

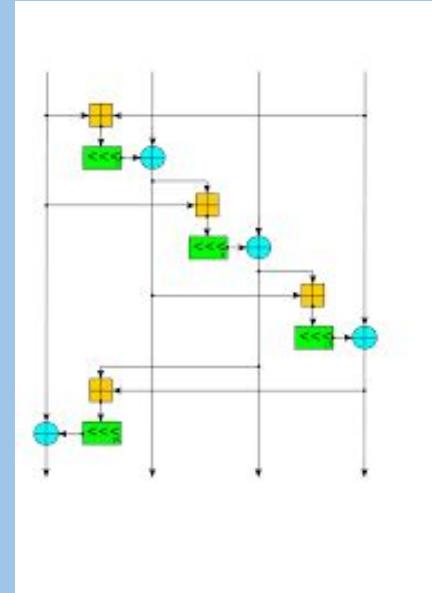
Salsa20



Nick Whiteman & Nathan Metens

Agenda

- 1) Background on Salsa20
- 2) How it works
- 3) Examples and Demonstration
- 4) How it's used
- 5) Chacha20 an upgrade
- 6) Is Salsa20 Dead?
- 7) Conclusion



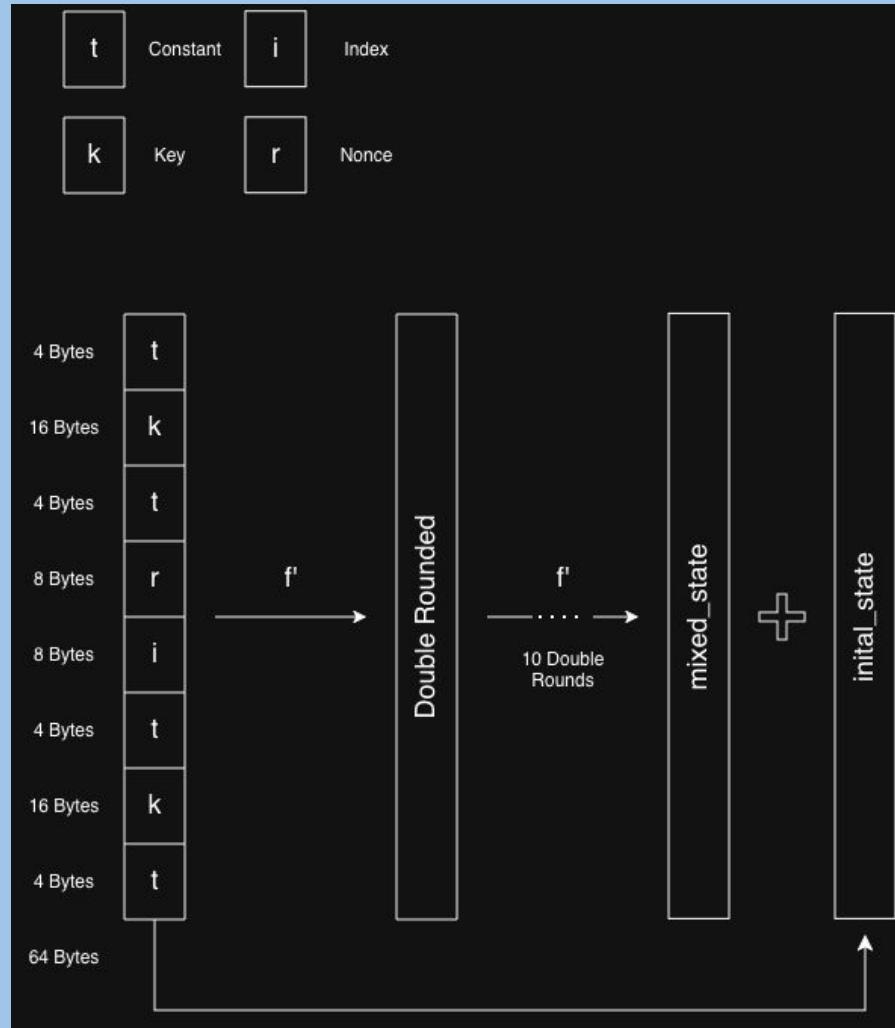
Background

- Salsa20 was designed by [Daniel J. Bernstein](#) in 2005
- Salsa20 encryption is a stream cipher
 - Same key used for encryption and decryption.
- Generates pseudo-random key stream that is XORed with the plaintext to produce ciphertext.
 - $\text{ciphertext} = \text{plaintext} \oplus \text{Salsa20(key, nonce)}$
- It is used in various applications:
 - wireless networks
 - secure sockets layer (SSL)
 - virtual private networks (VPNs)
- Not far from the fastest cryptographic function at the same conjectured security level



Overview—How it works

- Built off: Key, Nonce, Index, Constant
- Two Constants depending on version of Salsa
 - **Salsa20/128**: “expand 16-byte k”
 - **Salsa20/256**: “expand 32-byte k”
- Utilizes a ARX design
 - A: 32-bit addition
 - R: Bitwise rotation
 - X: XOR addition
- Each block is structured around 10 “double rounds.”
 - Round 1: column round
 - Round 2: row round
- A block ends with the mixed state being added with the initial state to make it non-reversible



Salsa20—Example Encryption Scheme

- Each “word” is 32 bits
 - Orange: Key
 - Green: Nonce
 - Red: Index
 - Blue: Constant
- Start with a column round (1st column):
 - $w4 \oplus \text{ROTL} (w0 + w12, 7)$
 - $w8 \oplus \text{ROTL} (w4 + w0, 9)$
 - $w12 \oplus \text{ROTL} (w8 + w4, 13)$
 - $w0 \oplus \text{ROTL} (w12 + w8, 18)$
- Repeat but with rows to complete a rowround
- Do this 10 times for standard Salsa/20, but can be done less or more if needed

w0	w1	w2	w3
w4	w5	w6	w7
w8	w9	w10	w11
w12	w13	w14	w15

Run Through—Example

Data is encrypted by generating a pseudo-random keystream and XORing it with the plaintext message.

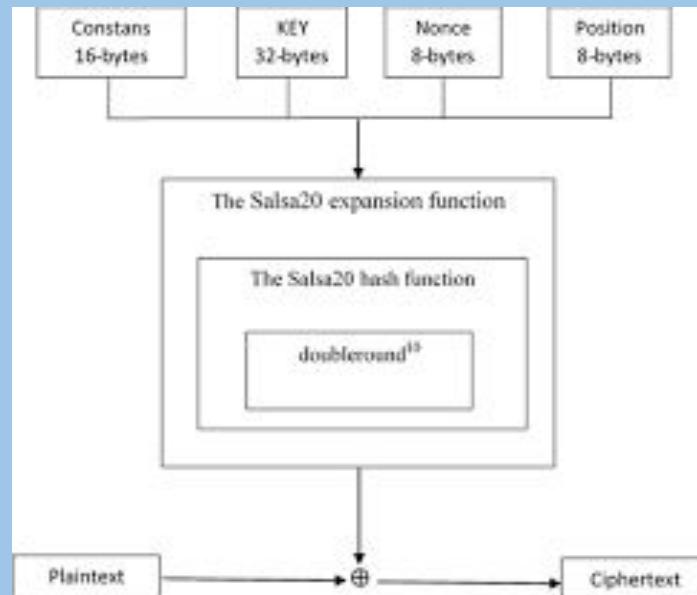
- M = plaintext message
- C = ciphertext
- K = Salsa20 keystream

Encryption

$$- C = M \oplus K$$

Decryption

$$- M = C \oplus K$$



XOR makes encryption and decryption identical.

The Inputs

Salsa20 keystream takes:

- 256-bit key (**32 bytes**)
 - Ex: e8d7b4fb5fa98f4709298bdada208c7ade23d3d360f91a50344cf7afa6caaaa2
 - 64-bit nonce (**8 bytes**):
 - Random “number used once” to prevent replay attacks and ensure uniqueness.
 - Ex: aa2c4e02806f6c02
 - 64-bit block counter (starts at 0)
 - Prevents keystream repetition across blocks
 - Ensures secure XOR-based encryption
- Python **secrets** [module](#): a secure way to generate cryptographically strong 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.

The 16-Word 4x4 State Matrix

Salsa20 constructs a 4x4 matrix; each word is 32 bits:

- 1) Green: constants from “expand 32-byte k”
- 2) Pink: 8 words of the 256-bit key
- 3) Yellow: nonce
- 4) Red: counter blocks

w0 expa	w1 key	w2 key	w3 key
w4 key	w5 nd 3	w6 nonce	w7 nonce
w8 counter	w9 counter	w10 2-by	w11 key
w12 key	w13 key	w14 key	w15 te k

```
def _initial_state_256(key32: bytes, nonce8: bytes, counter64: int) -> list[int]:  
  
    c = SIGMA # b"expand 32-byte k"  
    k0, k1 = key32[:16], key32[16:] # Key split  
    f = 0xffffffff # 32 bit  
  
    return [ # Return the 16 word 4x4 matrix where each word is 32 bits:  
        # "expa"           w1           w2           w3  
        lb232(c[0:4]),   lb232(k0[0:4]),   lb232(k0[4:8]),   lb232(k0[8:12]),  
        # w1               "nd 3"       "none"       "nonce"  
        lb232(k0[12:16]), lb232(c[4:8]),   lb232(nonce8[0:4]), lb232(nonce8[4:8]),  
        # c               c           "2-by"       w11  
        counter64 & f,    (counter64 >> 32) & f,  lb232(c[8:12]),   lb232(k1[0:4]),  
        # w12              w13          w14          "te k"  
        lb232(k1[4:8]),   lb232(k1[8:12]),   lb232(k1[12:16]), lb232(c[12:16]),  
    ]
```

The 16-Word 4x4 State Matrix

Salsa20 constructs a 4x4 matrix; each word is 32 bits and all blocks are ordered in Little-Endian:

"expa" → 0x61707865 k0: fb b4 d7 e8 → 0xe8d7b4fb

"nd 3" → 0x3320646e k1: 47 8f a9 5f → 0x5fa98f47

"2-by" → 0x79622d32 k2: da 8b 29 09 → 0x09298bda

"te k" → 0x6b206574 k3: 7a 8c 20 da → 0xda208c7a

n0 → 0xaa2c4e02 k4: d3 d3 23 de → 0xde23d3d3

n1 → 0x806f6c02 k5: 50 1a f9 60 → 0x60f91a50

ctr0 = 0x00000000 k6: af f7 4c 34 → 0x344cf7af

ctr1 = 0x00000000 k7: a2 aa ca a6 → 0xa6caaaa2

	Col1	Col2	Col3	Col4
R1	w0 expa 61707865	w1 key 0 e8d7b4fb	w2 key 1 5fa98f47	w3 key 2 09298bda
R2	w4 key 3 da208c7a	w5 nd 3 3320646e	w6 nonce0 aa2c4e02	w7 nonce 1 806f6c02
R3	w8 counter 00000000	w9 counter 00000000	w10 2-by 79622d32	w11 key 4 de23d3d3
R4	w12 key 5 60f91a50	w13 key 6 344cf7af	w14 key 7 a6caaaa2	w15 te k 6b206574

The Salsa20/20 Core (20 Rounds)

Each keystream block is produced by running 20 rounds of mixing on the matrix.

Rounds come in pairs:

- A double-round
 - 1. ColumnRound
 - 2. RowRound
- 20 rounds = 10 double rounds

Each round has QuarterRound:

- $(y_0, y_1, y_2, y_3) \rightarrow (z_0, z_1, z_2, z_3)$

QuarterRound uses ARX operations:

- Addition mod 2^{32}
- Rotate left
- XOR

```
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  
  
def _rowround(y: list[int]) -> list[int]:  
    y0, y1, y2, y3 = _quarterround(y[0], y[1], y[2], y[3])  
    y5, y6, y7, y4 = _quarterround(y[5], y[6], y[7], y[4])  
    y10, y11, y8, y9 = _quarterround(y[10], y[11], y[8], y[9])  
    y15, y12, y13, y14 = _quarterround(y[15], y[12], y[13], y[14])  
    return [y0, y1, y2, y3, y4, y5, y6, y7, y8, y9, y10, y11, y12, y13, y14, y15]  
  
def _columnround(x: list[int]) -> list[int]:  
    x0, x4, x8, x12 = _quarterround(x[0], x[4], x[8], x[12])  
    x5, x9, x13, x1 = _quarterround(x[5], x[9], x[13], x[1])  
    x10, x14, x2, x6 = _quarterround(x[10], x[14], x[2], x[6])  
    x15, x3, x7, x11 = _quarterround(x[15], x[3], x[7], x[11])  
    return [x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13, x14, x15]
```

All of this makes the cipher super fast and resistant to timing attacks.

Column Round—Example

Col1: $(w_0, w_4, w_8, w_{12}) \rightarrow (w'_0, w'_4, w'_8, w'_{12})$

$= QR(expa, key3, counter, key5)$

$= QR(61707865, da208c7a, 00000000, 60f91a50)$

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

$z_1 = y_1 \oplus \text{ROL32}(\text{ADD32}(y_0 + y_3), 7)$

$= da208c7a \oplus \text{ROL32}((61707865 + 60f91a50) \bmod 2^{32}, 7)$

$= da208c7a \oplus \text{ROL32}(c26992b5, 7)$

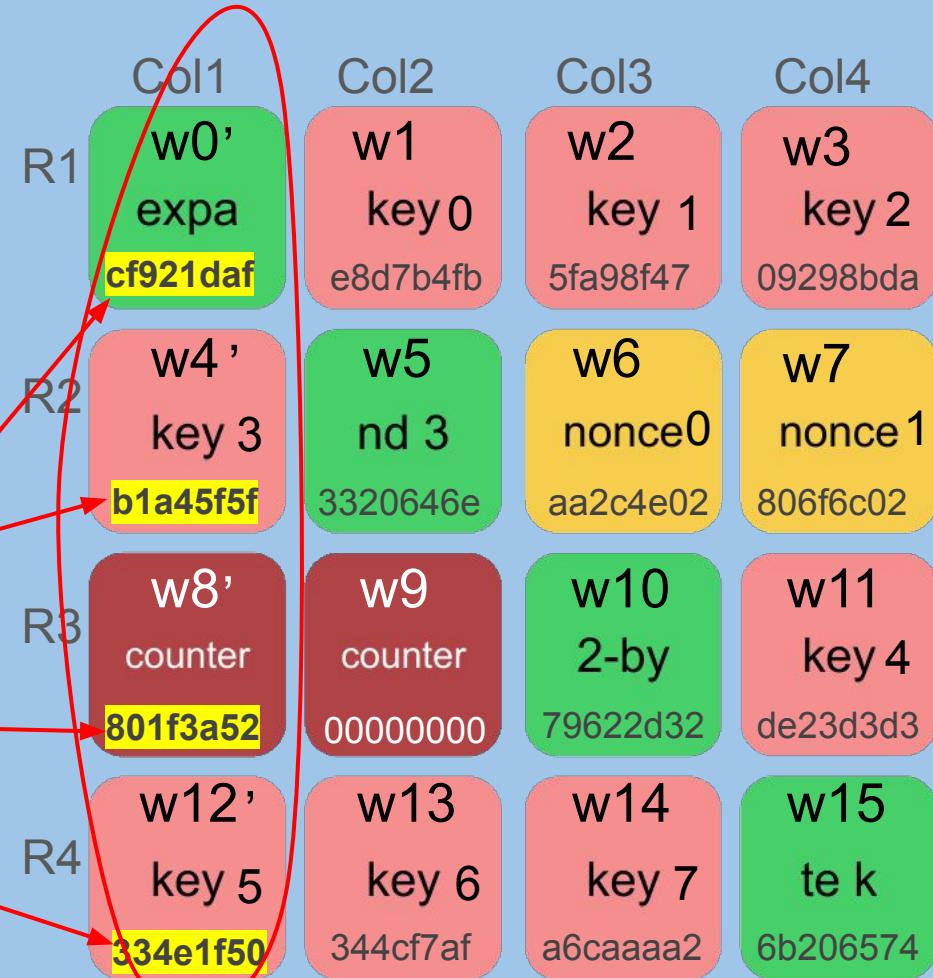
$= da208c7a \oplus 6B84D325 = b1a45f5f$

$z_2 = y_2 \oplus \text{ROL32}(\text{ADD32}(z_1 + y_0), 9)$

$z_3 = y_3 \oplus \text{ROL32}(\text{ADD32}(z_2 + z_1), 13)$

$z_0 = y_0 \oplus \text{ROL32}(\text{ADD32}(z_3 + z_2), 18)$

Repeat for every column!



Row Round—Example

R1: $(w0', w1', w2', w3') \rightarrow (w0'', w1'', w2'', w3'')$

$= QR(expa, key0, key1, key2)$

$= QR(b1a45f5f, e8d7b4fb, 5fa98f47, 09298bda)$

$(z0, z1, z2, z3) = QR(y0, y1, y2, y3):$

$z1 = y1 \oplus \text{ROL32}(\text{ADD32}(y0 + y3), 7)$

$= e8d7b4fb \oplus \text{ROL32}((b1a45f5f + 09298bda) \bmod 2^{32}, 7)$

$= e8d7b4fb \oplus \text{ROL32}(bacdeb39, 7)$

$= e8d7b4fb \oplus 66f59cdd = 8e222826$

$z2 = y2 \oplus \text{ROL32}(\text{ADD32}(z1 + y0), 9)$

$z3 = y3 \oplus \text{ROL32}(\text{ADD32}(z2 + z1), 13)$

$z0 = y0 \oplus \text{ROL32}(\text{ADD32}(z3 + z2), 18)$

Repeat for every row!

	Col1	Col2	Col3	Col4
R1	w0'' expa cf921daf	w1'' key 0 8e222826	w2'' key 1 5fa98f47	w3'' key 2 09298bda
R2	w4' key 3 b1a45f5f	w5' nd 3 3320646e	w6' nonce0 aa2c4e02	w7' nonce 1 806f6c02
R3	w8' counter 801f3a52	w9' counter ff5a01f5	w10' 2-by 79622d32	w11' key 4 de23d3d3
R4	w12' key 5 334e1f50	w13' key 6 344cf7af	w14' key 7 a6caaaa2	w15' te k 6b206574

After the 20 Rounds—More diffusion

Feedforward to strengthen diffusion:

$$output_i = (w_i + x_i) \bmod 2^{32}$$

Where:

- x_i = original state word
- w_i = final mixed word

```
def _salsa20_hash(state_words: list[int]) -> bytes:  
    """  
        Apply Salsa20/20 to a 16-word (32-bit) state and return 64 bytes.  
    """  
    assert len(state_words) == 16  
    x = state_words[:]                      # original  
    w = state_words[:]                      # working  
    for _ in range(10):                     # 20 rounds = 10 doublerounds  
        w = _doubleround(w)  
    out = [(w[i] + x[i]) & 0xffffffff for i in range(16)]  
    return b"".join(_u32_to_le_bytes(v) for v in out)
```

Output: 64-Byte Keystream Block

- The original 16 words (4 bytes each: 32 bits) in the matrix are serialized into 64 bytes:
 - $K = \underline{a7d7b4fb5fa98f4709218bdada208c7ade23d3d360f91a50344cf7afa6ccada5}$
- This becomes the keystream block $K[0\dots63]$
- When the plaintext is longer than 64 bytes, Salsa20 increments the counter and generates the next block:
 - Block 0 → keystream bytes 0-63
 - Block 1 → keystream bytes 64-127
 - Block 2 → ... etc ...



Final Encryption Step

- The last step is to XOR:
 - $C[i] = M[i] \oplus K[i]$
- Process of encryption/decryption:
 - a) **Fast:** XOR is one of the **simplest** and **fastest** CPU operations.
 - b) **Reversible:** XOR is its own inverse.
- To decrypt the ciphertext, we need the same keystream (**nonce** and **key**):
 - $M[i] = C[i] \oplus K[i]$



Salsa20—Demo App

Salsa20 GitHub Link: <https://github.com/nmetens/Salsa20?tab=readme-ov-file>

```
#####
# Salsa20 STREAM DEMO
#####
1) Encrypt
2) Decrypt
3) Quit

Enter a menu option (1, 2, or 3): 1
Enter a message to encrypt (blank for 'hello salsa20'): N1cK & N47hAn Ar3 aw350Me!

[+] Key      : a7374fdbbe6c4fbe4063a4619874ad9a8ff1edfd52e8e1060818ecb9805ee9f2c
[+] Nonce    : 77de38dbc0656667
[+] Plaintext : b'N1cK & N47hAn Ar3 aw350Me!'
[+] Ciphertext : 39f3fc637739a43cb040778e2557fa63c9261a2986bedf8bf269

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

How is Salsa20 used?

Some uses of the Salsa20 cipher are now abandoned (upgraded). However, a decade ago, it was used in many ways to securely encrypt user data:

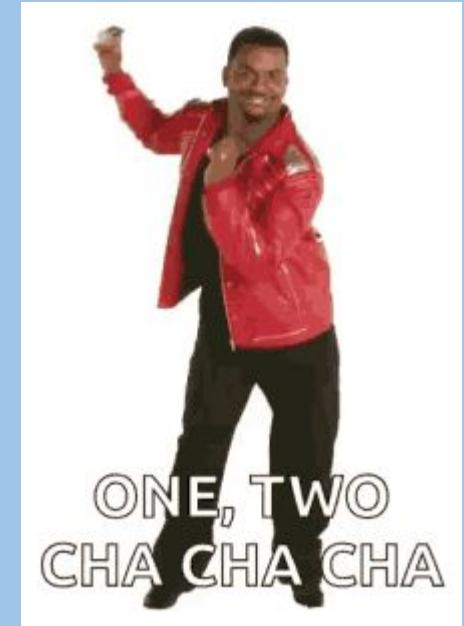
- **Protocols**
 - [DNSCrypt](#): encrypted DNS between a client and a resolver
 - [CurveCP](#): a secure transport protocol
- **Networks**
 - [Yggdrasil](#): a fully end-to-end encrypted network
 - [SAFE](#): A new Secure way to access a world of existing apps where the security of your data is put above all else
- **Password Managers**
 - [KeePassX](#): saves password, usernames, etc. in a database
 - [freepass](#): The free password manager for power users

The Chacha

- An upgraded salsa created in 2008
- The ChaCha was designed by Daniel J. Bernstein to improve:
 - Diffusion per round
 - Increase resistance to cryptanalysis
 - And preserve—and often improve—time per round

It's used more widely today:

- **Protocols**
 - [SSH](#): via chacha20-poly1305@openssh.com
 - [TLS](#): Transport Layer Security
- **Operating systems**
 - [FreeBSD](#): used in random number generation, WireGuard, and OpenSSH
 - [OPNsense](#): an open-source, easy-to-use, and easy-to-build HardenedBSD-based firewall and routing platform
- **Software**
 - [Chrome](#): used in both TLS and QUIC as ChaCha20-Poly1305
 - [Firefox](#): used in TLS
 - [Safari](#): support verified in iOS 12.2 and OS X 10.14.4



Salsa20— Is it broken?

As of now, Salsa20/20 (the 20-round version) **remains unbroken**.

So far, **no published attacks** recover the key or distinguish the keystream faster than brute force (2^{256}).

Attacks on **Salsa20/8** and **Salsa20/7** rely on:

- **Low diffusion:** The first few rounds of Salsa20 don't spread changes very far, so bit differences remain partly predictable.
- **Truncated differentials:** Salsa20/5 could be distinguished from random, or key recovery attempted (though at ridiculously high cost, e.g., 2^{165} operations). - Paul Crowley in 2005
- **Related-cipher scenarios:** If the **same secret key** is reused in two Salsa20 variants (Salsa20/12 and Salsa20/8) with different nonces (IVs), then they can recover the 256-bit key with time complexity around $2^{193.6}$ under ideal conditions. - Lin Ding et al. 2018–2019

These don't apply to 20 rounds. Adding more rounds (or using distinct keys for distinct variants) defeats them.

Every known attack applies only to **reduced-round variants** (≤ 12 rounds) or to **related-key/related-cipher** scenarios that do not occur in real use.

Takeaway: If you reduce rounds to 8 or so (for speed), you increase risk. For the full 20-round Salsa20, these truncated-differential attacks do *not* apply.

Chacha Improvements

Larger nonce (12 bytes instead of 8): a security advantage, as it allows for more messages to be encrypted with a single key, reducing the risk of nonce reuse, which is a catastrophic security failure.

Greater **diffusion per round** than Salsa20; **does not use more operations**.

A ChaCha quarter-round uses the same ingredients as Salsa20:

- **16 additions, 16 XORs, and 16 fixed rotations** on 32-bit words. The difference is **ordering and repetition**:

ChaCha updates each word **twice** in a quarter-round instead of once:

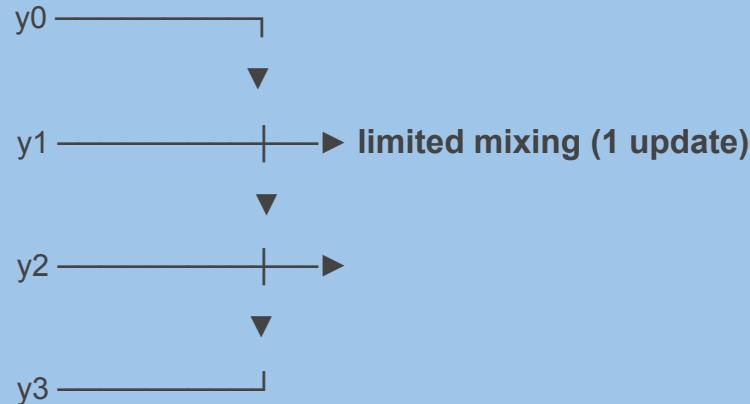
- $a += b;$ $d ^= a;$ $d <<<= 16;$
- $c += d;$ $b ^= c;$ $b <<<= 12;$
- $a += b;$ $d ^= a;$ $d <<<= 8;$
- $c += d;$ $b ^= c;$ $b <<<= 7;$

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

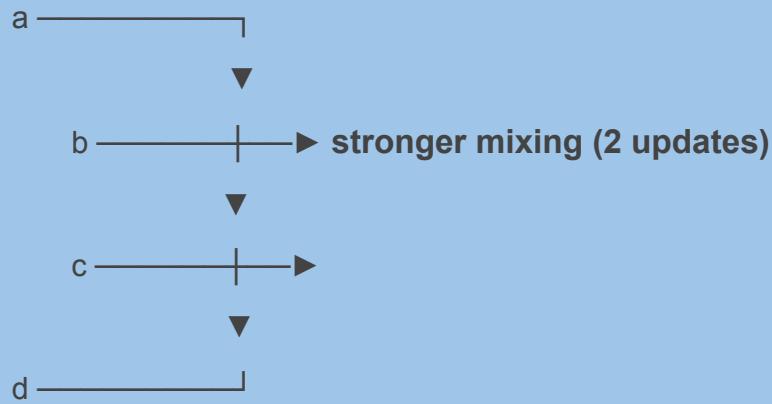
ChaCha's quarter-round spreads differences across the state **faster and more uniformly** than Salsa20's quarter-round, which results in **stronger diffusion** per round, producing faster avalanche and better resistance to low-round differential attacks.

Salsa20 vs. ChaCha20

Salsa20 Quarter-Round



ChaCha Quarter-Round



Result: lower diffusion per round.

- A change in y_0 affects only a few nearby words after one round.

Result: higher diffusion per round.

- A change in "a" affects all 4 words more rapidly.

Conclusion

- Salsa20 is a **very fast stream cipher** that generates a pseudo-random key stream that is XORed with the plaintext to produce the ciphertext.
 - **ciphertext = plaintext ⊕ Salsa20(key, nonce)**
- The encryption/decryption functions are the same.
- Utilizes a **ARX** design
 - **A**: 32-bit addition
 - **R**: Bitwise rotation
 - **X**: XOR addition
- Salsa20 is still used today:
 - **protocols, networks, and password managers.**
- Its shorter round versions (Salsa20/7 and Salsa20/8) have been cracked, but Salsa20/12 and above are considered safe.
 - Shorter rounds increase speed
 - Longer rounds increase security
- The **Chacha20** increases diffusion per round and does not use more operations. It is more resistant to shorter round differential attacks.



Sources

- Grigg, Ian. “**Salsa20 Usage & Deployment.**” *Ianix*, updated 23 Oct. 2025, <https://ianix.com/pub/salsa20-deployment.html>.
- Ianix. “**ChaCha Usage & Deployment.**” *Ianix*, 2 Oct. 2025, <https://ianix.com/pub/chacha-deployment.html>.
- Nagaraj, Karthikeyan. “**Understanding Salsa20 Encryption: A Comprehensive Guide.**” *System Weakness*, 18 Mar. 2023, <https://systemweakness.com/understanding-salsa20-encryption-a-comprehensive-guide-2023-2d6688889e4>.
- Wikipedia contributors. “**Salsa20.**” *Wikipedia: The Free Encyclopedia*, Wikimedia Foundation, <https://en.wikipedia.org/wiki/Salsa20>. Accessed 25 Nov. 2025.
- “**Salsa20.**” *Libsodium Documentation*, GitBook, https://libsodium.gitbook.io/doc/advanced/stream_ciphers/salsa20. Accessed 25 Nov. 2025.
- Bernstein, Daniel J. **Salsa20 Speed.** University of Illinois at Chicago, 2005, <https://cr.yp.to/snuffle/speed.pdf>.
- Bernstein, Daniel J. **Salsa20 Specification.** 27 Apr. 2005, <https://cr.yp.to/snuffle/spec.pdf>.
- Computerphile. “**ChaCha20: Security and Speed.**” *YouTube*, uploaded by Computerphile, 8 May 2018, <https://www.youtube.com/watch?v=Eqx0pxfn9GY&t=806s>.
- OpenAI. **ChatGPT**, model GPT-5.1, 30 Nov. 2025, <https://chat.openai.com/>
- Shaw, David. “**The Secrets Module.**” *Real Python*, <https://realpython.com/ref/stdlib/secrets/>. Accessed 25 Nov. 2025.
- Amadori, Andrea, et al. “**Improved Related-Cipher Attack on Salsa20 Stream Cipher.**” *IACR Cryptology ePrint Archive*, 2025, <https://eprint.iacr.org/2025/289.pdf>.
- Amadori, Andrea, 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.
- Ferguson, Niels. “**Salsa20 Cryptanalysis.**” *Ciphergoth.org*, <https://www.ciphergoth.org/crypto/salsa20/salsa20-cryptanalysis.pdf>.
- Bernstein, Daniel J. “**ChaCha, a Variant of Salsa20.**” 20 Jan. 2008, <https://cr.yp.to/chacha/chacha-20080120.pdf>.
- Compile7. “What Is the Difference Between ChaCha20-256 vs Salsa20-128.” *Compile7*, <https://compile7.org/compare-encryption-algorithms/what-is-difference-between-chacha20-256-vs-salsa20-128/>