9.5.2.2 Salsa20

Salsa20[21] is an additive stream cipher—or rather a family of additive stream ciphers—that follows the design principle mentioned above. It was originally developed by Dan Bernstein and submitted to the eSTREAM project in 2005. Since then, it has been widely used in many Internet security protocols and applications. Its major advantages are simplicity and efficiency both in hardware or software implementations. In a typical software implementation, for example, the throughput of Salsa20 is about five times bigger than the throughput of RC4 that is already a fast stream cipher. As mentioned before, Salsa20 uses nonces, and it is therefore less important to periodically refresh the key. As of this writing, there are no published attacks against Salsa20/20 and Salsa20/12, i.e., the reduced-round version of Salsa20 with only 12 rounds. The best known attack is able to break Salsa20/8, i.e., the reduced-round version of Salsa20 with 8 rounds, but this attack is more theoretically interesting than practical.

Salsa20 operates on 64-byte (or 512-bit) blocks of data, meaning that a plaintext or ciphertext unit that is processed in one step is 64 bytes long.[22] Referring to formula (9.5), the encryption function of Salsa20 can be expressed as follows:

$$c = E_k(m,n) = Salsa20_k^{encrypt}(m,n) = m \oplus Salsa20_k^{expand}(n)$$

In this expression, $Salsa20^{encrypt}$ refers to the encryption function of Salsa20, whereas $Salsa20^{expand}$ refers to its expansion function. Both functions are keyed with k (that is typically 32 bytes long) and employ an 8-byte nonce n.[23] For the sake of simplicity, we don't distinguish between the Salsa20 encryption and expansion functions, and we use the term Salsa20 to refer to either of them (from the context it is almost always clear whether the encryption or expansion function is referred to). In addition to the Salsa20 encryption function and expansion functions, there is also a Salsa20 hash function that takes a 64-byte argument x and hashes it to a 64-byte output value $Salsa20(x)$. So the Salsa20 hash function does neither compress nor expand the argument, but it can still be thought of as being a cryptographic hash function, i.e., a function that has the same properties as a "normal" cryptographic hash function, such as pseudorandomness. We introduce the Salsa20 hash, expansion, and encryption functions in this order next.

21 https://cr.yp.to/salsa20.html.

22 In spite of this relatively large unit length, Salsa20 is still considered to be a stream cipher (and not a block cipher).

23 As explained below, $Salsa20_k^{expand}(n)$ refers to the iterated application of the Salsa20 expansion function. In each iteration, the 8-byte nonce n is concatenated with an 8-byte sequence number i. It is iterated as many times as required until a sufficiently long key stream is generated.

Hash Function

The Salsa20 hash function is word-oriented, meaning that it operates on words, referring to 4-byte or 32-bit values. It employs three basic operations on words:

- The addition modulo 2^{32} of two words w_1 and w_2, denoted as $w_1 + w_2$.
- The addition modulo 2 (XOR) of w_1 and w_2, denoted as $w_1 \oplus w_2$
- The c-bit left rotation of word w, denoted as $w \overset{\curvearrowleft}{\longleftarrow} c$ for some integer $c \geq 0$.[24]

For example, 0x77777777 + 0x01234567 = 0x789ABCDE, 0x01020304 $\oplus$ 0x789ABCDE = 0x7998BFDA, and 0x7998BFDA $\overset{\curvearrowleft}{\longleftarrow}$ 7 = 0111 1001 1001 1000 1011 1111 1101 1010 $\overset{\curvearrowleft}{\longleftarrow}$ 7 = 1100 1100 0101 1111 1110 1101 0011 1100 = 0xCC5FED3C. Mainly for performance reasons, the Salsa20 hash function only uses constant values for c (in the left rotation operation) and invokes neither word multiplications nor table lookups.

The Salsa20 hash function employs the three basic operations as building blocks in the following auxiliary functions:

- Let y be a 4-word value that consists of the 4 words y_0, y_1, y_2, and y_3, i.e., $y = (y_0, y_1, y_2, y_3)$. This means that y is $4 \cdot 32 = 128$ bits long. The quarterround function is defined as $quarterround(y) = z = (z_0, z_1, z_2, z_3)$, where

$$\begin{aligned} z_1 &= y_1 \oplus ((y_0 + y_3) \overset{\curvearrowleft}{\longleftarrow} 7) \\ z_2 &= y_2 \oplus ((z_1 + y_0) \overset{\curvearrowleft}{\longleftarrow} 9) \\ z_3 &= y_3 \oplus ((z_2 + z_1) \overset{\curvearrowleft}{\longleftarrow} 13) \\ z_0 &= y_0 \oplus ((z_3 + z_2) \overset{\curvearrowleft}{\longleftarrow} 18) \end{aligned}$$

  The quarterround function modifies the 4 words of y in place, i.e., y_1 is changed to z_1, y_2 is changed to z_2, y_3 is changed to z_3, and y_0 is finally changed to z_0. It is called "quarterround function," because it operates only on 4 words, whereas Salsa20 operates on 16 words (note that 4 is a quarter of 16).

24 Note that there is a subtle difference between a c-bit left rotation of word w, denoted as $w \overset{\curvearrowleft}{\longleftarrow} c$ (in this book), and a c-bit left shift of word w, denoted as $w \hookleftarrow c$. While the first operator ($\overset{\curvearrowleft}{\longleftarrow}$) means that the bits are rotated, meaning that the bits that fall out of the word on the left side are reinserted on the right side, this is not true for the second operator ($\hookleftarrow$). Here, zero bits are reinserted on the right side, and the bits that fall out of the word on the left side are lost. The same line of argumentation and notation apply to the c-bit right rotation of word w, denoted as $w \overset{\curvearrowright}{\longrightarrow} c$, and the c-bit right shift of w, denoted as $w \hookrightarrow c$.

- Let y be a 16-word value $(y_0, y_1, y_2, \ldots, y_{15})$ that can be represented as a square matrix

$$\begin{pmatrix} y_0 & y_1 & y_2 & y_3 \\ y_4 & y_5 & y_6 & y_7 \\ y_8 & y_9 & y_{10} & y_{11} \\ y_{12} & y_{13} & y_{14} & y_{15} \end{pmatrix}$$

  The rowround function takes y as input and modifies the rows of the matrix in parallel using the quarterround function mentioned above. More specifically, it generates a 16-word output value $z = rowround(y) = (z_0, z_1, z_2, \ldots, z_{15})$, where

$$\begin{aligned} (z_0, z_1, z_2, z_3) &= quarterround(y_0, y_1, y_2, y_3) \\ (z_5, z_6, z_7, z_4) &= quarterround(y_5, y_6, y_7, y_4) \\ (z_{10}, z_{11}, z_8, z_9) &= quarterround(y_{10}, y_{11}, y_8, y_9) \\ (z_{15}, z_{12}, z_{13}, z_{14}) &= quarterround(y_{15}, y_{12}, y_{13}, y_{14}) \end{aligned}$$

  This means that each row is processed individually (and independently from the other rows), and that the four words of each row are permuted in a specific way. In fact, the words of row i (for $1 \leq i \leq 4$) are rotated left for $i - 1$ positions. This means that the words of the first row are not permuted at all, the words of the second row are rotated left for one position, the words of the third row are rotated left for two positions, and the words of the fourth row are rotated left for three position before the quarterround function is applied.

- Similar to the rowround function, the columnround function takes a 16-word value $(y_0, y_1, y_2, \ldots, y_{15})$ and generates a 16-word output according to $z = columnround(y) = (z_0, z_1, z_2, \ldots, z_{15})$, where

$$\begin{aligned} (z_0, z_4, z_8, z_{12}) &= quarterround(y_0, y_4, y_8, y_{12}) \\ (z_5, z_9, z_{13}, z_1) &= quarterround(y_5, y_9, y_{13}, y_1) \\ (z_{10}, z_{14}, z_2, z_6) &= quarterround(y_{10}, y_{14}, y_2, y_6) \\ (z_{15}, z_3, z_7, z_{11}) &= quarterround(y_{15}, y_3, y_7, y_{11}) \end{aligned}$$

  The columnround function is somehow the transpose of the rowround function, i.e., it modifies the columns of the matrix in parallel by feeding a permutation of each column through the quarterround function.

- The rowround and columnround functions can be combined in a doubleround function. More specifically, the doubleround function is a columnround function followed by a rowround function. As such, it takes a 16-word sequence

as input and outputs another 16-word sequence. If $y = (y_0, y_1, y_2, \ldots, y_{15})$ is the input, then

$$\begin{aligned} z &= (z_0, z_1, z_2, \ldots, z_{15}) \\ &= doubleround(y) \\ &= rowround(columnround(y)) \end{aligned}$$

is the respective output. This means that the doubleround function first modifies the input's columns in parallel, and then modifies the rows in parallel. This, in turn, means that each word is modified twice.

- Finally, the littleendian function encodes a word or 4-byte sequence $b = (b_0, b_1, b_2, b_3)$ in little-endian order (b_3, b_2, b_1, b_0) that represents the value $b_3 \cdot 2^{24} + b_2 \cdot 2^{16} + b_1 \cdot 2^8 + b_0$. This value, in turn, is typically written in hexadecimal notation. For example, $littleendian(86, 75, 30, 9) = (9, 30, 75, 86)$ represents $9 \cdot 2^{24} + 30 \cdot 2^{16} + 75 \cdot 2^8 + 86$ that can be written as 0x091E4B56. Needless to say that the littlendian function can be inverted, so $littleendian^{-1}(0\text{x}091\text{E}4\text{B}56) = (86, 75, 30, 9)$.

Putting everything together, the Salsa20 hash function takes a 64-byte sequence $x = (x[0], x[1], \ldots, x[63])$ as input and generates another 64-byte sequence $Salsa20(x) = x + doubleround^{10}(x)$ as output. The input sequence x consists of 16 words in littleendian form:

$$\begin{aligned} x_0 &= littlendian(x[0], x[1], x[2], x[3]) \\ x_1 &= littlendian(x[4], x[5], x[6], x[7]) \\ &\ldots \\ x_{15} &= littlendian(x[60], x[61], x[62], x[63]) \end{aligned}$$

If $z = (z_0, z_1, z_2, \ldots, z_{15}) = doubleround^{10}(x_0, x_1, \ldots, x_{15})$, then the output of the hash function $Salsa20(x)$ is the concatenation of the 16 words that are generated as follows:

$$\begin{aligned} &littlendian^{-1}(z_0 + x_0) \\ &littlendian^{-1}(z_1 + x_1) \\ &\ldots \\ &littlendian^{-1}(z_{15} + x_{15}) \end{aligned}$$

The 20 rounds of Salsa20 come from the fact that the doubleround function is iterated 10 times, and each iteration basically represents two rounds, one standing for the columnround function and one standing for the rowround function.

Expansion Function

As its name suggests, the aim of the Salsa20 expansion function is to expand a 16-byte input n into a 64-byte output, using 32 or 16 bytes of keying material and 16 constant bytes. Depending on whether the keying material consists of 32 or 16 bytes, the constant bytes and the respective expansion functions are slightly different.

If the keying material consists of 32 bytes, then this material is split into two halves that represent two 16-bytes keys k_0 and k_1. In this case, the constant bytes look as follows (where each σ value consists of four bytes that are encoded using the littleendian function):

$$\begin{aligned}\sigma_0 &= (101, 120, 112, 97) = \text{0x61707865}\\ \sigma_1 &= (110, 100, 32, 51) = \text{0x3320646E}\\ \sigma_2 &= (50, 45, 98, 121) = \text{0x79622D32}\\ \sigma_3 &= (116, 101, 32, 107) = \text{0x6B206574}\end{aligned}$$

The Salsa20 expansion function is then defined as follows:

$$Salsa20_{k_0,k_1}(n) = Salsa20(\sigma_0, k_0, \sigma_1, n, \sigma_2, k_1, \sigma_3)$$

Note that $littleendian(\sigma_0) = littleendian(101, 120, 112, 97) = \text{0x61707865}$, so the argument that is subject to the Salsa20 hash function starts with the four bytes 0x61, 0x70, 0x78, and 0x65.

Otherwise, i.e., if the keying material consists of 16 bytes, then this material represents a single 16-byte key k that is applied twice. In this case, a slightly different set of 4-byte τ constants is used

$$\begin{aligned}\tau_0 &= (101, 120, 112, 97)\\ \tau_1 &= (110, 100, 32, \underline{49})\\ \tau_2 &= (\underline{54}, 45, 98, 121)\\ \tau_3 &= (116, 101, 32, 107)\end{aligned}$$

where the two bytes that are different from the respective σ constants are marked as underlined. In this case, the Salsa20 expansion function is defined as

$$Salsa20_k(n) = Salsa20(\tau_0, k, \tau_1, n, \tau_2, k, \tau_3).$$

In either case, σ and τ can be seen as constants c, k is the key, and n is the argument of the Salsa20 expansion function. Hence, the input to the function can be written in

a specific matrix layout:

$$\begin{pmatrix} c & k & k & k \\ k & c & n & n \\ n & n & c & k \\ k & k & k & c \end{pmatrix}$$

Obviously, this layout is somehow arbitrary and can be changed at will. As explained later, ChaCha20 is a variant of Salsa20 that uses a different matrix layout.

Encryption Function

Salsa20 is an additive stream cipher, meaning that an appropriately sized key stream is generated and added modulo 2 to the plaintext message. The Salsa20 expansion function is used to generate the key stream. More specifically, let k be a 32- or 16-byte key,[25] n an 8-byte nonce, and m an l-byte plaintext message that is going to be encrypted (where $0 \leq l \leq 2^{70}$). The Salsa20 encryption of m with nonce n under key k is denoted as $Salsa20_k(m, n)$. It is computed as $m \oplus Salsa20_k(n')$, where $Salsa20_k(n')$ represents a key stream that can be up to 2^{70} bytes long and n' is derived from n by adding a counter. Hence, the key stream is iteratively constructed as follows:

$$Salsa20_k(n, 0) \parallel Salsa20_k(n, 1) \parallel \ldots \parallel Salsa20_k(n, 2^{64} - 1)$$

In each iteration, the Salsa20 expansion function is keyed with k and applied to a 16-byte input that consists of the 8-byte nonce and an 8-byte counter i. If i is written bitwise, i.e., $i = (i_0, i_1, \ldots, i_7)$, then the respective counter is standing for $i_0 + 2^8 i_1 + 2^{16} i_2 + \ldots + 2^{56} i_7$. Each iteration of the Salsa20 expansion function yields $64 = 2^6$ bytes, so the maximal length of the key stream that can be generated this way is $2^{64} \cdot 2^6 = 2^{64+6} = 2^{70}$ bytes. It goes without saying that only as many bytes as necessary are generated to encrypt the l bytes of the message m. The bottom line is that the Salsa20 encryption function can be expressed as

$$c = (c[0], c[1], \ldots, c[l-1]) = (m[0], m[1], \ldots, m[l-1]) \oplus Salsa20_k(n')$$

or

$$c[i] = m[i] \oplus Salsa_k(n, \lfloor i/64 \rfloor)[i \bmod 64]$$

25 Consider the possibility of using a 16-byte key as an option. The preferred key size is 32 bytes referring to 256 bits.

for $i = 0, 1, \ldots, l-1$. Since Salsa20 is an additive stream cipher, the encryption and decryption functions are essentially the same (with m and c having opposite meanings).

Because the length of a nonce is controversially disucussed in the community, Bernstein proposed is variant of Salsa20 that can handle longer nonces. More specifically, *XSalsa20*[26] can take nonces that are 192 bits long (instead of 64 bits) without reducing the claimed security.

9.5.2.3 ChaCha20

In 2008, Bernstein proposed a modified version of the Salsa20 stream cipher named *ChaCha20* [10].[27] Again, the term refers to a family of stream ciphers that comprises ChaCha20/20 (20 rounds), ChaCha20/12 (12 rounds), and ChaCha20/8 (8 rounds). ChaCha20 is structurally identical to Salsa20, but it uses a different round function and a different matrix layout. Also, it uses a key that is always 32 bytes (256 bits) long, a nonce that is 12 bytes (96 bits) long, and a block counter that is only 4 bytes (32 bits) long. Remember that Salsa20 nonces and block counters are 8 bytes long each. Furthermore, the ChaCha20 specification also uses another notation to describe the quarterround function. Instead of using $y = (y_0, y_1, y_2, y_3)$ and $z = (z_0, z_1, z_2, z_3)$, it uses four 32-bit words a, b, c, and d. This means that

$$\begin{aligned} z_1 &= y_1 \oplus ((y_0 + y_3) \overset{\curvearrowright}{\leftarrow} 7) \\ z_2 &= y_2 \oplus ((z_1 + y_0) \overset{\curvearrowright}{\leftarrow} 9) \\ z_3 &= y_3 \oplus ((z_2 + z_1) \overset{\curvearrowright}{\leftarrow} 13) \\ z_0 &= y_0 \oplus ((z_3 + z_2) \overset{\curvearrowright}{\leftarrow} 18) \end{aligned}$$

can also be written as

$$\begin{aligned} b &= b \oplus ((a + d) \overset{\curvearrowright}{\leftarrow} 7) \\ c &= c \oplus ((b + a) \overset{\curvearrowright}{\leftarrow} 9) \\ d &= d \oplus ((c + b) \overset{\curvearrowright}{\leftarrow} 13) \\ a &= a \oplus ((d + c) \overset{\curvearrowright}{\leftarrow} 18) \end{aligned}$$

The operations performed by ChaCha20 are the same as the ones performed by Salsa20, but they are applied in a different order and each word is updated twice

26 https://cr.yp.to/snuffle/xsalsa-20081128.pdf.
27 https://cr.yp.to/chacha/chacha-20080128.pdf

instead of just once. The advantage is that the Chacha20 round function provides more diffusion than the Salsa20 round function. Also, the rotation distances are changed from 7, 9, 13, and 18 to 16, 12, 8, and 7, but this difference is less important. The ChaCha20 quarterround function updates a, b, c, and d as follows:

$$\begin{aligned}
&a = a + b;\ \ d = d \oplus a;\ \ d = d \overset{\curvearrowleft}{\leftarrow} 16;\\
&c = c + d;\ \ b = b \oplus c;\ \ b = b \overset{\curvearrowleft}{\leftarrow} 12;\\
&a = a + b;\ \ d = d \oplus a;\ \ d = d \overset{\curvearrowleft}{\leftarrow} 8;\\
&c = c + d;\ \ b = b \oplus c;\ \ b = b \overset{\curvearrowleft}{\leftarrow} 7;
\end{aligned}$$

Like Salsa20, ChaCha20 is an additive stream cipher, meaning that it pseudo-randomly generates a key stream that is then added modulo 2 to encrypt or decrypt a message. Hence, it is characterized by the PRG that is inherent in the design. The ChaCha20 PRG is overviewed in Algorithm 9.3. It operates on a (4×4)-matrix S of 4-byte words called the state. Hence, the state is exactly 64 bytes long. The ChaCha20 PRG takes as input a 32-byte key k, a 4-byte block counter i, and a 12-byte nonce n, and it generates as output a serialized version of the state. The respective bits are then added modulo 2 to the plaintext message (for encryption) or ciphertext (for decryption).

Algorithm 9.3 The ChaCha20 PRG algorithm.

(k, i, n)

$S = \sigma \parallel k \parallel i \parallel n$
$S' = S$
for $i = 1$ to 10 do begin
  $quarterround(S'_0, S'_4, S'_8, S'_{12})$
  $quarterround(S'_1, S'_5, S'_9, S'_{13})$
  $quarterround(S'_2, S'_6, S'_{10}, S'_{14})$
  $quarterround(S'_3, S'_7, S'_{11}, S'_{15})$
  $quarterround(S'_0, S'_5, S'_{10}, S'_{15})$
  $quarterround(S'_1, S'_6, S'_{11}, S'_{12})$
  $quarterround(S'_2, S'_7, S'_8, S'_{13})$
  $quarterround(S'_3, S'_4, S'_9, S'_{14})$
end
$S = S + S'$

Serialized(S)

In step one of the ChaCha20 PRG algorithm, S is constructed as the concatenation of the 4 constants σ_0, σ_1, σ_2, and σ_3 that are the same as the ones defined for Salsa20, k, i, and n. So the 48 bytes from k, i, and n are complemented with 16

constant bytes, and hence the total size of the state is 64 bytes. With regard to the matrix layout mentioned above, ChaCha20 uses the following (simplified) layout:

$$\begin{pmatrix} c & c & c & c \\ k & k & k & k \\ k & k & k & k \\ n & n & n & n \end{pmatrix}$$

The first row comprises the constants σ, the second and third rows comprise the key k, and the fourth row comprises the nonce n. To be precise, the nonce consists of a 4-byte block counter i and a 12-byte value n that represents the actual nonce.[28]

In step two of the ChaCha20 PRG algorithm, the state S is copied to the working state S'. This is the value that is processed iteratively in 10 rounds. In each round, the quarterround function is applied 8 times, where the first 4 applications refer to "column rounds" and the second 4 applications refer to "diagonal rounds" (remember that Salsa20 uses column and row rounds, but no diagonal rounds). Diagonal rounds are new in the ChaCha20 design, and they stand for themselves. 10 rounds with 8 applications of the quarterround function each yield $10 \cdot 8/4 = 20$ rounds. In the end, the original content of the state S is added modulo 2^{32} to S', and the result (in serialized form) refers to the 64-byte output of the ChaCha20 PRG algorithm. Serialization, in turn, is done by subjecting the words of S to the littleendian function and sequencing the resulting bytes. If, for example, the state begins with the two words 0x091E4B56 and 0xE4E7F110, then the output sequence begins with the 8 bytes 0x56, 0x4B, 0x1E, 0x09, 0x10, 0xF1, 0xE7, and 0xE4.

Similar to Salsa20, no practically relevant cryptanalytical attack against the ChaCha20 stream cipher is known to exist. It is therefore widely used on the Internet to replace RC4. Most importantly, it is often used with Bernstein's Poly1305 message authentication code (Section 10.3.3) to provide authenticated encryption.

9.6 BLOCK CIPHERS

As mentioned before, every practical symmetric encryption system processes plaintext messages unit by unit. In the case of a block cipher such a unit is called a *block*. Consequently, a block cipher maps plaintext message blocks of a specific length into ciphertext blocks of typically the same length, i.e., $\mathcal{M} = \mathcal{C} = \Sigma^n$ for some alphabet Σ and block length n (e.g., 128 bits).

In theory, a permutation on set S is just a bijective function $f : S \rightarrow S$ (Definition A.22). If we fix a block length n and work with the plaintext message

28 Note that the block counter i and the actual nonce n sum up to 16 bytes.