A BRIEF DISCUSSION ABOUT CONSTRUCTION OF CRYPTOGRAPHIC ALGORITHMS SALSA20 AND CHACHA

Boris Damjanović PhD[1], Nikola Novaković PhD [2], Negovan Stamenković PhD [3]

[1] Banja Luka College, Banja Luka, Republic of Srpska/BiH, boris.damanovic@blc.edu.ba

[2] Banja Luka College, Banja Luka, Republic of Srpska/BiH, nikola.novakovic@blc.edu.ba

[3] Banja Luka College, Banja Luka, Republic of Srpska/BiH, negovan.stamenkovic@blc.edu.ba

Abstract: Unlike block cryptographic algorithms, which encrypt data block by block, stream ciphers process data continuously, as pieces of plaintext arrive to them. Salsa20 is a family of stream ciphers designed in 2005 and presented at the eSTREAM competition for the best stream ciphers within the ECRYPT project. The ChaCha algorithm is very similar in construction to the Salsa algorithm, but in this algorithm the diffusion per round is increased and at the same time the performance is improved. These two algorithms are currently being implemented in a large number of cryptographic libraries and are becoming increasingly popular to use. In this text, the principles of functioning of the cryptographic algorithms Salsa20 and ChaCha will be presented.

Keywords: *Cryptography, stream ciphers, Salsa20, ChaCha.*

1. INTRODUCTION

Symmetric ciphers can be classified into two groups: block ciphers and stream ciphers [1]. Unlike block algorithms, which process data block by block, stream ciphers process data continuously, producing the ciphertext as the plaintext bytes arrive. Older stream ciphers are RC4, A5/1, A5/2 (whose use is no longer recommended), and newer Salsa, ChaCha20 ciphers [2].

Cryptographic algorithms, according to the method of construction and internal structure, can be divided into Substitution-permutation networks (SPNs) [3] [4], Feistel networks, Add-Rotate-XOR (ARX) algorithms, algorithms based on Feedback Shift Registers (NLFSRs) and Nonlinear Feedback Shift Registers (NLFSRs) and hybrid algorithms [5] [6].

An increasing number of cryptographic algorithms are created using ARX operations: 2n modulation, bit rotation and XOR operations (ARX - addition, rotation, XOR). Due to their very good software performance, ARX cryptographic algorithms are becoming more common. The most well-known cryptographic algorithms created using these principles are Salsa20 and ChaCha [7].

Stream ciphers often encrypt a series of bits (or a series of bytes), which cannot be considered data blocks [8]. In their work, they rely on the previously mentioned pseudo-random number generators (PRNG), where the open text is combined with the bytes and bits produced by the PRNG by using the for example XOR operation, to obtain the ciphertext. On the receiving side, it is necessary to generate an identical string of pseudo-random

numbers. The obtained data is then decrypted by combining the received stream (ciphertext) and the identical stream of pseudo-random numbers using the XOR operation to restore the text [2].
The Salsa20 cryptographic algorithm is a flow-through cryptographic algorithm developed by Daniel J. Bernstein [2]. Reduced versions of the Salsa20/12 and Salsa20/8 rounds have been proposed (using 12 and 8 rounds, respectively) [9], although the author himself admits that the security of the Salsa20/8 cryptographic algorithm is not great. Salsa20 is a family of stream ciphers designed in 2005 and entered on eSTREAM, the competition for the best stream cipher within the ECRYPT project [10]. As stated in [11], the Salsa20 algorithm is based on a Salsa20 hash function that takes a 64-byte input and produces a 64-byte output that has the same purpose as a counter in CTR mode. Salsa20 produces a cipher by creating a hash based on the key, the initialization vector (nonce) and the block number. The resulting hash is then merged with plain text using the XOR operation [2]. This algorithm represents the basis on which the cryptographic algorithm ChaCha is built. The ChaCha algorithm is very similar in construction to the Salsa algorithm, except in two points: the composition of its *quarterround* function (core function) and the composition of the initial matrix are different [12]. In this algorithm, the author increased the diffusion per round and at the same time improved the performance. Its slightly different core function was later used in the BLAKE hash function [1].

2. DESCRIPTION OF SALSA20 ALGORITHM FUNCTIONS

The Salsa20 algorithm consists of several functions used to construct its basic transformations. As stated in [11] and [13], these transformations, which are called core functions, are called *Quarter-Round (quarterround), Row-Round (rowround)* and *Column-Round (columnround)*. They are part of the Double-Round *(doubleround)* function. Within the algorithm, there is also a Little-Endian function, which is self-explanatory.
Salsa20 supports 128-bit and 256-bit keys. When we use 128-bit keys, the algorithm first duplicates the key to create a 256-bit key. Salsa20 operates on 32-bit words [7]. The input to the Salsa20 algorithm is a 512-bit string called State. This string is usually represented as a 4x4 matrix composed of 32-bit words [2]. The Salsa20 core function converts a 256-bit key (k_0, k_1, k_2, k_3, k_4, k_5, k_6, k_7), a 64-bit counter (p_0, p_1), a 64-bit nonce (v_0, v_1), and four 32- bit constants (c_0, c_1, c_2, c_3) into a 512-bit output [7].

Initial State

C_0	K_0	K_1	K_2
K_3	C_1	V_0	V_1
P_0	P_1	C_2	K_4
K_5	K_6	K_7	C_3

Figure 1: Input buffer State [2]

The constants are composed of ASCII codes of the English expression "expand 16-byte k" or "expand 32-byte k" as follows:
C0 = "expa",
C1 = "nd 3",
C2 = "2-by",
C3 = "te k".
These words will be used within the Salsa20 expansion function [2]. The basic permutation of the Salsa20 algorithm is a function called Quarter-Round (*quarterround*) [1]. Suppose that y is composed of four 32-bit words, that is, that y=(y_0, y_1, y_2, y_3). Then is the result of the function quarterround is:

$$quarterround(y) = (z_0, z_1, z_2, z_3) \quad (1)$$

where [1]

$$z_1 = y_1 \oplus [(y_0 + y_3) \lll 7]$$

$$z_2 = y_2 \oplus [(z_1 + y_0) \lll 9]$$

$$z_3 = y_3 \oplus [(z_2 + z_1) \lll 13]$$

$$z_0 = y_0 \oplus [(z_3 + z_2) \lll 18]$$

For example,

quarterround(0x00000001; 0x00000000; 0x00000000; 0x00000000) (2)

= (0x08008145; 0x00000080; 0x00010200; 0x20500000)

The Salsa20 algorithm transforms a 512-bit input buffer using the Row-Round (*rowround*) and Column-Round (*columnround*) operations that are part of the Double-Round *(doubleround)* function [2]. The Row-Round (*rowround*) function at the input takes a sequence of 16 words of 32 bits each, and at the output it gives a sequence of 16 words.
If y is given as a sequence composed of 16 words, each of which is 32 bits long:

$$y = (y_0, y_1, y_2, y_3, \ldots, y_{15}) \quad (3)$$

then the result of the function is: [11]

$$rowround(y) = (z_0, z_1, z_2, z_3, \ldots, z_{15}) \quad (4)$$

where:

$(z_0, z_1, z_2, z_3) = quarterround(y_0, y_1, y_2, y_3)$;

$(z_5, z_6, z_7, z_4) = quarterround(y_5, y_6, y_7, y_4)$;

$(z_{10}, z_{11}, z_8, z_9) = quarterround(y_{10}, y_{11}, y_8, y_9)$;

$(z_{15}, z_{12}, z_{13}, z_{14}) = quarterround(y_{15}, y_{12}, y_{13}, y_{14})$

The *rowround* function is executed during even rounds [14].
The *Column-Round (columnround)* function at the input takes a sequence of 16 words of 32 bits each, and at the output it gives a sequence of 16 words.
If x is given as a sequence of 16 words, each of which is 32 bits long [2]:

$$x = (x_0, x_1, x_2, x_3, \ldots, x_{15}) \quad (5)$$

then the result of the function is: [11]

$columnround(x) = (y_0, y_1, y_2, y_3, \ldots, y_{15})$ where:

(y_0, y_4, y_8, y_{12}); = $quarterround(x_0, x_4, x_8, x_{12})$;

$(y_5, y_9, y_{13}, y_1) = quarterround(x_5, x_9, x_{13}, x_1)$;

$(y_{10}, y_{14}, y_2, y_6) = quarterround(x_{10}, x_{14}, x_2, x_6)$;

$(y_{15}, y_3, y_7, y_{11}) = quarterround(x_{15}, x_3, x_7, x_{11})$;

An equivalent formula would be given with: [2]

$$(y_0, y_4, y_8, y_{12}, y_5, y_9, y_{13}, y_1, y_{10}, y_{14}, y_2, y_6, y_{15}, y_3, y_7, y_{11}) = \quad (6)$$

$$rowround(x_0, x_4, x_8, x_{12}, x_5, x_9, x_{13}, x_1, x_{10}, x_{14}, x_2, x_6, x_{15}, x_3, x_7, x_{11})$$

The *columnround* function is executed during odd rounds [14].
The *Double-round (doubleround)* function at the input takes a sequence of 16 words of 32 bits each, and at the output it gives a sequence of 16 words.
This function can be realized as a combination of the two previously described functions as follows: [11]

$$doubleround(x) = rowround(columnround(x)) \quad (7)$$

For the Salsa20 algorithm to function, the *Little-Endian (littleendian)* function is also important. For a given sequence b composed of 4 bytes,

$$b = (b_0, b_1, b_2, b_3) \quad (8)$$

the result of the function *littleendian(b)* is the word [11]:

$$littleendian(b) = b_0 + 2^8 b_1 + 2^{16} b_2 + 2^{24} b_3. \quad (9)$$

Salsa20 is performed during 20 rounds, ie 10 passes through the loop, and in each pass, odd and even rounds are processed.

3. BASIC TRANSFORMATIONS OF THE SALSA20 ALGORITHM

Salsa20 hash function (also *Salsa20 Core* function) [2] at the input it takes a sequence of 64 bytes, and at the output it also gives a sequence of bits with length of 64 bytes [11]

$$\text{Salsa20}(x) = x + \text{doubleround}^{10}(x) \tag{10}$$

where every 4 bytes from the sequence x are converted to a word in little-endian form. The exponent over the $\text{doubleround}^{10}(x)$ function refers to 10 iterations of the *doubleround()* function before realizing the addition [2].
The Salsa20 expansion function is used in the algorithm for every 64 bytes of data. The input to the *expansion* function consists of a input buffer State described by Figure 1.
If a key k of length 32 bytes is given and if an initialization vector (nonce) n is given as a sequence of 16 bytes, then $\text{Salsa20}_k(n)$ is a sequence of length 64 bytes.
For a 32-bit key, we need to define the previously mentioned string "*expand 32-byte k*" as

$$\sigma_0=(101, 120, 112, 97),\ \sigma_1=(110, 100, 32, 51),\ \sigma_2=(50, 45, 98, 121),\ \sigma_3=(116, 101, 32, 107) \tag{11}$$

If the k_0 and k_1 parts of the key are sequences of length of 16 bytes and if the initialization vector (nonce) n is a 16-byte sequence, then [11]

$$\text{Salsa20}_{k0,k1}(n) = \text{Salsa20}(\sigma_0, k_0, \sigma_1, n, \sigma_2, k_1, \sigma_3) \tag{12}$$

For a key k of length 16 bytes, we need to define the previously mentioned string "*expand 16-byte k*" as:

$$\tau_0=(101, 120, 112, 97),\ \tau_1=(110, 100, 32, 49),\ \tau_2=(54, 45, 98, 121),\ \tau_3=(116, 101, 32, 107) \tag{13}$$

If k is a 16-byte sequence and if the initialization vector (nonce) n is a 16-byte sequence, then the key is repeated as follows: [11]

$$\text{Salsa20}_k(n) = \text{Salsa20}(\tau_0, k, \tau_1, n, \tau_2, k, \tau_3) \tag{14}$$

The Salsa20 encryption function processes the message m using the key k and the initialization vector (nonce) n by using the XOR operation to mix the result of the Salsa20 expansion function with the message m: [2]

$$Salsa20_k(n) \oplus m \tag{15}$$

Let the message m be an l-bit sequence, where:

$$l \in \{0, 1, \dots, 2^{70}\} \tag{16}$$

Let the key k be a 32-byte or 16-byte sequence and let the initialization vector (nonce) v be given as an 8-byte sequence. The message m encryption function using the key k and the initialization vector n produces a cipher: [2]:

$$Salsa20_k(v) \oplus m \tag{17}$$

In the case of decryption, the ciphertext c is mixed with the Salsa20() stream using the XOR operation to obtain plaintext: [2]

$$Salsa20_k(v) \oplus c \tag{18}$$

4. CHACHA CRYPTOGRAPHIC ALGORITHM

During 2008, Bernstein introduced the ChaCha algorithm as a family of stream ciphers, which is a modification of the Salsa20 family. Within the ChaCha algorithm, the author increased the diffusion per round and at the same time improved the performance. The ChaCha algorithm follows the principles set out in the Salsa20 algorithm [15]. During the construction of this algorithm, the Quarter-Round function and the State buffer were modified [2].

Salsa20 transforms 4 words of 32-bit length each as follows [15]:

$$\begin{aligned} b &= b \oplus [(a + d) \lll 7] \\ c &= c \oplus [(b + a) \lll 9] \\ d &= d \oplus [(c + b) \lll 13] \end{aligned} \tag{19}$$

$$a = a \oplus [(d + c) \lll 18]$$

Like the Salsa algorithm, the ChaCha algorithm uses 4 additions, 4 XOR operations, and 4 rotations to transform 4 words of length by 32-bits each. However, the ChaCha algorithm transforms each word twice, as follows: [15]

(20)

$$a = a + b; d = d \oplus a; d = d \lll 16;$$

$$c = c + d; b = b \oplus c; b = b \lll 12;$$

$$a = a + b; d = d \oplus a; d = d \lll 8;$$

$$c = c + d; b = b \oplus c; b = b \lll 7;$$

The Salsa20 matrix consists of 4 words obtained by a combination of the nonce and the block counter (input, data known to the potential attacker), 8 key words and 4 constant words as follows [15]:

Table 1: Salsa20 initial matrix

constant	key	Key	key
key	constant	input	input
input	input	constant	key
key	key	key	constant

The ChaCha/n algorithm, like the Salsa20/n algorithm, creates a 4x4-word matrix that it then transforms during n rounds, and adds the result to the original matrix to get a 16-word (64-byte) output block. However, the ChaCha algorithm creates an initial matrix so that all the words known to the attacker are grouped at the bottom of the matrix [15]:

Table 2: ChaCha initial matrix

constant	constant	constant	constant
key	key	key	key
key	key	key	key
input	input	input	input

Unlike the Salsa20 algorithm, the ChaCha algorithm processes data in even and odd rounds in the same way. The ChaCha algorithm handles the State matrix in the same way during all rounds as follows [15]:

(21)

QUARTERROUND(x0, x4, x8,x12)

QUARTERROUND(x1, x5, x9,x13)

QUARTERROUND(x2, x6,x10,x14)

QUARTERROUND(x3, x7,x11,x15)

QUARTERROUND(x0, x5,x10,x15)

QUARTERROUND(x1, x6,x11,x12)

QUARTERROUND(x2, x7, x8,x13)

QUARTERROUND(x3, x4, x9,x14)

The ChaCha algorithm was the basis for creating the BLAKE hash function, which was a finalist in the NIST hash function selection competition. The ChaCha algorithm was also the basis of the idea for constructing the eXstended nonce (XChaCha) stream cipher. This algorithm allows ChaCha based ciphersuites to accept 192-bit nonce with a much lower probability of misuse.

5. CONCLUSION

The most used stream cipher until recently was RC4 used in the SSL (Secure Sockets Layer) protocol.
According to the recommendations of the IETF (Internet Engineering Task Force), Mozilla and Microsoft, it has been considered insecure since 2015. Therefore, the cryptographic community worked intensively on the development of support for TLS 1.3, which ensured the transition to the ChaCha20 algorithm [2].

Salsa20 is a simple, software-oriented algorithm optimized for execution on modern microprocessors. This algorithm, together with the very close ChaCha algorithm, has already been implemented in a number of libraries and protocols. Its designer Daniel J. Bernstein entered the Salsa20 algorithm into the eSTREAM competition. This algorithm is, due to its simplicity, very popular among developers [1].
The design of the Salsa20 cryptographic algorithm was very innovative at the time it appeared. Salsa20 is an algorithm with a rather original and flexible design, where very simple operations (modular arithmetic - addition, rotation and XOR) and lack of multiplication or S-box constructs are used to create an extremely fast cryptographic algorithm. This algorithm is the basis needed to get acquainted with the newer cryptographic algorithm ChaCha, which Bernstein introduced in 2008, within which the author increased diffusion per round and at the same time improved performance. Bernstein and his team implemented these algorithms within the NaCl cryptographic library (Networking and Cryptography library, pronounced as "salt").
The Google team began implementing algorithms ChaCha20 for symmetric encryption and Poly1305 for authentication in OpenSSL and NSS in March 2013 [16]. Support for ChaCha20 and Poly1305 has been added to OpenSSL 1.1.0 in 2016. The PHP programming language since version 7.2 includes the Sodium extension as its default cryptographic library. The libsodium implementation uses the Extended Salsa20 (XSalsa20) stream cipher with the Poly1305 authentication tag, while elliptic curve cryptography using the Curve25519 curve is used to exchange keys [17].
ChaCha20 typically provides good performance on systems where the CPU does not have hardware acceleration. As a result, the ChaCha20 algorithm is often the preferred algorithm in lower-power computing architectures, such as those found on smartphones. The ubiquity of smartphones and the rapid growth of IoT and similar devices based on such architectures, as well as the support of large software vendors, is a fact that will ensure the long-term use of these interesting algorithms.

REFERENCES

[1] J.-P. Aumasson, SERIOUS CRYPTOGRAPHY - A Practical Introduction to Modern Encryption, San Francisco: No Starch Press, Inc., 2018.

[2] B. Damjanović, Osnove kriptografije sa primjerima u programskom jeziku Java, Banja Luka: Besjeda, 2019.

[3] B. Damjanović i D. Simić, „Performance evaluation of aes algorithm under linux operating system,“ *Proceedings of the Romanian Academy - Series A: Mathematics, Physics, Technical Sciences, Information Science*, pp. 177-183, 2013 .

[4] B. Damjanović i D. Simić, „Tweakable parallel OFB mode of operation with delayed thread synchronization,“ *Security and Privacy,WIley*, t. 9, br. 10, pp. 1119-1131, 2015.

[5] G. Hatzivasilis, I. Papaefstathiou, K. Fysarakis i H. Manifavas, „A review of lightweight block ciphers,“ *Journal of Cryptographic Engineering*, t. 8, br. 2, pp. 141-184, 2017.

[6] Z. Eskandari, A. B. Kidmose, S. Kolbl i T. Tiessen, „Finding Integral Distinguishers with Ease,“ u *Selected Areas in Cryptography – SAC 2018 25th International Conference Calgary, AB, Canada, August 15–17, 2018 Revised Selected Papers (Carlos Cid • Michael J. Jacobson, Jr)*, Springer Nature Switzerland AG, 2018.

[7] N. Mouha i B. Preneel, „Towards finding optimal differential characteristics for ARX: Application to Salsa20,“ *Cryptology ePrint Archive, Report 2013/328*, 2013.

[8] S. Vaudenay, *A Classical Introduction To Cryptography - Applications for Communications Security*, Crissier, Switzerland: Springer Science+Business Media, Inc, 2006.

[9] D. J. Bernstein, "Salsa20/8 and Salsa20/12," [Online]. Available: https://cr.yp.to/snuffle/812.pdf. [Accessed 2020].

[10] D. J. Bernstein , „The Salsa20 family of stream ciphers,“ u *New Stream Cipher Designs. Lecture Notes in Computer Science, Robshaw M., Billet O. (eds)* , Berlin, Heidelberg, Springer.

[11] D. J. Bernstein, "Salsa20 specification, The eSTREAM Project - eSTREAM Phase 3," 2007. [Online]. Available: The eSTREAM Project - eSTREAM Phase 3. [Accessed 3 2019].

[12] I. Tsukasa, S. Kiyomoto i Y. Miyake, „Latin dances revisited: new analytic results of Salsa20 and ChaCha,“ u *International Conference on Information and Communications Security*, Springer, Berlin, Heidelberg, 2011.

[13] J. C. Hernandez-Castro, J. M. Tapiador i J.-J. Quisquater, „On the Salsa20 core function,“ *International Workshop on Fast Software Encryption, Springer, Berlin, Heidelberg*, 2008.

[14] J.-P. Aumasson, S. Fischer, S. Khazaei, W. Meier i C. Rechberger, „New Features of Latin Dances: Analysis of Salsa, ChaCha, and Rumba,“ u *International Workshop on Fast Software Encryption. Springer, Berlin, Heidelberg, 2008*, 2008.

[15] D. J. Bernstein, „ChaCha, a variant of Salsa20,“ u *Workshop Record of SASC 2008: The State of the Art of Stream Ciphers.*, 2008.

[16] S. Maitra, „Chosen IV cryptanalysis on reduced round ChaCha and Salsa,“ *Discrete Applied Mathematics*, t. 208, pp. 88-97, 2016.

[17] PARAGON INITIATIVE, "Using Libsodium in PHP Projects," [Online]. Available: https://paragonie.com/book/pecl-libsodium. [Accessed 11 11 2018].

[18] W. Stallings, Cryptography and Network Security: Principles and Practice, 7th Edition, Harlow, Essex, England: Pearson Education, 2017.

[19] K. R. Fall i R. Stevens, TCP/IP Illustrated, Volume 1, Pearson Education, Inc, 2012.