

# ChaCha20

Aaron Fihn, Prateep Rao, Nathaniel Webb, Shang Xiao

May 2, 2022

# Contents

<b>1</b>	<b>Algorithm Overview</b>	<b>3</b>
<b>2</b>	<b>Data Structures</b>	<b>3</b>
<b>3</b>	<b>Foundational Operations</b>	<b>4</b>
<b>4</b>	<b>Algorithm Steps</b>	<b>5</b>
4.1	Determine input size . . . . .	5
4.2	Create input blocks from key and nonce . . . . .	5
4.3	Transform each input block into an output block . . . . .	6
4.4	XOR the message with the keystream . . . . .	12
<b>5</b>	<b>Information about Technical Parts</b>	<b>12</b>
5.1	Implementation . . . . .	12
5.2	Visualization . . . . .	13
<b>6</b>	<b>How to Run Program</b>	<b>13</b>
<b>7</b>	<b>How to Format Program Input</b>	<b>14</b>
<b>8</b>	<b>Reflection on the Project</b>	<b>14</b>

# 1 Algorithm Overview

Salsa20 is an encryption algorithm that was designed by mathematician D.J. Bernstein, a professor at the University of Illinois at Chicago. It was created in 2005 with funding from the National Science Foundation and first published in 2007 [1]. In 2008, Bernstein published ChaCha20, a new cipher heavily based on Salsa20 that was designed to increase bit diffusion without sacrificing performance [2].

ChaCha20 is a symmetric encryption algorithm, meaning the same key is used to both encrypt and decrypt a message. It can encrypt messages of any size and requires a 32-byte key. The algorithm also uses a 12-byte nonce, or “number used once”, which must not be reused for encryption with the same key. Generally, the message sender requests a unique nonce from the recipient and then uses it along with the key to encode the message. This assures the message recipient that they are not receiving an older encrypted message from an attacker masquerading as the other party. (This is called a replay attack.)

ChaCha20 works by using the key and nonce to create a stream of bytes called the keystream. Encryption is performed by XORing the plaintext message with the keystream to create the ciphertext. Because XOR is a reversible operation, decryption works by XORing the keystream with the ciphertext to recreate the original plaintext.

The data structures, foundational operations, and basic structure of the algorithm are described below. Note that while Bernstein’s original version of ChaCha20 used an 8-byte nonce, most implementations we found instead relied on an IETF specification [3] that increased the nonce to 12 bytes. We implemented the IETF version of ChaCha20.

## 2 Data Structures

Along with streams of bytes, ChaCha20 uses two basic data structures. The first is a word, which represents an unsigned four-byte integer. Words are built from byte streams by assuming that the bytes are arranged in little-endian order. See below for an example. Because a word represents a single integer, we follow the common convention of writing the 8-digit hexadecimal number in big-endian order, which causes the byte order to appear swapped when compared to the byte stream.

Bytes	1	2	3	4
Text	L	o	g	s
Code Point (dec)	76	111	103	115
Code Point (hex)	4c	6f	67	73
Word (hex)	0x73676f4c			

Sequences of words are combined into data structures called blocks, which are visualized in the specification as a 4x4 grid of words. See below for an example:

1. Determine how many 64-byte blocks would be necessary to contain the plaintext message.
2. Create that many input blocks using the key and nonce.
3. Independently transform each input block into an output block to create the keystream.
4. Encrypt (or decrypt) the plaintext (or ciphertext) by XORing it with the keystream.

We describe these below along with an illustrative example. (The same example was used in the IETF specification, though not with the detail given below.) Assume we wish to perform an encryption using the following:

Block (with sample data)			
683f10f8	0b82181b	d4a63d06	5c3adffa
2bd9d0da	dc76133f	2293572c	4c78cd78
afd65851	805d2d44	36fb7da7	6bb8d439
a336d603	4dd4b7e0	adee697f	47a425b6

The 16 words in a block are described using indices from 0 to 15 as pictured below:

Block (with word indices)			
$w_0$	$w_1$	$w_2$	$w_3$
$w_4$	$w_5$	$w_6$	$w_7$
$w_8$	$w_9$	$w_{10}$	$w_{11}$
$w_{12}$	$w_{13}$	$w_{14}$	$w_{15}$

ChaCha20 uses the key and nonce to create a series of blocks, then permutes those blocks before unwinding them back into a series of bytes to create the keystream.

### 3 Foundational Operations

Three basic operations are used as a foundation for more complicated transformations. The first is addition between two words, which is always performed modulo  $2^{32}$  (meaning the sums “roll over” back to zero if they would be equal to or greater than  $2^{32}$ ).

$w_1$	$w_2$	$w_1 + w_2$
0xffffffffd	0x00000001	0xffffffffe
0xffffffffd	0x00000002	0xfffffffff
0xffffffffd	0x00000003	0x00000000
0xffffffffd	0x00000004	0x00000001

The second basic operation is a left-rotation, where the bits in a word are moved  $c$  bits to the left, and any bits that would “fall off” the left side are moved to the right. (We use the  $\lll$  operator to represent a left-rotation.)

$w$	$k$	$w \lll k$
0xc0a8787e	5	0x150f0fd8
0xc0a8787e	8	0xa8787e0a

The final basic operation is a simple XOR between two words.

$w_1$	$w_2$	$w_1 \wedge w_2$
0xff00ff00	0x00ff00ff	0xffffffff
0xff00ff00	0x00000000	0xff00ff00
0xff00ff00	0xffffffff	0x00ff00ff
0xc0a8787e	0x9fd1161d	0x5f796e63

## 4 Algorithm Steps

ChaCha20 can be described as a series of four steps:

1. Determine how many 64-byte blocks would be necessary to contain the plaintext message.
2. Create that many input blocks using the key and nonce.
3. Independently transform each input block into an output block to create the keystream.
4. Encrypt (or decrypt) the plaintext (or ciphertext) by XORing it with the keystream.

We describe these below along with an illustrative example. (The same example was used in the IETF specification, though not with the detail given below.) Assume we wish to perform an encryption using the following:

**Plaintext:**

“Ladies and Gentlemen of the class of ‘99: If I could offer you only one tip for the future, sunscreen would be it.”

**32-byte Key:**

03020100 07060504 0b0a0908 0f0e0d0c

13121110 17161514 1b1a1918 1f1e1d1c

**12-byte Nonce:**

00000000 4a000000 00000000

### 4.1 Determine input size

The plaintext message is 116 bytes long, so we would need two 64-byte blocks to contain it. Thus we must create two input blocks.

### 4.2 Create input blocks from key and nonce

Each input block is constructed from a 4x4 grid of words as follows. First, the arbitrary 16-byte sequence “**expand 32-byte k**” is transformed from bytes to words and used to fill the first row of each input block.

Character	e	x	p	a	n	d		3	2	-	b	y	t	e		k
Code Point (hex)	65	78	70	61	6e	64	20	33	32	2d	62	79	74	65	20	6b
Byte	61707865				3320646e				79622d32				6b206574			

Input Block			
61707865	3320646e	79622d32	6b206574

Next, the 32-byte key is inserted into the second and third rows of each input block.

Input Block			
61707865	3320646e	79622d32	6b206574
03020100	07060504	0b0a0908	0f0e0d0c
13121110	17161514	1b1a1918	1f1e1d1c

The lower-left word is a 4-byte block counter that starts at zero and counts up by one for each input block. This is the only word that initially differs between input blocks. (The IETF specification says that the block counter may optionally start from numbers other than zero.)

Input Block			
61707865	3320646e	79622d32	6b206574
03020100	07060504	0b0a0908	0f0e0d0c
13121110	17161514	1b1a1918	1f1e1d1c
00000000			

At this point in our example, we have two input blocks, which only differ by the block counter.

### 4.3 Transform each input block into an output block

This is the meat of the algorithm; each input block goes through 20 rounds of transformations before being added back to the original input block to become an output block of the keystream. In odd-numbered rounds, the columns of each block are independently transformed; in even-numbered rounds, the diagonals are independently transformed. To understand the transformations, we use the ground function, which was defined in the ChaCha20 specification.

The **ground** function takes a series of four words (labeled a, b, c, d) as an input and returns a series of four words as an output. During columnround, these inputs are the four words in a column from top to bottom; ground is called once for each column and the output is used to overwrite the original inputs. See the diagram below, where different colors of backgrounds represent different invocations of ground during a column round.

ground arguments, column round			
a	a	a	a
b	b	b	b
c	c	c	c
d	d	d	d

Since `ground_ChaCha` is an improved version of `ground_Salsa`, before analyzing `ground_ChaCha`, we should understand the algorithm principle of `ground_Salsa` at beginning. The main process of the algorithm is as follows:

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

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

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

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

The data unit of the algorithm is word, a word is composed of 4 bytes, that is, 32 bits, and its input and output are 4 words.  $\oplus$  means XOR,  $+$  means addition modulo  $2^{32}$ , and  $\lll$  means b-bit rotation of a 32-bit integer towards high bits. For computers, these operations are very easy to implement and very efficient. On the other hand, these seemingly simple operations combined can actually achieve the same level of security as any other operation option.

The first round of the algorithm is to add  $y_1$  and  $y_3$  modulo  $2^{32}$ , then rotate the result to the left by 7 bits, and finally XOR with  $y_1$  to get  $z_1$ . The next three operations are similar to the first round, but the initial word and the generated  $z$  are used for modulo addition. Also, the number of bits of the cyclic shift is also different. Algorithms are designed like this so that each bit in the plaintext affects many bits in the ciphertext, or each bit in the ciphertext is affected by many bit in the plaintext.

The data formats of `ground_ChaCha` and `ground_Salsa` are the same, and the input and output are also the same. Also, the basic operation of the algorithm is the same. However, ChaCha applies

the operations in a different order, and in particular updates each word twice rather than once. Specifically, ChaCha updates a, b, c, d as follows:

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

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

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

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

Although the basic operation looks similar, ground\_ChaCha is more diffusive and efficient due to the different iterations of each round. The first round iterative algorithm of the two is similar, and the effect is almost the same. The difference is only reflected in the number of bits to rotate left. However, due to the assignment operation, at the end of the first round, the four initial values a and d of a, b, c, d have been confused and changed. From this point of view, ground\_ChaCha seems to be twice as efficient as the previous algorithm.

An important indicator to measure a good cryptographic algorithm is diffusivity, that is, let each bit in the plaintext affect many bits in the ciphertext, or let each bit in the ciphertext be affected by many bits in the plaintext. Each assignment allows ground\_ChaCha to diffuse more thoroughly through more iterations. The time complexity of the algorithm is almost unchanged, but the space overhead is slightly increased. These are the promotions of ground\_ChaCha.

During diagonalround, the arguments are still labeled from top to bottom, but they come from upper-left diagonals instead of columns. Note that diagonals “wrap around” the sides of the block.

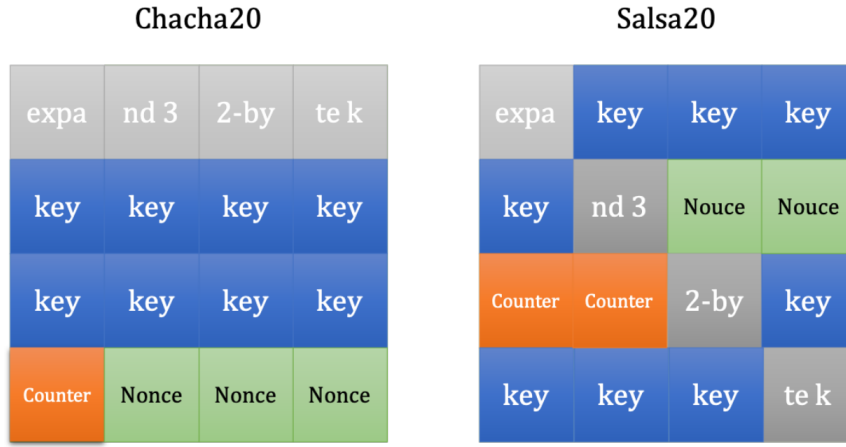
ground arguments, diagonal round			
a	a	a	a
b	b	b	b
c	c	c	c
d	d	d	d

The diagonalround function in ChaCha is similar to the "rowround" function in Salsa, but there are still differences between the two.

```
# chacha diagonal round
z[0], z[5], z[10], z[15] = ground_chacha([y[0], y[5], y[10], y[15]])
z[1], z[6], z[11], z[12] = ground_chacha([y[1], y[6], y[11], y[12]])
z[2], z[7], z[8], z[13] = ground_chacha([y[2], y[7], y[8], y[13]])
z[3], z[4], z[9], z[14] = ground_chacha([y[3], y[4], y[9], y[14]])

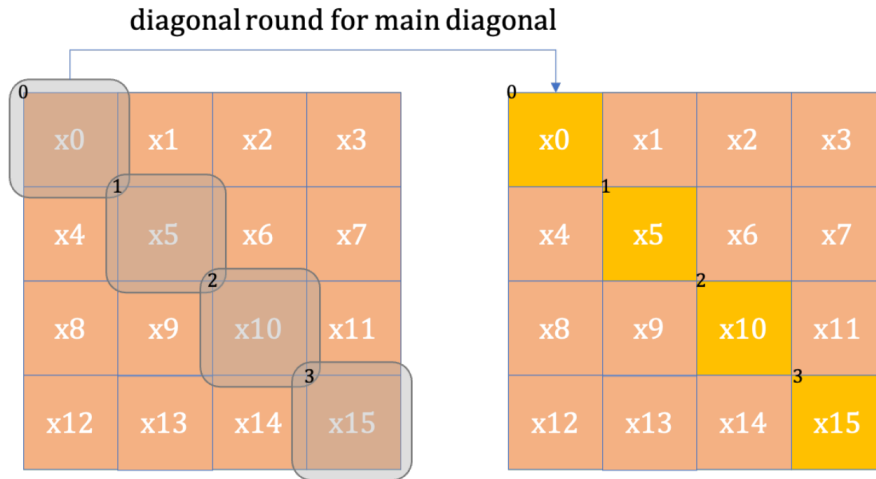
# salsa20 row round
z[0], z[1], z[2], z[3] = ground_salsa(y[0:4])
z[5], z[6], z[7], z[4] = ground_salsa(y[5:8], y[4:5])
z[10], z[11], z[8], z[9] = ground_salsa(y[10:12], y[8:10])
z[15], z[12], z[13], z[14] = ground_salsa(y[15:16], y[12:15])
```

One thing that must be shown is that the initial state of ChaCha is different from the one in Salsa20. In Salsa20, the initial state matrix is composed unorderedly while the ChaCha matrix puts all the stuff in order. As shown in the figure below, the gray block represents the constant, the blue block represents the key, the orange block represents the counter of message and the green block represents nonce which can be used only once.



Each block represents one Word

After knowing that, the diagonalround function has a similar effect to rowround which shuffle the rows of matrix. However, the diagonal transformation has both vertical and horizontal effects. Specifically, the function firstly transforms the main diagonal using the quarter-round which has been explained before. Then, apply the same function to lists (i.e. [1,6,11,12], [2,7,8,13] and [3,4,9,14]).



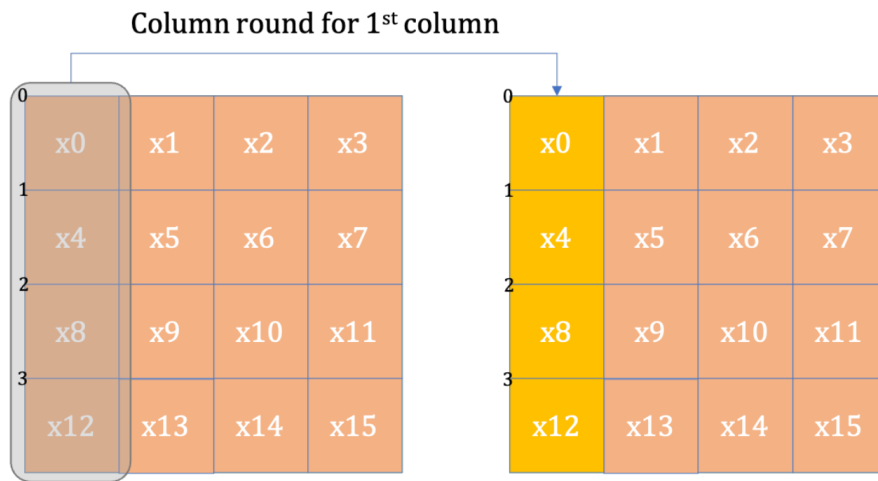
The columnround function of ChaCha20 has the same effect of columnround of Salsa20, which shuffle the rows in initial and intermediate state matrix. Specifically, the round function applies each row respectively.

```
# salsa20
y[ 0], y[ 4], y[ 8], y[12] = ground_salsa([x[ 0], x[ 4], x[ 8], x[12]])
y[ 5], y[ 9], y[13], y[ 1] = ground_salsa([x[ 5], x[ 9], x[13], x[ 1]])
y[10], y[14], y[ 2], y[ 6] = ground_salsa([x[10], x[14], x[ 2], x[ 6]])
y[15], y[ 3], y[ 7], y[11] = ground_salsa([x[15], x[ 3], x[ 7], x[11]])

# chacha20
y[0], y[4], y[8], y[12] = ground_chacha([x[0], x[4], x[8], x[12]])
y[1], y[5], y[9], y[13] = ground_chacha([x[1], x[5], x[9], x[13]])
y[2], y[6], y[10], y[14] = ground_chacha([x[2], x[6], x[10], x[14]])
y[3], y[7], y[11], y[15] = ground_chacha([x[3], x[7], x[11], x[15]])
```

As shown in the previous code segment, columnround in ChaCha feed the function in order while the one in Salsa20 does not. Shuffling the order is not beneficial for security and also handling the input in order can be speeded up in some machines. Therefore, the order shuffling was canceled for no good reasons.





The doubleround function is composed of two round functions (i.e. columnround and diagonalround), just determine the order of the two functions "columnround" and "diagonalround" which is as same as the one in Salsa20.

```
# chacha
diagonalround(columnround(x))
# salsa20
rowround(columnround(x))
```

The **qround** consists of the same twelve operations, regardless of whether used on a column or diagonal. The operations write over the original words to create the new output words.

1.  $a += b$
2.  $d \wedge= a$
3.  $d = d \lll 16$
4.  $c += d$
5.  $b \wedge= c$
6.  $b = b \lll 12$
7.  $a += b$
8.  $d \wedge= a$
9.  $d = d \lll 8$
10.  $c += d$
11.  $b \wedge= c$
12.  $b = b \lll 7$

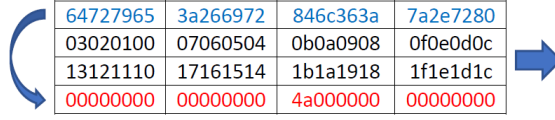
Note the circular order of the transformations:  $b$  modifies  $a$ , which modifies  $d$ , which modifies  $c$ , which modifies  $b$ , etc. These operations are used to thoroughly and irreversibly mix up the block's bits, giving a unique keystream. See below of an example of a column round, which is displayed as four qround operations happening in parallel.

Begin with the first input block found earlier and perform  $a = a + b$ :

61707865	3320646e	79622d32	6b206574	64727965	3a266972	846c363a	7a2e7280
03020100	07060504	0b0a0908	0f0e0d0c	03020100	07060504	0b0a0908	0f0e0d0c
13121110	17161514	1b1a1918	1f1e1d1c	13121110	17161514	1b1a1918	1f1e1d1c
00000000	00000000	4a000000	00000000	00000000	00000000	4a000000	00000000

Then  $\mathbf{d} = \mathbf{d} \wedge \mathbf{a}$  :


64727965	3a266972	846c363a	7a2e7280
03020100	07060504	0b0a0908	0f0e0d0c
13121110	17161514	1b1a1918	1f1e1d1c
00000000	00000000	4a000000	00000000



64727965	3a266972	846c363a	7a2e7280
03020100	07060504	0b0a0908	0f0e0d0c
13121110	17161514	1b1a1918	1f1e1d1c
<b>64727965</b>	<b>3a266972</b>	<b>ce6c363a</b>	<b>7a2e7280</b>

Then  $\mathbf{d} = \mathbf{d} <<< 16$  :

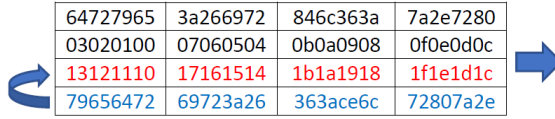
64727965	3a266972	846c363a	7a2e7280
03020100	07060504	0b0a0908	0f0e0d0c
13121110	17161514	1b1a1918	1f1e1d1c
64727965	3a266972	ce6c363a	7a2e7280



64727965	3a266972	846c363a	7a2e7280
03020100	07060504	0b0a0908	0f0e0d0c
13121110	17161514	1b1a1918	1f1e1d1c
<b>79656472</b>	<b>69723a26</b>	<b>363ace6c</b>	<b>72807a2e</b>

Then  $\mathbf{c} = \mathbf{c} + \mathbf{d}$  :

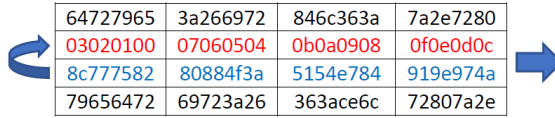
64727965	3a266972	846c363a	7a2e7280
03020100	07060504	0b0a0908	0f0e0d0c
13121110	17161514	1b1a1918	1f1e1d1c
79656472	69723a26	363ace6c	72807a2e



64727965	3a266972	846c363a	7a2e7280
03020100	07060504	0b0a0908	0f0e0d0c
<b>8c777582</b>	<b>80884f3a</b>	<b>5154e784</b>	<b>919e974a</b>
79656472	69723a26	363ace6c	72807a2e

Then  $\mathbf{b} = \mathbf{b} \wedge \mathbf{c}$  :


64727965	3a266972	846c363a	7a2e7280
03020100	07060504	0b0a0908	0f0e0d0c
8c777582	80884f3a	5154e784	919e974a
79656472	69723a26	363ace6c	72807a2e



64727965	3a266972	846c363a	7a2e7280
<b>8f757482</b>	<b>878e4a3e</b>	<b>5a5eee8c</b>	<b>9e909a46</b>
8c777582	80884f3a	5154e784	919e974a
79656472	69723a26	363ace6c	72807a2e

Then  $\mathbf{b} = \mathbf{b} <<< 12$  :

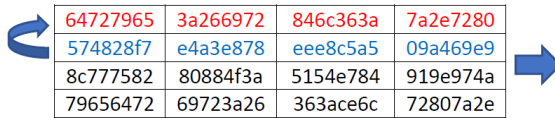
64727965	3a266972	846c363a	7a2e7280
8f757482	878e4a3e	5a5eee8c	9e909a46
8c777582	80884f3a	5154e784	919e974a
79656472	69723a26	363ace6c	72807a2e



64727965	3a266972	846c363a	7a2e7280
<b>574828f7</b>	<b>e4a3e878</b>	<b>eee8c5a5</b>	<b>09a469e9</b>
8c777582	80884f3a	5154e784	919e974a
79656472	69723a26	363ace6c	72807a2e

Then  $\mathbf{a} = \mathbf{a} + \mathbf{b}$  :

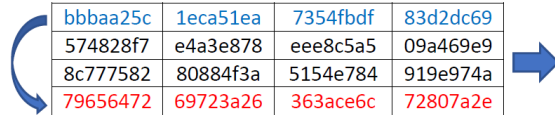
64727965	3a266972	846c363a	7a2e7280
574828f7	e4a3e878	eee8c5a5	09a469e9
8c777582	80884f3a	5154e784	919e974a
79656472	69723a26	363ace6c	72807a2e



<b>bbbaa25c</b>	<b>1eca51ea</b>	<b>7354fbdf</b>	<b>83d2dc69</b>
574828f7	e4a3e878	eee8c5a5	09a469e9
8c777582	80884f3a	5154e784	919e974a
79656472	69723a26	363ace6c	72807a2e

Then  $\mathbf{d} = \mathbf{d} \wedge \mathbf{a}$  :


bbbaa25c	1eca51ea	7354fbdf	83d2dc69
574828f7	e4a3e878	eee8c5a5	09a469e9
8c777582	80884f3a	5154e784	919e974a
79656472	69723a26	363ace6c	72807a2e



bbbaa25c	1eca51ea	7354fbdf	83d2dc69
574828f7	e4a3e878	eee8c5a5	09a469e9
8c777582	80884f3a	5154e784	919e974a
<b>c2dfc62e</b>	<b>77b86bcc</b>	<b>456e35b3</b>	<b>f152a647</b>

Then  $\mathbf{d} = \mathbf{d} <<< 8$  :

bbbaa25c	1eca51ea	7354fbdf	83d2dc69
574828f7	e4a3e878	eee8c5a5	09a469e9
8c777582	80884f3a	5154e784	919e974a
c2dfc62e	77b86bcc	456e35b3	f152a647



bbbaa25c	1eca51ea	7354fbdf	83d2dc69
574828f7	e4a3e878	eee8c5a5	09a469e9
8c777582	80884f3a	5154e784	919e974a
<b>dfc62ec2</b>	<b>b86bcc77</b>	<b>6e35b345</b>	<b>52a647f1</b>

Then  $\mathbf{c} = \mathbf{c} + \mathbf{d}$  :

bbbaa25c	1eca51ea	7354fbdf	83d2dc69
574828f7	e4a3e878	eee8c5a5	09a469e9
8c777582	80884f3a	5154e784	919e974a
dfc62ec2	b86bcc77	6e35b345	52a647f1

bbbaa25c	1eca51ea	7354fbdf	83d2dc69
574828f7	e4a3e878	eee8c5a5	09a469e9
6c3da444	38f41bb1	bf8a9ac9	e444df3b
dfc62ec2	b86bcc77	6e35b345	52a647f1

Then  $\mathbf{b} = \mathbf{b} \wedge \mathbf{c}$ :

bbbaa25c	1eca51ea	7354fbdf	83d2dc69
574828f7	e4a3e878	eee8c5a5	09a469e9
6c3da444	38f41bb1	bf8a9ac9	e444df3b
dfc62ec2	b86bcc77	6e35b345	52a647f1

bbbaa25c	1eca51ea	7354fbdf	83d2dc69
3b758cb3	dc57fc39	51625f6c	ede0b6d2
6c3da444	38f41bb1	bf8a9ac9	e444df3b
dfc62ec2	b86bcc77	6e35b345	52a647f1

And finally,  $\mathbf{b} = \mathbf{b} \lll 7$ :

bbbaa25c	1eca51ea	7354fbdf	83d2dc69
3b758cb3	dc57fc39	51625f6c	ede0b6d2
6c3da444	38f41bb1	bf8a9ac9	e444df3b
dfc62ec2	b86bcc77	6e35b345	52a647f1

bbbaa25c	1eca51ea	7354fbdf	83d2dc69
bac6599d	2bf9e4ee	b12fb628	f05b6976
6c3da444	38f41bb1	bf8a9ac9	e444df3b
dfc62ec2	b86bcc77	6e35b345	52a647f1

This constitutes a full column round. It is followed by a diagonal round, which uses the same 12 operations but on diagonals as described above. Then 18 more rounds are performed, alternating between columns and diagonals.

Once all 20 rounds are completed, each scrambled block is finally summed with its original input block. This creates the final output block.

Original input block			
61707865	3320646e	79622d32	6b206574
03020100	07060504	0b0a0908	0f0e0d0c
13121110	17161514	1b1a1918	1f1e1d1c
00000000	00000000	4a000000	00000000

After 20 rounds of scrambling			
dead8d4a	16153c4d	0738054f	43edaef6
27a057d2	b245c669	a8924dee	a0d0d5e3
69c66a73	8a44f68e	eb896808	3d94f193
d239a241	c874fc0d	c3567117	4b1e9c9c

Add both to create output block			
401e05af	4935a0bb	809a3281	af0e146a
2aa258d2	b94bcb6d	b39c56f6	afdee2ef
7cd87b83	a15b0ba2	06a38120	5cb30eaf
d239a241	c874fc0d	0d567117	4b1e9c9c

Once each input block has been transformed into an output block, the words can be converted back into a stream of bytes (again using little-endian encoding) to form the keystream.

First input block			
61707865	3320646e	79622d32	6b206574
03020100	07060504	0b0a0908	0f0e0d0c
13121110	17161514	1b1a1918	1f1e1d1c
00000000	00000000	4a000000	00000000

First output block			
401e05af	4935a0bb	809a3281	af0e146a
2aa258d2	b94bcb6d	b39c56f6	afdee2ef
7cd87b83	a15b0ba2	06a38120	5cb30eaf
d239a241	c874fc0d	0d567117	4b1e9c9c

Second input block			
61707865	3320646e	79622d32	6b206574
03020100	07060504	0b0a0908	0f0e0d0c
13121110	17161514	1b1a1918	1f1e1d1c
00000001	00000000	4a000000	00000000

Second output block			
f3514f22	e1d91b40	6f27de2f	ed1d63b8
821f138c	e2062c3d	ecca4f7e	78cff39e
a30a3b8a	920a6072	cd7479b5	34932bed
40ba4c79	cd343ec6	4c2c21ea	b7417df0

Keystream													
af	05	1e	40	bb	a0	35	49	81	32	9a	80	6a	...

## 4.4 XOR the message with the keystream

Finally, the plaintext message is encrypted by XORing it with the keystream to create the ciphertext. (If the keystream is longer than the message, then simply truncate the keystream beforehand.)

Keystream													
af	05	1e	40	bb	a0	35	49	81	32	9a	80	6a	...

^

Plaintext													
4c	61	64	69	65	73	20	61	6e	64	20	47	65	...

=

Ciphertext													
e3	64	7a	29	de	d3	15	28	ef	56	ba	c7	0f	...

To decrypt the ciphertext back into plaintext, the message recipient recreates the keystream using the same key and nonce, then XORs it into the ciphertext.

## 5 Information about Technical Parts

### 5.1 Implementation

Our ChaCha20 implementation was programmed collaboratively on GitHub [4] using Python, the language that most of our group members were familiar with. We first implemented Salsa20; both algorithms are very similar and the Salsa20 specification included more example outputs for various parts of the code, giving us more confidence that they were correct. The Salsa20 specification handily broke functions down into the following:

- word addition
- word XOR
- word left-rotate
- word addition
- quarterround (a.k.a. **qround**)
- rowround (an application of qround to each row)
- columnround (an application of qround to each column)
- a hash function (converts 64 bytes representing an input block into an output block)
- an expansion function (converts the key and nonce into a keystream given a block number)
- an encryption/decryption function (uses the message and expansion function to generate ciphertext/plaintext)

We kept our code largely in the same form so that unit tests could be created using the specification's examples. Salsa20 code is included in code repository at `/Salsa20/Salsa.py`, while unit tests are `/tests.py`. As the specification does not include an example of encryption from start to finish, the code was validated against the Salsa20 package in the PyPI repository [5].

Salsa20 was then converted to the IETF version of ChaCha20, which required the following:

- Rearranging the words of the input blocks
- Changing the internal operations of the qround function
- Changing qround's calling code to operate on diagonals instead of rows during even-numbered rounds

- Changing from an 8-byte block counter to a 4-byte block counter
- Changing from an 8-byte nonce to a 12-byte nonce
- Adding code to allow the block counter to optionally begin at a number other than zero

The ChaCha20 code can be found in `/Salsa20/ChaCha.py`, with unit tests in `/Salsa20/ChaCha-tests.py`. The IETF specification included examples that were converted into successfully passing unit tests.

## 5.2 Visualization

To visualize the algorithm, we wrote a visualizer class, found in `/Salsa20/ChaChaviz.py`. It utilizes `graphics.py`, a lightweight graphics library written by Dr. John Zelle of Wartburg College [6], to draw to a window on the screen.

When running the program (see below) the user can choose whether or not to visualize the operations of the algorithm. If they choose to do so, they then enter a refresh speed, which is used to determine how quickly the visualizer draws each step. This option is given because ChaCha20 performs a large number of operations, so with visualization on, the algorithm can take quite a while to run.

The visualizer class has a single public method, `ground()`. The algorithm passes this method the current state of the block, the indices of the four words involved in the current round, and some display information (the double-round count, whether we're performing the column or diagonal of the current double-round, and the quarter-round count within the column or diagonal).

The visualizer then performs the current quarter round, drawing each step to the screen in two parts: first, it shows the operation (add, XOR, or rotate). It highlights the word to be changed in red, and in the case of add and XOR, highlights the other word in green. Then the visualizer shows the new state of the block (with the operation applied), with the changed word highlighted in blue.

In short, all the algorithm need do is call the `ground()` method of the visualizer each time it performs a quarter-round. For purposes of code cleanliness, we created a separate file for the ChaCha20 algorithm with visualization integration. This file is at `/Salsa20/ChaCha_with_viz.py`. It is identical to `ChaCha.py`, but includes visualizer calls.

## 6 How to Run Program

**IMPORTANT:** the program will not run in IDLE, or any environment running Python 3.8 or earlier. The code relies on typing and import syntax that was introduced in Python 3.9. Our team developed and tested the code in pycharm.

**Step 1:** Set your secret key and nonce. These are given in the files `ChaCha_secret.txt` and `ChaCha_nonce.txt` respectively. The secret can be any string of 32 characters. The nonce can be any string of 12 characters. Remember, you can use the same secret every time, but for maximum security, you should generate a new nonce for every message.

**Step 2:** To launch the program, run the file `/Salsa20/ChaCha_app.py`. This runs a CLI allowing you to encrypt and decrypt using ChaCha20.

**Step 3:** If you wish to run the visualizer, answer "yes" to the first prompt. At the second prompt, enter a refresh speed for the visualizer. Lower values run faster. (Note: you will not be asked for a refresh speed if you aren't running the visualizer.)

**Step 4:** Now you are prompted to enter text to encrypt using ChaCha20. Enter any string you wish.

**Step 5:** The algorithm runs (with visualization if desired) and prints the result to the console.

**Step 6:** At the encrypt/decrypt prompt, enter "quit" to finish running the program.

**NOTE ON STEPS 4-5:** ChaCha20 encrypts and decrypts identically, so you could theoretically use this CLI to encrypt and decrypt messages. However, when Python prints encryption

results to the console, it must interpret the bytes object. Any byte that cannot be interpreted as an ASCII character is escaped using the pattern 00. If you copy this printout and input it to the CLI, each character is interpreted separately, not as part of an escape. Thus the algorithm will not decrypt to the original string.

For demonstration purposes, the app automatically decrypts the encryption result and prints the decrypted string as well. This string should match your original input (with the exception of being a bytes object). To observe round-trip encryption and decryption, please see also the unit test suite.

## 7 How to Format Program Input

Since ChaCha20 operates on plaintext strings, there are no special formatting rules when inputting data to encrypt.

## 8 Reflection on the Project

We had several advantages that we made good use of in this project. Our team had a diverse skill set (Python, LaTeX, PowerPoint, etc.) that let us have different members focus on different parts of the project. The original Salsa20 specification helpfully broke the algorithm into chunks that could be implemented in stages and gave example outputs for each part; we used these in unit tests to build confidence in our results. We were able to make consistent progress and were confident we could hit the deadline.

Not all went easily. Bernstein's ChaCha20 specification is somewhat vague in parts, forcing us to spend time hunting for other papers or implementations to help understand it. Some of our tasks were hard to work on in parallel; for example, most of the visualization work couldn't begin until the base ChaCha20 algorithm was largely completed. This meant all of us effectively took a turn sitting idle. And we probably underestimated the time required in drafting the report and presentation, which took roughly as long as implementing the algorithm itself. If we could have done anything differently, we probably should have assigned a single person to work on the report/presentation from day one, or swapped the responsibility during idle times over the past few weeks. More upfront effort in designing an API might have allowed us to work on the visualization and base algorithm in tandem, and would have let us build unit tests early enough to avoid some mistakes.

## References

- [1] Daniel J. Bernstein. The salsa20 family of stream ciphers. 2007. Department of Mathematics, Statistics, and Computer Science (M/C 249). The University of Illinois at Chicago. Chicago, IL 60607-7045.
- [2] Daniel J. Bernstein. Chacha, a variant of salsa20. 2007. Department of Mathematics, Statistics, and Computer Science (M/C 249). The University of Illinois at Chicago. Chicago, IL 60607-7045.
- [3] Yoav Nir and Adam Langley. Chacha20 and poly1305 for ietf protocols. May 2015. Internet Research Task Force (IRTF). ISSN: 2070-1721. Informational.
- [4] Aaron Fihn, Prateep Malyala, Shang Xiao, and Nathaniel Webb. Chacha20 algorithm project for cs5800. GitHub, May 2022. URL: <https://github.com/nwebb-roux/5800-chacha>.
- [5] PyPI. *salsa20 0.3.0*, 0.3.0 edition, November 2013. Maintainers: Filippo Valsorda and Maxtaco.
- [6] John M. Zelle. Professor of computer science, phd. Wartburg College.