RC4OK. An improvement of the RC4 stream cipher

Khovayko O.
Schelkunov D., Ph.D.
Emercoin
olegh@emercoin.com
ReCrypt LLC
d.schelkunov@gmail.com, schelkunov@re-crypt.com

Abstract

In this paper we present an improved version of the classical RC4 stream cipher. The improvements allow to build lightweight high-performance cryptographically strong random number generator suitable for use in IoT and as a corresponding component of operating systems. The criterion for high performance is both a high speed of generating a stream of random numbers and low overhead costs for adding entropy from physical events to the state of the generator.

Keywords: stream cipher, random number generator, RC4

1 Introduction

Modern operating systems have a built-in source of random numbers that can be used by application programs. This source contains a pseudo-random number generator (PRNG), the state of which changes both when reading data and adding entropy - asynchronous data from physical events in the computer (mainly interrupts from devices). The latter guarantees that the random stream is truly random, that is, it cannot be reproduced given the initial state.

In a multiprocessor environment, the asynchronous addition of entropy is very nontrivial, since the simultaneous modification of the PRNG state by various processes can violate its state (PRNG internal state) and disable it, or else lead to its degradation, that is, to deviation of the parameters of the output stream from the random one.

To correctly add entropy, various access sharing mechanisms are used. For example, in FreeBSD, asynchronous events are queued, from which the data is retrieved by a thread serving the PRNG.

We have developed a simple cryptographically strong PRNG that allows you to add entropy without locks and the risk of breaking the generator state. Thus, complex synchronization mechanisms become unnecessary, which will drastically reduce the system kernel latency when adding entropy, and thereby increase its performance. The RC4 generator [1] was chosen as the basis for the following reasons:

- extremely simple implementation;
- the generator is stream and byte-oriented, allowing it to generate byte sequences of any length;
- high performance.

RC4 stream cipher is currently losing popularity due to a number of vulnerabilities found in it [2] - [6] . We have modified a classical RC4 [1] to fix them and to improve randomness properties.

2 The original RC4 algorithm

As mentioned above, the original RC4 is taken as a basis, to which a number of modifications have been added. Let us briefly remind the classical RC4 algorithm. RC4 generates a pseudorandom keystream. This keystream can be used for encryption by combining it with the plaintext using bitwise modulo 2 addition. The same keystream is used for the decryption. To generate the keystream, RC4 uses a secret internal state. This internal state is initialized using the key-scheduling algorithm (KSA). Once the internal state has been initialized, the stream of bits is generated using the pseudo-random generation algorithm (PRGA).

KSA is used to initialize the permutation in the array S (*keylength* is the number of bytes in the key):

$i \leftarrow 0$
while $i \leq 255$ **do**
    $S[i] \leftarrow i$
    $i \leftarrow i + 1$
end while
$j \leftarrow 0$
$i \leftarrow 0$
while $i \leq 255$ **do**

[1] https://github.com/emercoin/rc4ok

$j \leftarrow (j + S[i] + key[i \mod keylength]) \mod 256$
$t \leftarrow S[i]$
$S[i] \leftarrow S[j]$
$S[j] \leftarrow t$
$i \leftarrow i + 1$
end while

Algorithm PRGA of the classical RC4 is presented below:

$i \leftarrow 0$
$j \leftarrow 0$
while *GeneratingOutput* **do**
$i \leftarrow (i + 1) \mod 256$
$j \leftarrow (j + S[i]) \mod 256$
$t \leftarrow S[i]$
$S[i] \leftarrow S[j]$
$S[j] \leftarrow t$
$u \leftarrow (S[i] + S[j]) \mod 256$
$K \leftarrow S[u]$ ▷ Output K
end while

This algorithm produces a stream of $K[0]$, $K[1]$, ...,$K[l]$ (keystream) which could be xor-ed with plaintext to get an encrypted message. To decrypt one the same keystream is used.

At the current moment the classical RC4 has a lot of security issues and is considered to be deprecated [2], [7], [8].

3 The modifications of RC4

As mentioned above the original RC4 uses two index bytes i, j. Moreover, i is increased by one, and j - by the value $S[i]$, where S is an array of 256 bytes filled with numbers 0..255. In our approach (RC4OK), the index i is also byte, but incremented by 11 after each step. The number 11 is prime, and thus provides access to all cells of S. But at the expense of a larger step, a more "loose" update of the array is provided here.

The most important modification concerns the j variable. Unlike RC4, this variable represents a 32-bit number (4 bytes), of which only the least significant byte is used for actual indexing. We define j_0, j_1, j_2, j_3 to be the bytes of the variable j, where j_0 is the least significant one. We can also represent the variable j as two 16-bit words j_{w0} and j_{w1} where j_{w1} is the most significant one (Figure 1).

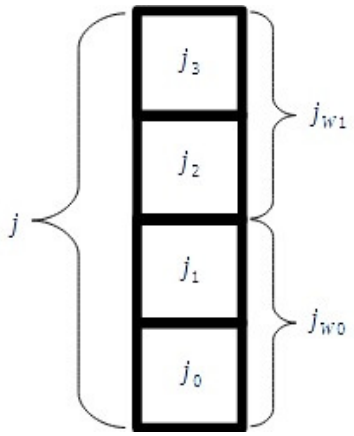

Figure 1. Variable j

At the stage of generating the next byte of the keystream, before adding $S[i]$ to j_0, the algorithm makes a circular left shift by one bit of the entire 32-bit value j. Thus, at the next generation of a byte for the output stream, j_0 depends not only on its previous value and $S[i]$, but also on 31 earlier values of j. This ensures that the generator is pulled out of a possible reduced ring of states, which is proven by practice.

These modifications are practically undemanding to the state of three "additional" bytes of j (j_1, j_2, j_3). Indeed, even if they are zeroed after each step, the generator is reduced to the classic RC4, which, although not ideal, gives a very decent result. In practice, the modification of these three bytes can be performed asynchronously without affecting the main state of the generator - the contents of the S array and the i, j_0 bytes.

The modified PRGA is presented below:

$i \leftarrow S[j \oplus 85]$ ▷ Randomize i
$j \leftarrow 0$ ▷ Clear j
while *GeneratingOutput* **do**
  $i \leftarrow (i + 11) \mod 256$
  $j \leftarrow (j << 1) + (j >> 31)$ ▷ circular left shift by 1
  $j \leftarrow j + S[i]$
  $t \leftarrow S[i]$
  $S[i] \leftarrow S[j_0]$
  $S[j_0] \leftarrow t$
  $u \leftarrow (S[i] + S[j_0]) \mod 256$
  $K \leftarrow S[u]$ ▷ Output K
end while

The proposed algorithm uses the two most significant bytes of j (j_2, j_3)

to asynchronously add entropy. Entropy is added by cyclic left shifting these two bytes by one and arithmetically adding the 16-bit value from the entropy source. See the algorithm below:

while $HarvestingEntropy$ **do**
  $j_{w1} \leftarrow (j_{w1} << 1) + (j_{w1} >> 15)$ ▷ circular left shift by 1
  $j_{w1} \leftarrow j_{w1} + entropy \mod 65536$
end while

In the original RC4, many attacks were developed against the key sheduling algorithm, KSA. For example, [9]. Attacks are related to the leakage of key bits at the beginning of the output stream or the state of the S-box, other than random shuffling immediately after the KSA. The improved version of the original KSA is presented below:

$i \leftarrow 0$
$j \leftarrow 0$
while $i \leq 255$ **do**
  $j \leftarrow j + 233 \mod 256$
  $S[i] \leftarrow j$
  $i \leftarrow i + 1$
end while
$j \leftarrow 0$
$i \leftarrow 0$
while $i \leq 255$ **do**
  $j \leftarrow (j + S[i] + key[i \mod keylength]) \mod 256$
  $t \leftarrow S[i]$
  $S[i] \leftarrow S[j]$
  $S[j] \leftarrow t$
  $i \leftarrow i + 1$
end while

Additionally, after expanding the key using our modified KSA algorithm, we extract 256 bytes from the generator and ignore them. Combined with the previously described modification of j this provides a good mixing of the S array, which eliminates the deviations from randomness for the initial state of the generator.

4 Conclusion

To test the generator, we used the PractRand toolkit of randomness tests. This test is good at finding patterns in the data submitted for verification.

For example, according to the statement of the author of PractRand (and verified by us), the exhaust stream from the original RC4 stops passing the test after 1 terabyte of input data.

Using PractRand we have tested 32 terabytes of the output of our generator. The generator is tested both in the mode without adding entropy, and in the mode of simulating the addition of entropy. Both modes of the generator demonstrate high quality of the output stream, stable operation and no degradation of state [2].

Thus, the generator proposed here can be used both for the initial purpose - a system PRNG, and for replacing the original RC4 when building cryptosystems based on stream ciphers. On the advice of RC4 author Ron Rivest, we named our generator RC4OK, where the suffix OK is derived from the initials of the first author.

This generator is designed as a reference implementation for 32-bit arithmetic, which is used in cheap controllers and other devices of this type. Naturally, modern 64-bit computers can perform these 32-bit operations without sacrificing performance. As an obvious modification focused on 64-bit computers, we can suggest the use of a 64-bit word for storing j. This would increase the bit depth of the added values from entropy sources, or would make the sources multichannel. In addition, this should have a positive effect on the stability of the generator, although it already shows excellent results, it will not get worse. We did not test this modification, focusing on researching the underlying algorithm.

We tested the use of the assembler instruction $crc32$ as an alternative to cyclic shift followed by adding the value of $S[i]$ to j, replacing two machine instructions with one. This modification also passed the PractRand test. Unfortunately, this instruction is only available on x86, so the modification cannot be recommended for widespread use, especially for encryption purposes. However, for generating random numbers within programs or x86-based OS, this modification is quite applicable.

We have have changed the old Emercoin [3] PRNG algorithm with one based on RC4OK. The blockchain download speed has increased up to 3 times.

5 References

[1] Rivest, Ron; Schuldt, Jacob, *https://link.springer.com/content/pdf/10.1007%2F3-540-45473-X*$_1$*3.pdf*, 2014.

[2] https://github.com/emercoin/rc4ok/tree/main/PractRandTest

[3] https://github.com/emercoin/emercoin/

[2] Andrei Popov, *doi:10.17487/RFC7465*, 2015.

[3] John Leyden, *https://www.theregister.com/2013/09/06/nsa$_c$ryptobreaking$_b$ullrun$_a$nalysis/*, 2013.

[4] Green, Matthew, *https://blog.cryptographyengineering.com/2013/03/12/attack-of-week-rc4-is-kind-of-broken-in/*, 2013.

[5] Sepehrdad, P., Vaudenay, S., Vuagnoux, M., "Discovery and Exploitation of New Biases in RC4", *Selected Areas in Cryptography. SAC 2010. Lecture Notes in Computer Science*, **6544**, Springer, 2011, 74–91.

[6] Biham, E., Carmeli, Y., "Efficient Reconstruction of RC4 Keys from Internal States", *LNCS*, **5086**, Springer, 2008, 270-288.

[7] *https://wiki.mozilla.org/Security/Server$_S$ide$_T$LS*, Mozilla, 2015.

[8] *http://blogs.technet.com/b/srd/archive/2013/11/12/security-advisory-2868725-recommendation-to-disable-rc4.aspx*, Microsoft, 2013.

[9] Fluhrer, S., Mantin, I., Shamir, A., "Weaknesses in the Key Scheduling Algorithm of RC4", *Selected Areas in Cryptography. SAC 2001. Lecture Notes in Computer Science*, **2259**, Springer, 2001, 1-24.