ChaCha20 Report

Aaron Fihn, Prateep Rao, Nathaniel Webb, Shang Xiao, 5/5/2022, CS 5800

# 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]](#endnote-1) In 2008, Bernstein published ChaCha20, a new cipher heavily based on Salsa20 that was designed to increase bit diffusion without sacrificing performance.[[2]](#endnote-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 be 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]](#endnote-3) that increased the nonce to 12 bytes. We implemented the IETF version of ChaCha20.

## 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.



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.



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



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.

## 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 232 (meaning the sums “roll over” back to zero if they would be equal to or greater than 232).



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 <<< operator to represent a left-rotation.)



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



## 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

### 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.

### 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 |  |  |  |

Finally, the last three words are filled using the 12-byte nonce.

|  |  |  |  |
| --- | --- | --- | --- |
| Input Block | | | |
| 61707865 | 3320646e | 79622d32 | 6b206574 |
| 03020100 | 07060504 | 0b0a0908 | 0f0e0d0c |
| 13121110 | 17161514 | 1b1a1918 | 1f1e1d1c |
| 00000000 | 00000000 | 4a000000 | 00000000 |

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

### 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 qround function, which was defined in the ChaCha20 specification.

The qround 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 column rounds, these inputs are the four words in a column from top to bottom; qround 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 qround during a column round.

|  |  |  |  |
| --- | --- | --- | --- |
| qround arguments, column round | | | |
| a | a | a | a |
| b | b | b | b |
| c | c | c | c |
| d | d | d | d |

During diagonal rounds, 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.

|  |  |  |  |
| --- | --- | --- | --- |
| qround arguments, diagonal round | | | |
| a | a | a | a |
| b | b | b | b |
| c | c | c | c |
| d | d | d | d |

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 ^= a
3. d = d <<< 16
4. c += d
5. b ^= c
6. b = b <<< 12
7. a += b
8. d ^= a
9. d = d <<< 8
10. c += d
11. b ^= c
12. b = b <<< 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 **d** = d ^ a:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 64727965 | 3a266972 | 846c363a | 7a2e7280 |  | 64727965 | 3a266972 | 846c363a | 7a2e7280 |
| 03020100 | 07060504 | 0b0a0908 | 0f0e0d0c |  | 03020100 | 07060504 | 0b0a0908 | 0f0e0d0c |
| 13121110 | 17161514 | 1b1a1918 | 1f1e1d1c |  | 13121110 | 17161514 | 1b1a1918 | 1f1e1d1c |
| 00000000 | 00000000 | 4a000000 | 00000000 |  | **64727965** | **3a266972** | **ce6c363a** | **7a2e7280** |

Then **d** = d <<< 16:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 64727965 | 3a266972 | 846c363a | 7a2e7280 |  | 64727965 | 3a266972 | 846c363a | 7a2e7280 |
| 03020100 | 07060504 | 0b0a0908 | 0f0e0d0c |  | 03020100 | 07060504 | 0b0a0908 | 0f0e0d0c |
| 13121110 | 17161514 | 1b1a1918 | 1f1e1d1c |  | 13121110 | 17161514 | 1b1a1918 | 1f1e1d1c |
| 64727965 | 3a266972 | ce6c363a | 7a2e7280 |  | **79656472** | **69723a26** | **363ace6c** | **72807a2e** |

Then **c** = c + d:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 64727965 | 3a266972 | 846c363a | 7a2e7280 |  | 64727965 | 3a266972 | 846c363a | 7a2e7280 |
| 03020100 | 07060504 | 0b0a0908 | 0f0e0d0c |  | 03020100 | 07060504 | 0b0a0908 | 0f0e0d0c |
| 13121110 | 17161514 | 1b1a1918 | 1f1e1d1c |  | **8c777582** | **80884f3a** | **5154e784** | **919e974a** |
| 79656472 | 69723a26 | 363ace6c | 72807a2e |  | 79656472 | 69723a26 | 363ace6c | 72807a2e |

Then **b** = b ^ c:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 64727965 | 3a266972 | 846c363a | 7a2e7280 |  | 64727965 | 3a266972 | 846c363a | 7a2e7280 |
| 03020100 | 07060504 | 0b0a0908 | 0f0e0d0c |  | **8f757482** | **878e4a3e** | **5a5eee8c** | **9e909a46** |
| 8c777582 | 80884f3a | 5154e784 | 919e974a |  | 8c777582 | 80884f3a | 5154e784 | 919e974a |
| 79656472 | 69723a26 | 363ace6c | 72807a2e |  | 79656472 | 69723a26 | 363ace6c | 72807a2e |

Then **b** = b <<< 12:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 64727965 | 3a266972 | 846c363a | 7a2e7280 |  | 64727965 | 3a266972 | 846c363a | 7a2e7280 |
| 8f757482 | 878e4a3e | 5a5eee8c | 9e909a46 |  | **574828f7** | **e4a3e878** | **eee8c5a5** | **09a469e9** |
| 8c777582 | 80884f3a | 5154e784 | 919e974a |  | 8c777582 | 80884f3a | 5154e784 | 919e974a |
| 79656472 | 69723a26 | 363ace6c | 72807a2e |  | 79656472 | 69723a26 | 363ace6c | 72807a2e |

Then **a** = a + b:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 64727965 | 3a266972 | 846c363a | 7a2e7280 |  | **bbbaa25c** | **1eca51ea** | **7354fbdf** | **83d2dc69** |
| 574828f7 | e4a3e878 | eee8c5a5 | 09a469e9 |  | 574828f7 | e4a3e878 | eee8c5a5 | 09a469e9 |
| 8c777582 | 80884f3a | 5154e784 | 919e974a |  | 8c777582 | 80884f3a | 5154e784 | 919e974a |
| 79656472 | 69723a26 | 363ace6c | 72807a2e |  | 79656472 | 69723a26 | 363ace6c | 72807a2e |

Then **d** = d ^ a:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| bbbaa25c | 1eca51ea | 7354fbdf | 83d2dc69 |  | bbbaa25c | 1eca51ea | 7354fbdf | 83d2dc69 |
| 574828f7 | e4a3e878 | eee8c5a5 | 09a469e9 |  | 574828f7 | e4a3e878 | eee8c5a5 | 09a469e9 |
| 8c777582 | 80884f3a | 5154e784 | 919e974a |  | 8c777582 | 80884f3a | 5154e784 | 919e974a |
| 79656472 | 69723a26 | 363ace6c | 72807a2e |  | **c2dfc62e** | **77b86bcc** | **456e35b3** | **f152a647** |

Then **d** = d <<< 8:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| bbbaa25c | 1eca51ea | 7354fbdf | 83d2dc69 |  | bbbaa25c | 1eca51ea | 7354fbdf | 83d2dc69 |
| 574828f7 | e4a3e878 | eee8c5a5 | 09a469e9 |  | 574828f7 | e4a3e878 | eee8c5a5 | 09a469e9 |
| 8c777582 | 80884f3a | 5154e784 | 919e974a |  | 8c777582 | 80884f3a | 5154e784 | 919e974a |
| c2dfc62e | 77b86bcc | 456e35b3 | f152a647 |  | **dfc62ec2** | **b86bcc77** | **6e35b345** | **52a647f1** |

Then **c** = c + d:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| bbbaa25c | 1eca51ea | 7354fbdf | 83d2dc69 |  | bbbaa25c | 1eca51ea | 7354fbdf | 83d2dc69 |
| 574828f7 | e4a3e878 | eee8c5a5 | 09a469e9 |  | 574828f7 | e4a3e878 | eee8c5a5 | 09a469e9 |
| 8c777582 | 80884f3a | 5154e784 | 919e974a |  | **6c3da444** | **38f41bb1** | **bf8a9ac9** | **e444df3b** |
| dfc62ec2 | b86bcc77 | 6e35b345 | 52a647f1 |  | dfc62ec2 | b86bcc77 | 6e35b345 | 52a647f1 |

Then **b** = b ^ c:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| bbbaa25c | 1eca51ea | 7354fbdf | 83d2dc69 |  | bbbaa25c | 1eca51ea | 7354fbdf | 83d2dc69 |
| 574828f7 | e4a3e878 | eee8c5a5 | 09a469e9 |  | **3b758cb3** | **dc57fc39** | **51625f6c** | **ede0b6d2** |
| 6c3da444 | 38f41bb1 | bf8a9ac9 | e444df3b |  | 6c3da444 | 38f41bb1 | bf8a9ac9 | e444df3b |
| dfc62ec2 | b86bcc77 | 6e35b345 | 52a647f1 |  | dfc62ec2 | b86bcc77 | 6e35b345 | 52a647f1 |

And finally, **b** = b <<< 7:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| bbbaa25c | 1eca51ea | 7354fbdf | 83d2dc69 |  | bbbaa25c | 1eca51ea | 7354fbdf | 83d2dc69 |
| 3b758cb3 | dc57fc39 | 51625f6c | ede0b6d2 |  | **bac6599d** | **2bf9e4ee** | **b12fb628** | **f05b6976** |
| 6c3da444 | 38f41bb1 | bf8a9ac9 | e444df3b |  | 6c3da444 | 38f41bb1 | bf8a9ac9 | e444df3b |
| dfc62ec2 | b86bcc77 | 6e35b345 | 52a647f1 |  | 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 | | | |  | After 20 rounds of scrambling | | | |
| 61707865 | 3320646e | 79622d32 | 6b206574 |  | dead8d4a | 16153c4d | 0738054f | 43edaef6 |
| 03020100 | 07060504 | 0b0a0908 | 0f0e0d0c |  | 27a057d2 | b245c669 | a8924dee | a0d0d5e3 |
| 13121110 | 17161514 | 1b1a1918 | 1f1e1d1c |  | 69c66a73 | 8a44f68e | eb896808 | 3d94f193 |
| 00000000 | 00000000 | 4a000000 | 00000000 |  | 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 | | | |  | Second input block | | | |
| 61707865 | 3320646e | 79622d32 | 6b206574 |  | 61707865 | 3320646e | 79622d32 | 6b206574 |
| 03020100 | 07060504 | 0b0a0908 | 0f0e0d0c |  | 03020100 | 07060504 | 0b0a0908 | 0f0e0d0c |
| 13121110 | 17161514 | 1b1a1918 | 1f1e1d1c |  | 13121110 | 17161514 | 1b1a1918 | 1f1e1d1c |
| 00000000 | 00000000 | 4a000000 | 00000000 |  | 00000001 | 00000000 | 4a000000 | 00000000 |
|  |  |  |  |  |  |  |  |  |
| First output block | | | |  | Second output block | | | |
| 401e05af | 4935a0bb | 809a3281 | af0e146a |  | f3514f22 | e1d91b40 | 6f27de2f | ed1d63b8 |
| 2aa258d2 | b94bcb6d | b39c56f6 | afdee2ef |  | 821f138c | e2062c3d | ecca4f7e | 78cff39e |
| 7cd87b83 | a15b0ba2 | 06a38120 | 5cb30eaf |  | a30a3b8a | 920a6072 | cd7479b5 | 34932bed |
| d239a241 | c874fc0d | 0d567117 | 4b1e9c9c |  | 40ba4c79 | cd343ec6 | 4c2c21ea | b7417df0 |

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

### 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.

# Information about Technical parts

Our ChaCha20 implementation was programmed collaboratively on GitHub[[4]](#endnote-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
* 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]](#endnote-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.

[Add more about the visualization and the program launcher]

# How to Run Program

[Add this entire section]

# How to Format Program Input

[Add this entire section]

# 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.

# Citations

[I just pasted some links for endnotes; these need to be converted to a proper citation format.]

1. https://cr.yp.to/snuffle/salsafamily-20071225.pdf [↑](#endnote-ref-1)
2. https://cr.yp.to/chacha/chacha-20080128.pdf [↑](#endnote-ref-2)
3. https://datatracker.ietf.org/doc/html/rfc7539 [↑](#endnote-ref-3)
4. https://github.com/nwebb-roux/5800-chacha [↑](#endnote-ref-4)
5. https://pypi.org/project/salsa20/ [↑](#endnote-ref-5)