is set to 0 after every N rounds. The assumption that the first element equals 0 holds every N cycles on average, so we have to multiply the result from above with N^2. The result is $\mathcal{O}(2^{60.8})$ samples. Subhadeep and Takanori also confirmed the result experimentally. They successfully attacked Spritz ciphers with $N = 16$ and $N = 32$. In both cases they needed less samples than calculated. [5].

6. Conclusion

As we have seen, RC4 was a very widespread and popular stream cipher. Not surprisingly considering its simplicity and considering it is not very computationally complex. Simplicity however came at a cost of having significant biases in the early portions of output and some lesser biases throughout the keystream. We have examined some improved algorithms based on the original idea and concluded that they were showing similar weaknesses. From 2015 on, RC4 is not used for encryption of data any more. It is however still used as a random number generator in applications, where having very minor biases is not essential. It is one of the choices for a random generator function in Mac OS and it is also used as comparison for any newly developed pseudo-random generator. The reason for this is simplicity of the PRGA part of the algorithm. That allows us to connect the PRGA procedure and biases in the output. As described in our work some of those biases have been successfully exploited for distinguishing RC4 output from a random sequence more or less efficiently. If no portion of keystream is dropped we have some relatively effective distinguishers. On the other hand, if a portion of keystream is dropped the best distinguishers still requires around 2^{60} samples.

7. Literature

[1] M. Subhamoy, P. Goutam, S. S. Gupta, S. Sarkar. "(Non-)Random Sequences from (Non-)Random Permutations - Analysis of RC4 Stream Cipher". *Journal of Cryptology* 27 (2014), 67–108.

[2] Imperva Inc. "Attacking SSL when using RC4". *Hacker Intelligence Initiative* (2015).

[3] B. Riddhipratim, G. Shirshendu, M. Subhamoy, P. Goutam. "A Complete Characterization of the Evolution of RC4 Pseudo Random Generation Algorithm". *Journal of Mathematical Cryptology* 2.3 (2008), 257–289.

[4] Y. Tsunoo, T. Saito, H. Kubo, M. Shigeri, T. Suzaki, T. Kawabata. "The Most Efficient Distinguishing Attack on VMPC and RC4A" (2005).

[5] B. Subhadeep, I. Takanori. "Cryptanalysis of the Full Spritz Stream Cipher". *Lecture Notes in Computer Science* 9783 (2016), 63–77.