[FALL 2025 CS485/585]
Project report: Salsa20

Nathan Metens and Nick Whiteman

Department of Computer Science
Portland State University

{*metens,nicholaw*}*@pdx.edu*

December 11, 2025

Abstract

We set out to understand the Salsa20 stream cipher by building it from scratch, running it, observing its behavior, and creating a visual program to demonstrate the process in detail. We wrote our own Python implementation based on the official specification and broke the algorithm into small, clear functions so we could see exactly what was happening in each round. From there, we created a menu-driven program that lets users encrypt and decrypt messages, generate random keys and nonces, view the history of past runs, and even print a full trace of all 20 rounds of state mixing. This trace shows how the words in the matrix change during each column and row round and how the final keystream is produced before it is XORed with the plaintext to obtain the ciphertext.

To ensure our code was correct, we tested parts of it with PyTest and compared our results against the example values published by Daniel Bernstein in his papers. We also examined the security of Salsa20 and found that although reduced-round versions are vulnerable to attack, the standard 20-round version remains secure. Lastly, we compared Salsa20 to ChaCha20. Chacha20—an upgraded version of Salsa20—is based on the same principles but enables faster word diffusion and is now used in many modern protocols, including TLS and WireGuard. Overall, building Salsa20 from the ground up helped us understand how stream ciphers work in practice, how ARX-based designs achieve speed and security, and why implementation details matter.

1 Introduction

Encryption is everywhere, but most people never see it. When we open a website, connect to a VPN, or send information over Wi-Fi, cryptography silently works in the background to keep everything private and secure. Stream ciphers are vital in these situations because they are extremely fast and can handle continuous data. One cipher that fits this role very well is Salsa20, designed by Daniel J. Bernstein in 2005. Instead of relying on complicated mathematical structures, Salsa20 uses simple operations—adding numbers, rotating bits, and XORing values—to produce a stream of random-looking bytes. Those bytes that form the keystream are XORed with the plaintext to produce the ciphertext.

At first glance, Salsa20 looks almost too simple. It uses a random key, a random nonce, a counter, and a constant byte string, fills them all into a 4×4 matrix of sixteen 32-bit words, and then repeats the same "mixing" procedure 20 times on the rows and columns of the matrix. Yet, this simplicity is what made it popular: ARX operations (add-rotate-XOR) are fast, easy to implement in hardware and software, and always run in constant time. This helps protect against timing attacks. Salsa20 showed that a cipher need not be complicated to be secure. Even today, the 20-round version has no practical attacks and remains one of the fastest algorithms at its security level.

For our project, we wanted to understand Salsa20 not just by reading about it, but by building it. We wrote a Python implementation that followed the official specification, broke the algorithm into small functions to visualize what happened at each step, and then created a menu-driven program to interact with it. The program we built can encrypt and decrypt messages, generate random keys and nonces, show a history log, and even display a complete trace of the 20 rounds of mixing that produces the keystream. Seeing the matrix's internal state change round by round made the algorithm feel much more real, and our public project repository is accessible to anyone interested in using our application and learning about how it works, with the illustrations we created.

We also tested our code using PyTest and compared it to the example values published by Bernstein to verify correctness. Then, we dug into the security side of the Salsa20 stream cipher. We learned that reduced-round variants such as Salsa20/7 and Salsa20/8 (7- and 8-round, respectively) have been attacked using differential cryptanalysis, but the full 20-round version remains secure. Finally, we examined ChaCha20, which was also designed by Daniel in 2008 and builds on the same principles as Salsa20 to increase diffusion per round of matrix mixing. ChaCha20 is now used in major protocols such as TLS and in web browsers like Safari and Firefox. We gained a helpful perspective on how stream ciphers evolve through our exploration.

2 Preliminaries

Salsa20 creates a pseudo-random keystream. To do so, its core structure is a 4x4 block 512-bit matrix. Each block is known as a word. Each word is 32 bits large, so we have 16 32-bit words. Each word block in the matrix is comprised of a mixture of a 256-bit key (separated into eight words and spread around the matrix), a 64-bit nonce (a number used once that is spread into two words), and a two-word counter (64 bits total). The remaining four words (128 bits) are the fixed ASCII string constant: "expand 32-byte k." In total, eight words from the matrix are the key, two words are the nonce, two words are the counter, and the last four words are the constant string.

Figure 1: Initial 4×4 Salsa20 state matrix showing constants, key, nonce, and counter distribution.

3 How Salsa20 works: A deep dive

Data is encrypted by generating a pseudo-random keystream and XORing it with the plaintext. Let's say we have M denoting the plaintext message, C denoting the ciphertext, and K denoting the Salsa20 keystream. We perform the encryption like so to gain the ciphertext C:

$$C = M \oplus K$$

And the decryption is in the same way to regain the original plaintext message:

$$M = C \oplus K$$

XOR makes encryption and decryption identical and super fast. The Salsa20 keystream takes a 256-bit key (32 bytes), such as:

$$K = e8d7b4fb5fa98f4709298bdada208c7ade23d3d360f91a50344cf7afa6caaaa2$$

A 64-bit nonce (8 bytes), which is a random "number used once" to prevent replay attacks and ensure uniqueness, such as:

$$\text{nonce} = aa2c4e02806f6c02$$

And a 64-bit block counter (starting at 0) that prevents keystream repetition across blocks and ensures secure XOR-based encryption. To generate a random key and nonce in our program, we

used the Python secrets module. The module provides a safe way to generate cryptographically substantial random numbers suitable for managing sensitive data, such as passwords, account credentials, security tokens, and related secrets. The key and nonce pairs should never be reused to avoid security breaches.

Next, Salsa20 constructs a 4x4 matrix using the key, nonce, counter, and a constant; each word is 32 bits as described in the preliminaries section. The layout of the initial state is shown in **Figure 2**. Each color indicates a different component of the state. To make the layout of the state easier to read, each group of words is shown with a different color in the diagram:

- Green blocks are the four constant words taken from the ASCII string "expand 32-byte k".
- Pink blocks are the eight words of the 256-bit key.
- Yellow blocks are the two words of the 64-bit nonce.
- Red blocks are the two words of the 64-bit counter.

We can start adding the corresponding values in bytes for the key, nonce, counter, and the constant as for the example:

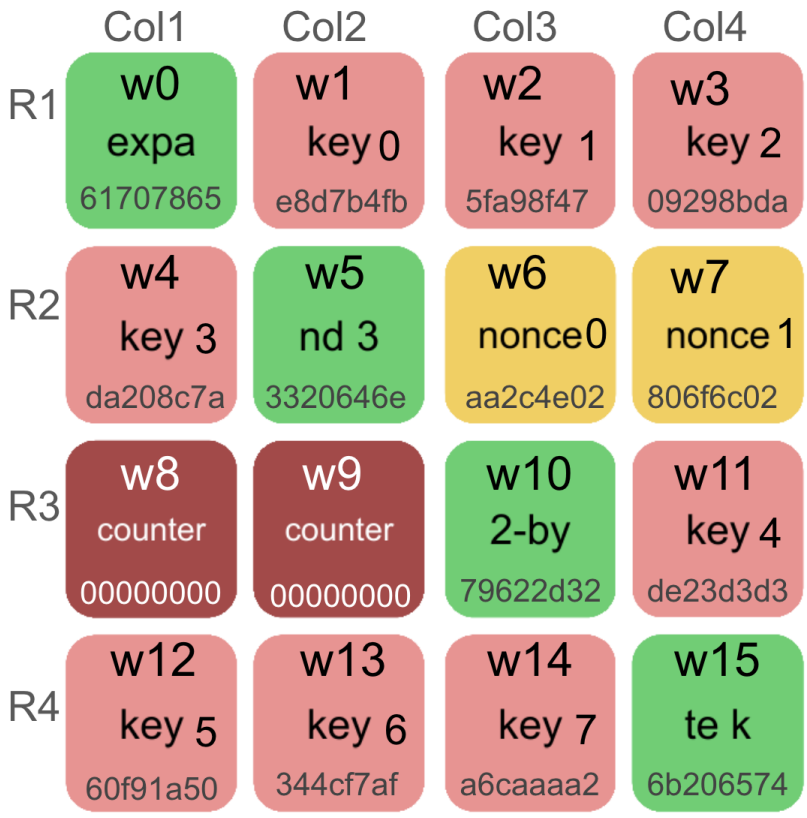

Figure 2: Initial 4×4 Salsa20 state matrix with key, nonce, constant, and counter filled in.

Source	Hex Value	Word Index
“expa”	0x61707865	w0
“nd 3”	0x3320646e	w5
“2-by”	0x79622d32	w10
“te k”	0x6b206574	w15
n_0	0xaa2c4e02	w6
n_1	0x806f6c02	w7
ctr_0	0x00000000	w8
ctr_1	0x00000000	w9
k_0: fb b4 d7 e8	0xe8d7b4fb	w1
k_1: 47 8f a9 5f	0x5fa98f47	w2
k_2: da 8b 29 09	0x09298bda	w3
k_3: 7a 8c 20 da	0xda208c7a	w4
k_4: d3 d3 23 de	0xde23d3d3	w11
k_5: 50 1a f9 60	0x60f91a50	w12
k_6: af f7 4c 34	0x344cf7af	w13
k_7: a2 aa ca a6	0xa6caaaa2	w14

Table 1: Mapping of constants, key, nonce, and counter bytes to their word positions in the Salsa20 state matrix. All bytes are ordered in little-endian.

Now that all blocks in the matrix are in their initial state, the rounds begin to mix this state 20 times. Each ”round” is actually a double round, meaning that the first round is a column round and the second is a row round. There are 10 of these double rounds, for a total of 20 rounds of mixing on the initial matrix state. The rounds are what cause the matrix state to shift, mix, or shuffle, like a Rubik’s Cube. Each round (either a column round or a row round) consists of a quarter-round, which means that we take either a row of the matrix or a column of the matrix (w0, w1, w2, w3 for row 1 or w0, w4, w8, w12 for col 1) and pass each of these words into the quarter-round function, which shuffles them by using ARX operations. ARX stands for Addition, Rotation, and XOR. Since each word is 32 bits, we perform these operations modulo 2^{32} to ensure the sizes are consistent. Salsa20 uses fixed rotation amounts (7, 9, 13, 18), which help spread small changes rapidly across the state while keeping operations simple and fast. The quarter-round:

Listing 1: Salsa20 quarterround

```
def _quarterround(y0: int, y1: int, y2: int, y3: int) -> tuple[int, int,
    int, int]:
    z1 = y1 ^ _rotl32(_u32(y0 + y3), 7)
    z2 = y2 ^ _rotl32(_u32(z1 + y0), 9)
    z3 = y3 ^ _rotl32(_u32(z2 + z1), 13)
    z0 = y0 ^ _rotl32(_u32(z3 + z2), 18)
    return z0, z1, z2, z3
```

Apply the ARX (add-rotate-xor) operation sequence to the corresponding row or column and return the resulting new row or column mixture that we put back into the matrix.

$$
\begin{aligned}
z_1 &= y_1 \oplus \text{ROTL}_{32}(y_0 + y_3, 7) \\
z_2 &= y_2 \oplus \text{ROTL}_{32}(z_1 + y_0, 9) \\
z_3 &= y_3 \oplus \text{ROTL}_{32}(z_2 + z_1, 13) \\
z_0 &= y_0 \oplus \text{ROTL}_{32}(z_3 + z_2, 18)
\end{aligned}
$$

The parameters $y0$, $y1$, $y2$, $y3$ are the four 32-bit words taken from the matrix. The function returns the tuple of the new row or column that has been mixed: the four transformed 32-bit words (z0, z1, z2, z3) after applying the ARX sequence. This ensures that the final matrix is pseudorandom and appears random to the adversary, making it very unlikely to be reverse-engineered or to require the immense effort needed to deter any potential threat from attempting to decrypt our secret message.

Let's investigate how the first column round occurs:

$$\begin{aligned}\text{Column 1: } (w_0, w_4, w_8, w_{12}) &\longrightarrow (w'_0, w'_4, w'_8, w'_{12}) \\ &= QR(\text{expa},\ \text{key3},\ \text{counter},\ \text{key5}) \\ &= QR(0x61707865,\ 0xda208c7a,\ 0x00000000,\ 0x60f91a50).\end{aligned}$$

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

$$\begin{aligned}z_1 &= y_1 \oplus \text{ROL}_{32}(y_0 + y_3,\ 7) \\ &= 0xda208c7a \oplus \text{ROL}_{32}((0x61707865 + 0x60f91a50) \bmod 2^{32},\ 7) \\ &= 0xda208c7a \oplus \text{ROL}_{32}(0xc26992b5,\ 7) \\ &= 0xda208c7a \oplus 0x6b84d325 \\ &= 0xb1a45f5f.\end{aligned}$$

Our new word value is for $z1$, which is the second word in column 1, is "b1a45f5f," and we get the other values like so:

$$\begin{aligned}z_0 &= \texttt{cf921daf} \\ z_2 &= \texttt{801f3a52} \\ z_3 &= \texttt{334e1f50}\end{aligned}$$

After one cycle of the column quarter round, our new matrix looks like this:

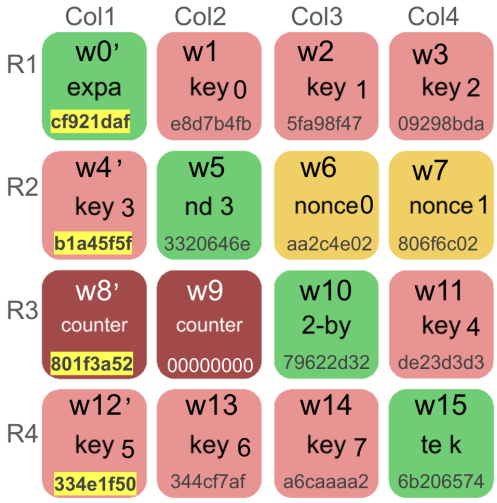

Figure 3: Round 1 of column round on the 4×4 Salsa20 state matrix showing first cycle of column 1 being mixed.

Keep in mind, this process is repeated for each column; we show only the first. After one column round, the other three columns go through the quarter-round operation, and all four columns become mixed.

Next, we discuss the row round, which occurs in the updated matrix state after the column rounds finish. We take the first row as an example:

$$\begin{aligned}\text{R1: } (w_0', w_1', w_2', w_3') &\longrightarrow (w_0'', w_1'', w_2'', w_3'')\\ &= QR(\text{expa},\ \text{key1},\ \text{key2},\ \text{key3})\\ &= QR(0xb1a45f5f,\ 0xe8d7b4fb,\ 0x5fa98f47,\ 0x09298bda).\end{aligned}$$

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

$$\begin{aligned}z_1 &= y_1 \oplus \text{ROL}_{32}(y_0 + y_3,\ 7)\\ &= 0xe8d7b4fb \oplus \text{ROL}_{32}\big((0xb1a45f5f + 0x09298bda) \bmod 2^{32},\ 7\big)\\ &= 0xe8d7b4fb \oplus \text{ROL}_{32}(0xbacdeb39,\ 7)\\ &= 0xe8d7b4fb \oplus 0x66f59cdd\\ &= 0x8e222826.\end{aligned}$$

We can see that column and row round operations are essentially the same. This first cycle of the row round operation mixed the first row of the matrix, producing new values for each word in that row. The first row, word 1, has been updated to "8e222826," while the other three words become:

$$\begin{aligned}z_0 &= \texttt{cf921daf}\\ z_2 &= \texttt{5fa98f47}\\ z_3 &= \texttt{09298bda}\end{aligned}$$

Here is the updated matrix after one cycle of the row round operation:

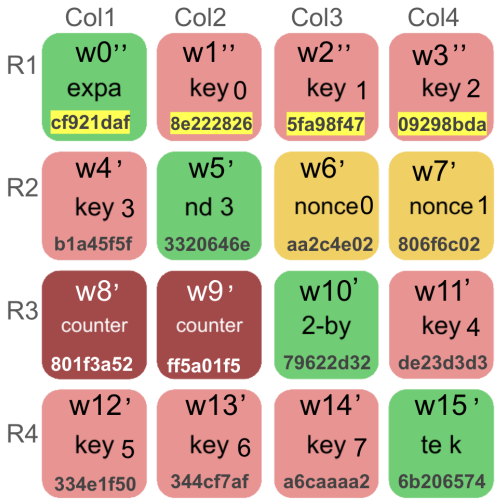

Figure 4: Round 1 of row round on the 4×4 Salsa20 state matrix showing the first cycle of row 1 being mixed.

Once again, this row round is not complete until all rows have been mixed in the quarter-round function. Once that has finished, we have completed one full double round on the matrix, consisting of a full column round followed by a full row round, two total rounds. The Salsa20 hash performs 20 rounds. Then, after all 20 rounds are complete, the hash performs a feedforward pass to increase the matrix's diffusion.

Listing 2: Salsa20 hash

```
def _salsa20_hash(state_words: list[int]) -> bytes:
    # Original state
    x = state_words[:]        # 16 words
    w = state_words[:]        # Working buffer

    # Perform 20 rounds (10 double-rounds)
    for _ in range(10):
        w = _doubleround(w)

    # Feed-forward addition: (w + x) mod 2^32
    out = [(w[i] + x[i]) & 0xffffffff for i in range(16)]

    # Serialize 16 words      64 bytes (little endian)
    return b"".join(_u32_to_le_bytes(v) for v in out)
```

The feedforward operation takes the final mixed matrix w and adds it to the initial state matrix before any rounds were performed. Then it returns a little-endian 64-byte keystream block. For example, after all the rounds and feed forward, the final mixed result keystream could be:

$$K = a2c9b4ad54a552470f308b0a0a608c1aee33d3136ab91ab0158cf7aaaacd9017$$

One thing to mention is that this keystream is XORed with the plaintext. Now, if the plaintext is longer than 64 bytes, we need to extend the keystream. Salsa20 increments the counter and generates the next keystream block by adding another matrix every time the plaintext exceeds the length:

- Block 0 $\rightarrow$ keystream bytes 0–63
- Block 1 $\rightarrow$ keystream bytes 64–127
- Block 2 $\rightarrow$ keystream bytes 128–191
- ...

Now, the final step in the encryption process is to XOR the pseudorandom keystream with the plaintext.

$$C[i] = M[i] \oplus K[i]$$

The process of encryption/decryption is the same:

$$M[i] = C[i] \oplus K[i]$$

This is fast: XOR is one of the simplest and fastest CPU operations. This is also reversible: XOR is its own inverse. To decrypt the ciphertext, we need the same keystream (nonce and key).

4 The Codebase

The Salsa20 encryption algorithm was implemented in Python. The full source code is available in the project repository[1]. We chose to implement the cipher as a set of small functions, each responsible for a specific part of the algorithm. The `quarterround`, `rowround`, `columnround`, and `doubleround` functions follow the design described by Bernstein in the official specification. [2] ChatGPT was used occasionally as a reference to clarify parts of Bernstein's paper and assist with debugging. We verified correctness by testing our output against the example vectors published by Bernstein using the `pytest` framework.

4.1 Codebase Structure

The program is organized into several modules to support modularity and clarity. The layout of the codebase is shown below:

```
.
|-- constants.py
|   `-- "expand 32-byte k"
|-- core.py
|   |-- _initial_state_256(...)
|   |-- _salsa20_hash(...)
|   `-- salsa20_block(...)
|-- helpers.py
|   |-- _u32(...)
|   |-- _rotl32(...)
|   |-- _le_bytes_to_u32(...)
|   `-- _u32_to_le_bytes(...)
|-- main.py
|   |-- do_encrypt()
|   |-- do_decrypt()
|   |-- print_menu()
|   |-- print_history()
|   `-- show_xor_with_final_block(...)
|-- rounds.py
|   |-- _quarterround(...)
|   |-- _rowround(...)
|   |-- _columnround(...)
|   |-- _doubleround(...)
|   `-- _salsa20_hash(...)
|-- stream.py
|   `-- salsa20_stream_xor(...)
|-- logs
|   |-- history.log
|   `-- salsa20_trace.txt
`-- test
    |-- test_core.py
    |-- test_helpers.py
    |-- test_rounds.py
    `-- test_salsa20.py
```

[1] https://github.com/nmetens/CS585-Project
[2] https://cr.yp.to/snuffle/spec.pdf

The `constants.py` file stores the constant string "expand 32-byte k" in byte form. The helper functions in `helpers.py` provide utilities such as 32-bit rotation, byte-to-word conversion, and little-endian casting. The `stream.py` module takes the key, nonce, and plaintext. The key and nonce are passed to the `_salsa20_hash()` method to get the resulting keystream, which is then XORed with the plaintext to produce the ciphertext. The `rounds.py` file holds all matrix manipulation and mixing operations for the quarter-rounds, which are performed on each row and column. The `_doubleround()` method loops ten times and calls `_columnround()` and `_rowround()` (one of each per round) to mix the keystream matrix. The `_salsa20_hash()` method performs the feedforward step (modulo 32 addition) by adding the initial matrix to the final mixed matrix after twenty rounds.

Lastly, the `main.py` module implements the menu interface, allowing the user to perform encryption and decryption, view history, show all twenty rounds of the last encryption, and display session results. All session data is stored in an external file named "history.log," which tracks all encryption and decryption events. The file `salsa20_trace.txt` in the `logs` folder holds the trace of the most recent encryption, including each round shown in matrix form, the feedforward result, and the final XOR that produces the ciphertext.

4.2 Program Menu

The program provides a menu-based interface that allows users to interact with the algorithm. The five options are described below.

Encrypt The user enters a message to encrypt or presses `Enter` to use the default string "hello salsa20". The program generates a random key and nonce using Python's `secrets` module.[3] The plaintext is converted to bytes, and the algorithm produces the ciphertext by running through 20 rounds of state mixing. The key, nonce, and ciphertext are saved to the log file.

Decrypt This option asks for a key, nonce, and ciphertext, then prints the recovered plaintext. Invalid hex input produces an error. All decryption events are logged.

Print History This option prints every encryption and decryption in the history file, including timestamps, the key, nonce, plaintext, and ciphertext.

View Salsa20 Round Trace This option displays a detailed trace of the most recent encryption. The program prints the initial 16-word matrix, then shows each round of row and column operations. After 20 rounds, the final state is added to the original matrix (feed-forward), and each character of plaintext is XORed with the keystream to produce ciphertext. This provides a comprehensive view of how the algorithm evolves internally.

Quit This option displays all encryption/decryption history from the current session, saves the log, and exits the program.

[3] https://docs.python.org/3/library/secrets.html

4.3 Program In Action

We now present our Salsa20 implementation. The menu looks like this to start:

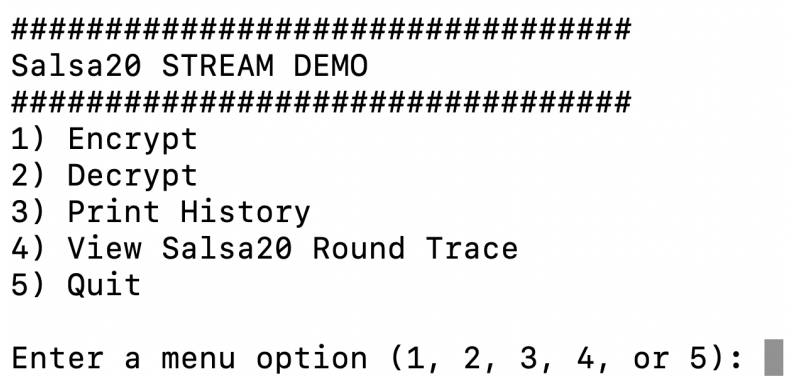

Figure 5: Menu Screen for Salsa20 Demo App.

Then we can encrypt a message with menu option 1:

```
Enter a menu option (1, 2, 3, 4, or 5): 1
Enter a message to encrypt (blank for 'hello salsa20'): attack at midnight, they will be asleep

[+] Key        : f9b1a0ad9d1343b1299590738155bbe6d259edb36e6b14d0626b0b71498f6cf2
[+] Nonce      : 4efb1866de97332a
[+] Plaintext  : b'attack at midnight, they will be asleep'
[+] Ciphertext : f538db222a95b7a28fc0768b169a1b0380d7d47a140b629d378238afe806ebcc7637832fe3bf8e

Save the key, nonce, and ciphertext above if you want to decrypt later.
```

Figure 6: Encrypt a secret message.

As shown, the key and nonce are generated using the Python secrets module and are used to perform 20 rounds of mixing, which produce the keystream that is XORed with the plaintext to produce the ciphertext. Next, we can decrypt the message using the generated key and nonce:

```
Enter a menu option (1, 2, 3, 4, or 5): 2
Enter key (hex): f9b1a0ad9d1343b1299590738155bbe6d259edb36e6b14d0626b0b71498f6cf2
Enter nonce (hex): 4efb1866de97332a
Enter ciphertext (hex): f538db222a95b7a28fc0768b169a1b0380d7d47a140b629d378238afe806ebcc7637832fe
3bf8e

[+] Decrypted bytes (hex): 61747461636b206174206d69646e696768742c20746865792077696c6c206265206173
6c656570
[+] Decrypted text      : attack at midnight, they will be asleep
```

Figure 7: Decrypt a secret message.

We can then print the history:

```
------------------------------------------------------------
Timestamp  :  2025-12-10T14:30:21.752684
Operation  :  encrypt
Key        :  f9b1a0ad9d1343b1299590738155bbe6d259edb36e6b14d0626b0b71498f6cf2
Nonce      :  4efb1866de97332a
Plaintext  :  attack at midnight, they will be asleep
Plain(hex) :  61747461636b206174206d69646e696768742c2074686579207769 6c6c2062652061736c656570
Cipher(hex): f538db222a95b7a28fc0768b169a1b0380d7d47a140b629d378238afe806ebcc7637832fe3bf8e
------------------------------------------------------------
Timestamp  :  2025-12-10T14:35:17.276092
Operation  :  decrypt
Key        :  f9b1a0ad9d1343b1299590738155bbe6d259edb36e6b14d0626b0b71498f6cf2
Nonce      :  4efb1866de97332a
Cipher(hex): f538db222a95b7a28fc0768b169a1b0380d7d47a140b629d378238afe806ebcc7637832fe3bf8e
Plain(hex) : 61747461636b206174206d69646e696768742c2074686579207769 6c6c2062652061736c656570
Plaintext  : attack at midnight, they will be asleep
------------------------------------------------------------
```

Figure 8: History of encryption/decryption.

Next, we can view the entire encryption process (trace) from our last encryption:

```
=== SALSA20 ROUND TRACE ===


Initial state (round 0):
      c0        c1        c2        c3
r0   61707865  ada0b1f9  b143139d  73909529
r1   e6bb5581  3320646e  6618fb4e  2a3397de
r2   00000000  00000000  79622d32  b3ed59d2
r3   d0146b6e  710b6b62  f26c8f49  6b206574

After doubleround 1 (round 2):
      c0        c1        c2        c3
r0   8f565566  7b441534  ccfdcf6f  c9db5f2e
r1   4012a899  959b40ca  2a0cd091  79f6562f
r2   22e12c8b  dad8f601  3cce683d  0caca613
r3   5e8019aa  d363d8ed  31aa291a  fde68665

After doubleround 2 (round 4):
      c0        c1        c2        c3
r0   c5da0b68  48fda1e5  283362db  a0d56d0d
r1   3fb8ac80  8cf50fe8  3cc7b264  26b4f288
r2   9e7b3954  cbb74b60  7205d643  74d0bcd6
r3   a4b28199  7a54419c  8450edc9  f2b0aa12

After doubleround 3 (round 6):
      c0        c1        c2        c3
r0   bd807dbd  1697ed5b  a95c5b10  e96b6ec0
r1   9a44eac4  7e641263  5f17c9ed  d87753e9
r2   599441a0  5d88c72d  69225af2  fba7399b
r3   a8e571eb  7b798bae  3d279a95  64f05b28
```

Figure 9: Trace through initial state to round 6.

```
After doubleround 7 (round 14):
      c0        c1        c2        c3
r0   da1f6b83  dfc0d6ac  0b9edc40  1f737239
r1   861bfae5  de358ca0  d567b339  ff0d1adb
r2   38c346bf  c52c74fd  789d3848  d271ec35
r3   2e706be7  77de4169  69541afd  6b46a8d6

After doubleround 8 (round 16):
      c0        c1        c2        c3
r0   c3a2dd3c  5b356c00  23ae07ba  ad4eb87b
r1   506fee0e  018ec910  50b95191  bec33b6c
r2   bca0b439  56658a95  2102b70c  05128f89
r3   d3946ac1  2c25f848  2740e5ad  cebf7a45

After doubleround 9 (round 18):
      c0        c1        c2        c3
r0   b7df8354  81485dc8  60f2983b  1a8a2355
r1   2368e50e  3bf711ed  f6e88901  b776bc26
r2   6dc2d274  e935b324  9c2b1cb3  1f3c17e1
r3   392d2d4d  ada51733  b1d4b7cf  6e96a388

After doubleround 10 (round 20):
      c0        c1        c2        c3
r0   e23ed42f  15f74c50  30d8cd5e  f0e25f49
r1   743d4e67  b0e6fef2  5d38f9c9  7f558ea6
r2   43f05656  6dfeda86  37645dc5  57f0858f
r3   532656c4  4970cbdb  dc559e14  d5582d5c


=== END OF TRACE ===
```

Figure 10: Trace through rounds 14 through 20.

After each round trace, the program then displays the resulting final round's keystream before it is added to the initial keystream to complete the Salsa20 hash function:

```
=== CORE STATE AFTER 20 ROUNDS (before feed-forward) ===

Words (hex): ['0xe23ed42f', '0x15f74c50', '0x30d8cd5e', '0xf0e25f49', '0x743d4e67', '0xb0e6fef2',
 '0x5d38f9c9', '0x7f558ea6', '0x43f05656', '0x6dfeda86', '0x37645dc5', '0x57f0858f', '0x532656c4'
, '0x4970cbdb', '0xdc559e14', '0xd5582d5c']

Bytes (LE): 2fd43ee2504cf7155ecdd830495fe2f0674e3d74f2fee6b0c9f9385da68e557f5656f04386dafe6dc55d6
4378f85f057c4562653dbcb7049149e55dc5c2d58d5

4×4 Core Matrix (20 rounds, no feed-forward):
  e23ed42f  15f74c50  30d8cd5e  f0e25f49
  743d4e67  b0e6fef2  5d38f9c9  7f558ea6
  43f05656  6dfeda86  37645dc5  57f0858f
  532656c4  4970cbdb  dc559e14  d5582d5c

=== FINAL SALSA20 BLOCK (after feed-forward) ===

Words (hex): ['0x43af4c94', '0xc397fe49', '0xe21be0fb', '0x6472f472', '0x5af8a3e8', '0xe4076360',
 '0xc351f517', '0xa9892684', '0x43f05656', '0x6dfeda86', '0xb0c68af7', '0xbdddf61', '0x233ac232',
 '0xba7c373d', '0xcec22d5d', '0x407892d0']

Bytes (LE): 944caf4349fe97c3fbe01be272f47264e8a3f85a606307e417f551c3842689a95656f04386dafe6df78ac
6b061dfdd0b32c23a233d377cba5d2dc2ced0927840

4×4 Final Block Matrix (keystream block):
  43af4c94  c397fe49  e21be0fb  6472f472
  5af8a3e8  e4076360  c351f517  a9892684
  43f05656  6dfeda86  b0c68af7  0bdddf61
  233ac232  ba7c373d  cec22d5d  407892d0
```

Figure 11: Final keystream.

Then, as the final section of this trace execution, the program shows the plaintext being XORed with the keystream to get the resulting ciphertext:

```
=== XOR DEMO: plaintext ⊕ keystream_block = ciphertext ===

idx plain  ks   cipher  | p  ⊕  k  =  c
--- -----  ---  ------  | -------------
  0   0x61  0x94    0xf5  |  a ⊕ \x94= \xf5
  1   0x74  0x4c    0x38  |  t ⊕ L   = 8
  2   0x74  0xaf    0xdb  |  t ⊕ \xaf= \xdb
  3   0x61  0x43    0x22  |  a ⊕ C   = "
  4   0x63  0x49    0x2a  |  c ⊕ I   = *
  5   0x6b  0xfe    0x95  |  k ⊕ \xfe= \x95
  6   0x20  0x97    0xb7  |    ⊕ \x97= \xb7
  7   0x61  0xc3    0xa2  |  a ⊕ \xc3= \xa2
  8   0x74  0xfb    0x8f  |  t ⊕ \xfb= \x8f
  9   0x20  0xe0    0xc0  |    ⊕ \xe0= \xc0
 10   0x6d  0x1b    0x76  |  m ⊕ \x1b= v
 11   0x69  0xe2    0x8b  |  i ⊕ \xe2= \x8b
 12   0x64  0x72    0x16  |  d ⊕ r   = \x16
 13   0x6e  0xf4    0x9a  |  n ⊕ \xf4= \x9a
 14   0x69  0x72    0x1b  |  i ⊕ r   = \x1b
 15   0x67  0x64    0x03  |  g ⊕ d   = \x03
 16   0x68  0xe8    0x80  |  h ⊕ \xe8= \x80
 17   0x74  0xa3    0xd7  |  t ⊕ \xa3= \xd7
 18   0x2c  0xf8    0xd4  |  , ⊕ \xf8= \xd4
 19   0x20  0x5a    0x7a  |    ⊕ Z   = z
 20   0x74  0x60    0x14  |  t ⊕ `   = \x14
 21   0x68  0x63    0x0b  |  h ⊕ c   = \x0b
 22   0x65  0x07    0x62  |  e ⊕ \x07= b
 23   0x79  0xe4    0x9d  |  y ⊕ \xe4= \x9d
 24   0x20  0x17    0x37  |    ⊕ \x17= 7
 25   0x77  0xf5    0x82  |  w ⊕ \xf5= \x82
 26   0x69  0x51    0x38  |  i ⊕ Q   = 8
 27   0x6c  0xc3    0xaf  |  l ⊕ \xc3= \xaf
 28   0x6c  0x84    0xe8  |  l ⊕ \x84= \xe8
 29   0x20  0x26    0x06  |    ⊕ &   = \x06
 30   0x62  0x89    0xeb  |  b ⊕ \x89= \xeb
 31   0x65  0xa9    0xcc  |  e ⊕ \xa9= \xcc
 32   0x20  0x56    0x76  |    ⊕ V   = v
 33   0x61  0x56    0x37  |  a ⊕ V   = 7
 34   0x73  0xf0    0x83  |  s ⊕ \xf0= \x83
 35   0x6c  0x43    0x2f  |  l ⊕ C   = /
 36   0x65  0x86    0xe3  |  e ⊕ \x86= \xe3
 37   0x65  0xda    0xbf  |  e ⊕ \xda= \xbf
 38   0x70  0xfe    0x8e  |  p ⊕ \xfe= \x8e
```

Figure 12: XOR plaintext with the final keystream to get the ciphertext.

```
Plaintext : b'attack at midnight, they will be asleep'
Keystream : 944caf4349fe97c3fbe01be272f47264e8a3f85a606307e417f551c3842689a95656f04386dafe6df78ac6b061dfdd0b32c2
3a233d377cba5d2dc2ced0927840
Ciphertext: f538db222a95b7a28fc0768b169a1b0380d7d47a140b629d378238afe806ebcc7637832fe3bf8e
=== END XOR DEMO ===
```

Figure 13: Resulting ciphertext after the Salsa20 encryption trace.

This ciphertext can be decrypted by selecting option two from the menu and passing the key and nonce from the previous encryption to verify correctness.

The final menu option is to quit the application, which will display the current session's encryption and decryption history:

```
Enter a menu option (1, 2, 3, 4, or 5): 5

========== SESSION HISTORY ==========

[1] Operation: ENCRYPT
    Key   : f9b1a0ad9d1343b1299590738155bbe6d259edb36e6b14d0626b0b71498f6cf2
    Nonce : 4efb1866de97332a
    Plaintext      : 'attack at midnight, they will be asleep'
    Plaintext (hex): 61747461636b206174206d69646e696768742c2074686579207769 6c6c2062652061736c656570
    Ciphertext(hex): f538db222a95b7a28fc0768b169a1b0380d7d47a140b629d378238afe806ebcc7637832fe3bf8e

[2] Operation: DECRYPT
    Key   : f9b1a0ad9d1343b1299590738155bbe6d259edb36e6b14d0626b0b71498f6cf2
    Nonce : 4efb1866de97332a
    Ciphertext(hex): f538db222a95b7a28fc0768b169a1b0380d7d47a140b629d378238afe806ebcc7637832fe3bf8e
    Plaintext (hex): 61747461636b206174206d69646e696768742c2074686579207769 6c6c2062652061736c656570
    Plaintext      : 'attack at midnight, they will be asleep'

====================================

History saved to history.log
Thanks for testing Salsa20!
```

Figure 14: Quit and display session history.

4.4 Testing

We verified correctness by writing a comprehensive test suite using `pytest`. Each component of the implementation was tested individually, and our results were compared to the example values in Bernstein's paper. This ensured that the mixing functions, state initialization, and keystream generation followed the specification exactly. This is available for free on our public GitHub repository.

5 Breaking Salsa20

Currently, there is no faster way to break Salsa20 outside of a brute force attack, which goes through all 2^{256} possible keys. Although the standard 20 rounds remain unbroken, there have been cryptanalysts who have targeted earlier rounds such as Salsa20/7 & Salsa20/8. The current attacks on Salsa20/7 & Salsa20/8 are built on the fact that the "avalanching" diffusion effect does not occur until about 10 rounds in Salsa. This means that an adversary could use truncated differentials to distinguish the key from a random word. An example of this is Salsa20/5, where Paul Crowley was able to break Salsa20/5 using clusters of truncated differentials with 2^{165} operations and 2^6 plaintexts. Dey, Maitra, Sarkar, and Kumar Sharma were able to break Salsa20/8.5 in a hypothetical scenario where the same key was used in two variants of Salsa (20/8 & 20/12), and different nonces. This still took a time complexity of $2^{193.6}$ operations under ideal circumstances. Both attacks show that many operations are still required to break Salsa, even with reduced rounds. These attacks also show that lowering rounds for speed reduces the cipher's security; however, using the full 20 rounds is highly secure, as truncated attacks are not feasible due to avalanche diffusion.

6 Chacha20: An Upgrade

ChaCha20 is a variant of Salsa20 developed by Daniel J. Bernstein in 2008. Although we do not develop a program that uses it, we include it here to identify improvements to the Salsa20 design structure. ChaCha follows in its older brother's footsteps, using the same ARX design with additional updates and a revised matrix. Instead of the original:

$$\begin{aligned} b &\oplus (a + d) \lll 7; \\ c &\oplus (b + a) \lll 9; \\ d &\oplus (c + b) \lll 13; \\ a &\oplus (d + c) \lll 18; \end{aligned} \tag{1}$$

ChaCha now does column and diagonal rounds with this ARX design:

$$\begin{aligned} a &+= b, \quad d \oplus= a, \quad d \lll 16; \\ c &+= d, \quad b \oplus= c, \quad b \lll 12; \\ a &+= b, \quad d \oplus= a, \quad d \lll 8; \\ c &+= d, \quad b \oplus= c, \quad b \lll 7; \end{aligned} \tag{2}$$

This design accelerates diffusion relative to Salsa20, as each word block is updated twice per round rather than once. Since the words no longer depend on each other within a step, running in parallel for speed is possible. An interesting feature that comes from the ARX design is that with two of the rounds being rotated at a multiples of 8, this helps boost efficiency on architectures like x86.

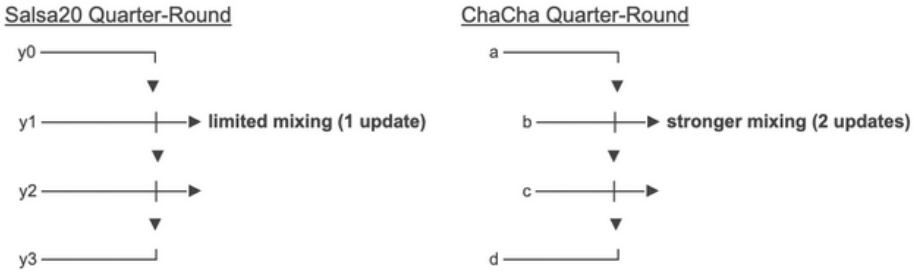

Figure 15: Salsa20 vs ChaCha Quarter-Round Diagram.

Due to this change in ARX formatting, the block's initial state also had to be updated. When performing a diagonal round, if this were done in Salsa20's initial matrix, diffusion would be extremely weak in the first few rounds because the constant is placed diagonally across the matrix. To counteract this, ChaCha20's initial matrix starts with the constant moving down the row, then follows the key, counter, and nonce.

Cons	Cons	Cons	Cons
Key	Key	Key	Key
Key	Key	Key	Key
Counter	Counter	Nonce	Nonce

Figure 16: Initial State of ChaCha20

ChaCha is currently being utilized in many modern software, such as Google's QUIC, WireGuard, TLS 1.2 & 1.3, etc. The strength of the ARX design and the efficiency of both Salsa and ChaCha make them strong candidates for environments with weaker hardware that require a modern, secure stream cipher.

7 Conclusion

Salsa20 is a highly efficient stream cipher that optimizes the ARX design to near its current limits. Being fast in both hardware and software, it is a compelling choice for those who want something lightweight and secure. It shows that, to be secure, the cipher itself need not be wildly complex: with just three simple operations, Salsa20 has proven incredibly hard to beat. ChaCha20 builds on that and further refines the ARX design, suggesting there is more potential in the simple design philosophy that may not yet have been realized. Given that ChaCha is the basis of the BLAKE hash function, this suggests that the development of these ciphers still has a long way to go.

The overall purpose of this topic was to learn how stream ciphers work. We wanted to explore a more hands-on approach that would let us implement it ourselves. We used the papers and research to acquire the knowledge, then built the encryption and decryption functions from scratch in our own public GitHub repository. We searched for many example encryptions of the Salsa20 stream cipher online. Still, they were not easy to find, if at all, so we decided to create the trace functionality in our program that clearly describes the process of each round and shows all the steps individually in the encryption process. We wanted to help others interested in the process see it for themselves, so we made our code public.

References

[GR25] Ian Grigg. *Salsa20 Usage & Deployment*. Ianix, updated 23 Oct. 2025. https://ianix.com/pub/salsa20-deployment.html.

[IA25] Ianix. *ChaCha Usage & Deployment*. Published 2 Oct. 2025. https://ianix.com/pub/chacha-deployment.html.

[Nag23] Karthikeyan Nagaraj. "Understanding Salsa20 Encryption: A Comprehensive Guide." *System Weakness*, 18 Mar. 2023. https://systemweakness.com/understanding-salsa20-encryption-a-comprehensive-guide-2023-2d6688889e4.

[Wik25a] Wikipedia contributors. "Salsa20." *Wikipedia: The Free Encyclopedia*. Accessed 25 Nov. 2025. https://en.wikipedia.org/wiki/Salsa20.

[Lib25] Libsodium Project. "Salsa20." *Libsodium Documentation*. Accessed 25 Nov. 2025. https://libsodium.gitbook.io/doc/advanced/stream_ciphers/salsa20.

[Ber05a] Daniel J. Bernstein. *Salsa20 Speed*. University of Illinois at Chicago, 2005. https://cr.yp.to/snuffle/speed.pdf.

[Ber05b] Daniel J. Bernstein. *Salsa20 Specification*. Published 27 Apr. 2005. https://cr.yp.to/snuffle/spec.pdf.

[Ber07] Daniel J. Bernstein. *The Salsa20 Family*. 25 Dec. 2007. http://cr.yp.to/snuffle/salsafamily-20071225.pdf.

[Com18] Computerphile. "ChaCha20: Security and Speed." YouTube video, uploaded 8 May 2018. https://www.youtube.com/watch?v=Eqx0pxfn9GY.

[OAI25] OpenAI. *ChatGPT (GPT-5.1)*. Accessed 30 Nov. 2025. https://chat.openai.com/.

[Sha25] David Shaw. "The Secrets Module." *Real Python*. Accessed 25 Nov. 2025. https://realpython.com/ref/stdlib/secrets/.

[Ama25] Andrea Amadori et al. "Improved Related-Cipher Attack on Salsa20 Stream Cipher." IACR Cryptology ePrint Archive, Report 2025/289. https://eprint.iacr.org/2025/289.pdf.

[Ama19] Andrea Amadori et al. "Improved Related-Cipher Attack on Salsa20 Stream Cipher." ResearchGate, 2019. https://www.researchgate.net/publication/331664229_Improved_Related-Cipher_Attack_on_Salsa20_Stream_Cipher.

[FerXX] Niels Ferguson. *Salsa20 Cryptanalysis*. https://www.ciphergoth.org/crypto/salsa20/salsa20-cryptanalysis.pdf.

[Ber08] Daniel J. Bernstein. *ChaCha, a Variant of Salsa20*. Published 20 Jan. 2008. https://cr.yp.to/chacha/chacha-20080120.pdf.

[Com25] Compile7. "What Is the Difference Between ChaCha20-256 vs Salsa20-128?" https://compile7.org/compare-encryption-algorithms/what-is-difference-between-chacha20-256-vs-salsa20-128/.

[ARX25] Wikipedia contributors. "ARX (add–rotate–XOR)." *Wikipedia: The Free Encyclopedia*. Accessed 25 Nov. 2025. https://en.wikipedia.org/wiki/Block_cipher#ARX_(add%E2%80%93rotate%E2%80%93XOR).

[BEL05] M. Bellare et al. "Related-Key Cryptanalysis of Salsa-Family Stream Ciphers." IACR ePrint Archive, Report 2005/375. https://eprint.iacr.org/2005/375.pdf.

[GC23] Google Cloud. "Introducing QUIC Support for HTTPS Load Balancing." https://cloud.google.com/blog/products/gcp/introducing-quic-support-https-load-balancing.

[WG25] WireGuard Project. "WireGuard Protocol." https://www.wireguard.com/protocol/.

[TLS25] Wikipedia contributors. "Transport Layer Security." *Wikipedia: The Free Encyclopedia*. Accessed 25 Nov. 2025. https://en.wikipedia.org/wiki/Transport_Layer_Security.

[WM25a] Nick Whiteman and Nathan Metens. *Salsa20 Presentation*. Portland State University, 2025.

[WM25b] Nick Whiteman and Nathan Metens. *Project Proposal: Salsa20*. Portland State University, 2025.

[AI25] OpenAI. *ChatGPT (Model GPT-5.1)*. Assistance used for explanation, debugging, diagram creation, and report structuring throughout the Salsa20 project. Accessed 2025. https://chat.openai.com/.

[WM25c] Nick Whiteman and Nathan Metens. *Salsa20 Project Repository*. Source code, documentation, trace outputs, and presentation materials for the CS 585 Salsa20 project. Portland State University, 2025. https://github.com/nmetens/Salsa20.