ChaCha20: A Modern Encryption Method

Internship Report

Under the supervision of

Dr. Mohamed Samir

Prepared by

Abdelaleem Baher
Mazen Ahmed

August 26, 2025

Table of Contents

1 Introduction to ChaCha20

ChaCha20 is a stream cipher designed by Daniel J. Bernstein in 2008. It was developed as a variant of Salsa20, with improvements in diffusion and performance. The algorithm focuses on simplicity, speed, and strong cryptographic security.

It operates on a 512-bit internal state, represented as a 4x4 matrix of 32-bit words, and uses only basic operations: modular addition, XOR, and rotation. This class of operations is known as ARX (Add-Rotate-XOR).

2 Use Cases and Adoption

ChaCha20 has been widely adopted in various cryptographic systems due to its efficiency and robustness:

- **TLS 1.3:** Used in web encryption protocols when AES is not optimal.
- **Web Browsers:** Both Chrome and Firefox implement ChaCha20, especially on mobile platforms.
- **OpenSSH:** Uses the ChaCha20-Poly1305 AEAD construction by default.
- **WireGuard VPN:** Relies exclusively on ChaCha20 for its encryption due to its performance and simplicity.
- **Mobile Apps and IoT:** Ideal for low-power and embedded devices thanks to its high performance in software.

3 History of ChaCha20

The development and adoption of ChaCha20 can be traced through several milestones:

- **2005:** Salsa20 submitted to the eSTREAM project by ECRYPT.
- **2008:** ChaCha20 introduced by Daniel J. Bernstein as an improved version of Salsa20.
- **2014:** Google adopts ChaCha20 for Android encryption, specifically in TLS.
- **2015:** The ChaCha20-Poly1305 AEAD mode is standardized in RFC 7539.
- **2020:** ChaCha20 becomes widely used across many security protocols.

4 Why ChaCha?

- Optimized for software implementation without requiring specialized hardware.
- Performs better than AES on mobile and embedded devices.
- Based on constant-time operations, reducing vulnerability to timing attacks.
- Secure against known cryptanalytic attacks, with up to 256-bit security.

5 The ChaCha20 State: The 4x4 Matrix

The internal state of ChaCha20 consists of 16 words (each 32 bits), forming a 512-bit state organized as a 4x4 matrix. The matrix is initialized as follows:

$$\begin{bmatrix} c_0 & c_1 & c_2 & c_3 \\ k_0 & k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 & k_7 \\ \text{ctr} & n_0 & n_1 & n_2 \end{bmatrix}$$

Where:

- First row: Constants from the ASCII string `"expand 32-byte k"`
- Second and third rows: The 256-bit (32-byte) key split into 8 words
- Fourth row: 32-bit block counter followed by 96-bit (12-byte) nonce

6 Core Operations (ARX)

ChaCha20 uses three basic arithmetic and logical operations known collectively as ARX:

- **Addition modulo 2^{32}**
- **Bitwise XOR**
- **Left Rotation**

7 The Quarter-Round Function

At the heart of the ChaCha20 cipher is the quarter-round function. This function operates on four words of the state and consists of a sequence of ARX operations that mix the bits thoroughly.

Each round of ChaCha20 includes both column and diagonal quarter-rounds applied to different subsets of the matrix, ensuring every word is affected.

8 ChaCha20 Flow Diagram

The following figure illustrates the full process of ChaCha20 encryption, from key and nonce initialization to ciphertext generation:

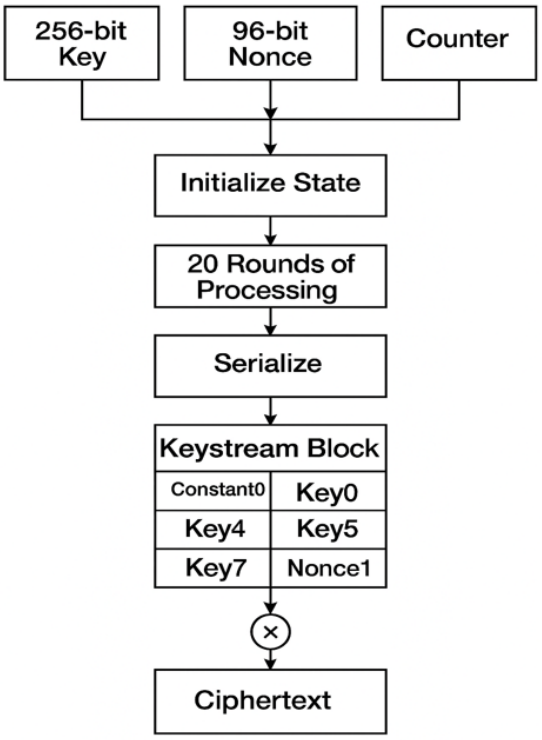

Figure 1: ChaCha20 Encryption Process Flow

9 ChaCha20 vs AES

- **Implementation:** AES requires hardware acceleration for peak performance, whereas ChaCha20 is designed to be fast in software.
- **Security:** Both offer strong encryption, but ChaCha20 avoids timing side channels more easily due to its constant-time nature.
- **Real-World Performance:** On platforms without AES-NI or cryptographic hardware, ChaCha20 significantly outperforms AES.
- **Use Cases:** ChaCha20 is the preferred choice in environments with low processing power or where side-channel resistance is critical.

10 Poly 1305

10.1 Definition

Poly1305 is a cryptographic message authentication code (MAC) algorithm designed by Daniel J. Bernstein in 2005. It's often used with stream ciphers (like ChaCha20) for authenticated encryption, notably in the ChaCha20-Poly1305 AEAD construction (used in TLS, SSH, etc.).

10.2 Purpose

- Authenticate a message by generating a tag (MAC) using a one-time key.
- Ensures the integrity and authenticity of the message.

Ensures the integrity and authenticity of the message.

10.3 Mathematics

- MAC = $(\sum_{i=1}^{n} m_i \times r^{-i}) \times mod(2^{130} - 5)$
- Based on modular arithmetic over a prime field.
- r is a secret, clamped 128-bit key derived from the full key
- m is the message split into 16-byte chunks.
- Requires a 256-bit key: First 128 bits: The r-value (masked to prevent small-subgroup attacks), Second 128 bits: The nonce-dependent part (s-value), added after the polynomial evaluation.

10.4 Usage in AEAD

- Generate a 256-bit one-time key using ChaCha20 with a counter of 0
- Encrypt the plaintext using ChaCha20 with a counter of 1+
- Authenticate the ciphertext and associated data (AAD) using Poly1305 with the one-time key.

10.5 Alternatives

- HPolyC : Combines Poly1305 with HC-128 stream cipher instead of ChaCha20.
- Poly1305-AES : Original design from Bernstein used AES to derive the one-time key.
- VMAC/UMAC : Other high-speed MACs (like VMAC, UMAC) also use universal hashing.

11 ChaCha20 Python Implementations

11.1 Manual Implementation (From Scratch)

```
import struct
import time

def rotate_left(value, shift):
    return ((value << shift) & 0xffffffff) | (value >> (32 - shift))

def quarter_round(a, b, c, d):
    a = (a + b) & 0xffffffff
    d ^= a
    d = rotate_left(d, 16)
    c = (c + d) & 0xffffffff
    b ^= c
    b = rotate_left(b, 12)
    a = (a + b) & 0xffffffff
    d ^= a
    d = rotate_left(d, 8)
    c = (c + d) & 0xffffffff
    b ^= c
    b = rotate_left(b, 7)
    return a, b, c, d

def chacha20_block(key, counter, nonce):
    constants = [0x61707865, 0x3320646e, 0x79622d32, 0x6b206574]
    key_words = list(struct.unpack('<8L', key))
    counter_nonce = [counter] + list(struct.unpack('<3L', nonce))
    state = constants + key_words + counter_nonce
    working_state = state.copy()

    for _ in range(10):
        working_state[0], working_state[4], working_state[8], working_state
            [12] = quarter_round(working_state[0], working_state[4],
            working_state[8], working_state[12])
        working_state[1], working_state[5], working_state[9], working_state
            [13] = quarter_round(working_state[1], working_state[5],
            working_state[9], working_state[13])
        working_state[2], working_state[6], working_state[10], working_state
            [14] = quarter_round(working_state[2], working_state[6],
            working_state[10], working_state[14])
        working_state[3], working_state[7], working_state[11], working_state
            [15] = quarter_round(working_state[3], working_state[7],
            working_state[11], working_state[15])
        working_state[0], working_state[5], working_state[10], working_state
            [15] = quarter_round(working_state[0], working_state[5],
            working_state[10], working_state[15])
        working_state[1], working_state[6], working_state[11], working_state
            [12] = quarter_round(working_state[1], working_state[6],
            working_state[11], working_state[12])
        working_state[2], working_state[7], working_state[8], working_state
            [13] = quarter_round(working_state[2], working_state[7],
            working_state[8], working_state[13])
        working_state[3], working_state[4], working_state[9], working_state
            [14] = quarter_round(working_state[3], working_state[4],
            working_state[9], working_state[14])
```

```
    result = [(working_state[i] + state[i]) & 0xffffffff for i in range(16)]
    return struct.pack('<16L', *result)

def chacha20_encrypt(key, nonce, counter, plaintext):
    assert len(key) == 32
    assert len(nonce) == 12
    keystream = b''
    blocks = (len(plaintext) + 63) // 64
    for i in range(blocks):
        block = chacha20_block(key, counter + i, nonce)
        keystream += block
    return bytes([p ^ k for p, k in zip(plaintext, keystream[:len(plaintext)
        ])])
```

Listing 1: Full Manual ChaCha20 Implementation with File-Based Encryption and Timing

Example Usage:

```
if __name__ == "__main__":
    key = bytes.fromhex("000102030405060708090
        a0b0c0d0e0f101112131415161718191a1b1c1d1e1f")
    nonce = bytes.fromhex("000000000000004a00000000")
    counter = 1

    with open("testing_file.txt", "rb") as f:
        plaintext = f.read()

    start = time.time()
    ciphertext = chacha20_encrypt(key, nonce, counter, plaintext)
    end = time.time()

    with open("encrypted_manual_output.bin", "wb") as f:
        f.write(ciphertext)

    print("Encryption completed.")
    print(f"Time taken: {(end - start) * 1000:.2f} ms")
```

Output:

```
Encryption completed.
Time taken: 4038.99 ms
```

11.2 Simplified Implementation Using `cryptography`

```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms
from cryptography.hazmat.backends import default_backend
import os
import time

key = os.urandom(32)
nonce = os.urandom(16)

with open("testing_file.txt", "rb") as f:
   plaintext = f.read()

start = time.time()
algorithm = algorithms.ChaCha20(key, nonce)
cipher = Cipher(algorithm, mode=None, backend=default_backend())
encryptor = cipher.encryptor()
ciphertext = encryptor.update(plaintext)
end = time.time()

with open("encrypted_crypto.bin", "wb") as f:
   f.write(ciphertext)

print("Cencryption completed.")
print(f"Time taken: {(end - start) * 1000:.4f} ms")
```

Output:

```
Encryption completed.
Time taken: 1.6136 ms
```

12 Performance Comparison: Manual vs Library-Based Implementation

To evaluate the performance of the ChaCha20 encryption algorithm in practice, we tested two Python implementations using a 1MB input file:

- **Manual Implementation:** Our custom-written ChaCha20 encryption from scratch using only standard Python libraries and bitwise operations.
- **Library-Based Implementation:** ChaCha20 encryption using the highly optimized `cryptography` library.

Both versions were tested using the same input file (`testing_file.txt`) and encryption parameters (key, nonce, and counter). The total time taken for the encryption process was recorded using Python's `time` module.

Results

Implementation	Encryption Time (1MB)
Manual (from scratch)	4038.99 ms
Library-based (`cryptography`)	1.61 ms

Analysis

The results demonstrate a significant performance difference:

- The **manual implementation** took over 4 seconds to encrypt 1MB of data, highlighting the cost of using pure Python for computationally intensive operations like ARX (Add-Rotate-XOR).
- In contrast, the **library-based implementation** completed the same task in under 2 milliseconds, showcasing the efficiency of optimized C-backed libraries.

13 Performance Improvement of Manual Implementation

While the original manual implementation of ChaCha20 was functionally correct and cryptographically sound, it suffered from long encryption times when processing large files. For example, encrypting a 1 MB file took:

4038.99 ms

Enhancements Applied

To improve performance, we introduced the following optimizations:

1. **Preallocated Output Buffer:** Instead of incrementally appending to a string (which creates many intermediate copies), we used a pre-allocated `bytearray` to store the ciphertext efficiently.

```
ciphertext = bytearray(len(plaintext))
```

Listing 2: Optimized Keystream Buffer Allocation

2. **In-place XOR Processing:** Rather than performing XOR over slices repeatedly, we processed blocks in-place and stored the result directly into the pre-allocated buffer.

```
for i in range(blocks):
    block = chacha20_block(key, (counter + i) & 0xffffffff, nonce)
    for j in range(min(64, len(plaintext) - i * 64)):
        ciphertext[i * 64 + j] = plaintext[i * 64 + j] ^ block[j]
```

Listing 3: Efficient Block XOR

3. **Modulo Handling of Counter:** To ensure proper 32-bit behavior for the counter, we wrapped it explicitly:

```
(counter + i) & 0xffffffff
```

Listing 4: 32-bit Counter Handling

4. **Avoided Redundant Memory Copies:** We removed unnecessary string concatenation that internally created temporary data on every iteration.

Improved Results

After applying the above enhancements, we re-ran the encryption on the same 1 MB test file and obtained:

2276.56 ms

This represents a speed-up of more than **1.77×**, making our implementationmore efficient while preserving correctness and cryptographic integrity.

14 Disadvantages of ChaCha 20

- AES benefits from widespread hardware acceleration (e.g., AES-NI on Intel CPUs, ARM Crypto Extensions).
- ChaCha20 uses a 512-bit (64-byte) internal state which can make it less memory-efficient for constrained environments (e.g., microcontrollers) compared to more compact ciphers.
- ChaCha20 is a stream cipher, while AES is a block cipher.so we can't use ChaCha20 directly in legacy protocols or modes (like ECB, CBC) that expect a block cipher it Requires redesign or reimplementation of encryption schemes if migrating from AES.
- ChaCha20 provides confidentiality only, not integrity or authenticity so it needs to be paired with a MAC (like Poly1305) in AEAD constructions.
- Mistaken use without proper MAC can lead to vulnerabilities.

15 Side Channel Attacks

Side channel attacks (SCAs) are implementation level attacks that target the physical properties of a cryptographic system rather than its underlying algorithmic security. They exploit unintentional leakage of sensitive information through physical behaviors such as power consumption, execution time, electromagnetic emissions, or memory access patterns.

Types of Side Channel Leakages

- **Timing Attacks:** Measure variations in operation time to infer secret dependent behavior.
- **Power Analysis:** Monitor changes in power usage to deduce key bits (e.g., via Differential Power Analysis).
- **Electromagnetic Emissions:** Capture EM signals to reverse engineer computations.
- **Cache-Based Attacks:** Observe memory and cache access patterns to leak key information.

- **Fault Injection:** Intentionally induce errors through voltage glitches or lasers to exploit faulty outputs.

ChaCha20 and Its Resistance to SCAs

ChaCha20 is inherently designed to be resistant to several types of SCAs:

- **No Lookup Tables:** Unlike AES, ChaCha20 avoids secret dependent memory access such as S-box lookups.
- **Constant-Time Operations:** Uses only addition, XOR, and bitwise rotation operations that can be executed in constant time.
- **Uniform Control Flow:** ChaCha20 does not contain secret dependent branches, reducing timing variability.

Potential Vulnerabilities

ChaCha20 offers strong theoretical resilience, real-world vulnerabilities may arise:

- **Compiler Optimizations:** May inadvertently introduce variable-timing code unless explicitly disabled.
- **Hardware-Level Attacks:** Power and EM-based attacks may still succeed if hardware lacks physical protections.
- **Key Handling:** Improper key management (e.g., not clearing memory) can expose sensitive data.

Best Practices for Secure Implementation

To maintain strong protection against SCAs when using ChaCha20:

- Use constant time cryptographic libraries such as `libsodium` or `BoringSSL`.
- Avoid compiler optimizations in critical paths.
- Securely erase keys after use and store them in protected memory.
- Evaluate with SCA testing tools (e.g., ChipWhisperer).
- Use shielding or noise injection for hardware protection.

16 References

1. Bernstein, D. J. (2008). *ChaCha, a variant of Salsa20.*
2. Langley, A., & Chang, W.-T. (2015). *ChaCha20 and Poly1305 for IETF Protocols.*
3. Google Security Team (2014). *Adopting ChaCha20-Poly1305 in TLS.*
4. WireGuard. *WireGuard Protocol and Cryptography.*