Master of Information Systems Security
Project Report

ChaCha20

Cryptography techniques

Submitted By,

Mohammed SALIH
mohammed.salih@uit.ac.ma
Computer Science

Bilal NASSER
bilal.nasser@uit.ac.ma
Computer Science

Under the guidance of,

Abderrahim ABDELLAOUI
Professor researcher
National School of Applied Sciences

Hiba BELFQIH
Professor researcher
National School of Applied Sciences

Department of Computer Science
National School of Applied Sciences
Ibn Tofail University

Contents

Introduction

ChaCha20 is a lightweight and performant stream cipher that uses 256-bit keys to encrypt and decrypt data. It was initially designed by Daniel J. Bernstein and published in 2008. It is a variant of the earlier Salsa20 cipher intended to be more resistant to cryptanalysis and side-channel attacks while improving time per round all while increasing diffusion [1]. ChaCha has other variants such as ChaCha8 and ChaCha20.

ChaCha is an addition-rotation-XOR (ARX) based stream cipher that uses modular addition, rotation, and XOR operations, as opposed to other popular block ciphers (such as AES, Camellia, … etc) that use substitution-boxes (S-Boxes). ARX causes lower diffusion and thus requires performing more rounds compared to S-Boxes. However, ARX avoids the need for S-Box lookups, which makes them efficient in software implementation and also avoids timing based side channel attacks [1].

ChaCha20 has become one of the widely adopted variants of the ChaCha cipher family, deployed in a variety of applications such as file encryption, blockchain, transport protocols such as Transport Layer Security (TLS), noise protocol, and also used in Internet of Things protocols such as Constrained Application Protocol (CoAP).

In this report, we will explore the technical aspects of ChaCha20, including mathematical foundation, implementation details, give a security review of ChaCha20, a comparison with other existing algorithms, and finally a benchmark comparison with other encryption algorithms.

1.1 Field of application

Due to its wide popularity gained from its security, performance. ChaCha is used in a wide range of applications:

- **Secure communication:** ChaCha20 is used in secure communication protocols such as Transport Layer Security (TLS), Secure Shell (SSH), and also Virtual Private Networks (VPN) such as WireGuard to encrypt data in transit. It is often used as an alternative to the more commonly used Advanced Encryption Standard (AES) cipher.
- **IoT devices:** ChaCha20 is used in Internet of Things (IoT) devices due to its low computational overhead, high performance on low-power devices, and also because it can be fully implemented in software, unlike ciphers that rely on S-Box. It is often used in wireless communication protocols such as Zigbee and LoRaWAN to encrypt data transmitted between devices.
- **Random number generation:** ChaCha20 is used in random number generation algorithms such as /dev/urandom in Linux to generate unpredictable and cryptographically secure random numbers.
- **Authenticated encryption:** ChaCha20 is used in authenticated encryption modes such as ChaCha20Poly1305 to provide both confidentiality and authenticity of data.

It is often used in messaging applications such as Signal and WhatsApp to encrypt messages end-to-end.

Implementation

In this section, we will explore the technical implementation of ChaCha20. We will exclusively cover in this report the ChaCha20 version as defined by the RFC8439 in [2].

2.1 Quarter Round

The basis of the ChaCha algorithm is the quarter round. It operates on 32-bit unsigned integers. This operation is performed multiple times to generate the keystream used for encryption and decryption.

The following pseudo-code shows how the quarter round operates on four 32-bits unsigned integers (denoted as a, b, c, and d):

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

In the previous pseudo-code, each symbol refers to a specific operation:

- "+" denotes integer addition modulo 2^{32} ($a + b \bmod (2^{32})$).
- "^" denotes bit-wise XOR.
- "<<<" denotes left rotation by n-bits.

The following code is an implementation of the quarter round operation:

```python
def quarter_round(a, b, c, d):
    a = (a + b)
    d = (d ^ a)
    d = (d << 16 | d >> 16)
    c = (c + d)
    b = (b ^ c)
    b = (b << 12 | b >> 20)
    a = (a + b)
    d = (d ^ a)
    d = (d << 8 | d >> 24)
    c = (c + d)
    b = (b ^ c)
    b = (b << 7 | b >> 25)
    return a, b, c, d
```

As an example, given that the vector numbers (a, b, c, and d) are:

```
a = 0x11111111
b = 0x01020304
c = 0x9b8d6f43
d = 0x01234567
```

We can calculate them using the quarter round function that we defined previously:

```
a = int('0x11111111', 16)
b = int('0x01020304', 16)
c = int('0x9b8d6f43', 16)
d = int('0x01234567', 16)
(a, b, c, d) = quarter_round(a, b, c, d)
print("a =", hex(a))
print("b =", hex(b))
print("c =", hex(c))
print("d =", hex(d))
```

We get the following results:

```
a = 0xea2a92f4
b = 0xcb1cf8ce
c = 0x4581472e
d = 0x5881c4bb
```

These results are identic to the tests given in [2].

2.2 Quarter Round for the ChaCha state

Chacha uses a 256-bit key, this means that it does not only have four 32-bit unsigned integers, it does have 16. Each Quarter Round operation is done on four predefined integers.

The 16 32-bit unsigned integers are arranged in a 4x4 matrix indexed from 0 to 15:

```
0   1   2   3
4   5   6   7
8   9   10  11
12  13  14  15
```

If we apply “quarter_round(0, 7, 8, 12)” to the state, this means we will apply the Quarter Round operation for the following integers marked by the asterisk:

```
0*    1   2   3
4     5   6   7*
8*    9   10  11
12*   13  14  15
```

2.3 Double Round

A Double Round is an operation composed of 8 Quarter Rounds, the first four Quarter Rounds are done on columns, and the four last ones are done on diagonal cells as the following:

```
// Odd round
quarter_round(0, 4,  8, 12) // column 1
quarter_round(1, 5,  9, 13) // column 2
quarter_round(2, 6, 10, 14) // column 3
quarter_round(3, 7, 11, 15) // column 4
// Even round
quarter_round(0, 5, 10, 15) // diagonal 1
quarter_round(1, 6, 11, 12) // diagonal 2
quarter_round(2, 7,  8, 13) // diagonal 3
quarter_round(3, 4,  9, 14) // diagonal 4
```

ChaCha20 uses 10 iterations of the Double Round which is equal to 20 rounds.

The whole procedure can be implemented in the following manner:

```
def rounds(state):
    steps = [
        [0, 4, 8, 12],
        [1, 5, 9, 13],
        [2, 6, 10, 14],
        [3, 7, 11, 15],
        [0, 5, 10, 15],
        [1, 6, 11, 12],
        [2, 7, 8, 13],
        [3, 4, 9, 14],
    ]
    for _ in range(10):
        for r in steps:
            state[r] = quarter_round(*(state[r]))
    return state
```

2.4 ChaCha20 block function

As per [2], ChaCha20 takes the following inputs:

- 256-bit key: treated as a concatenation of 16 32-bit little-endian unsigned integers.

- Nonce: Number used once, is a unique value that is combined with the encryption key to produce a unique keystream for each encryption. It is a concatenation of three 32-bit little-endian unsigned integers.
- Block counter: A block counter is used along with the nonce to generate the initial state.

The output is 64 random-looking bytes.

To initialize the ChaCha20 state:

- The first four 32-bit unsigned integers (indexes 0..3) are constants defined to 0x61707865, 0x3320646e, 0x79622d32, and 0x6b206574.
- The next eight 32-bit unsigned integers (indexes 4..11) are taken from the key given as input, the key must be read in little-endian format by 4-byte chunks.
- Word 12 is a block counter. Each block is 64-byte (because as previously noted, the output is a 64-bit slice, block size must be identic). This gives us the possibility to encrypt 256 gigabytes of data maximum.
- Words 13..15 are a nonce. Must not be repeated for the same key to increase output entropy. The 1, 2, and 3 32-bit unsigned little-endian integers in the nonce are respectively the 13, 14, and 15 words of the state.

The following matrix resumes the state layout:

```
cccccccc  cccccccc  cccccccc  cccccccc
kkkkkkkk  kkkkkkkk  kkkkkkkk  kkkkkkkk
kkkkkkkk  kkkkkkkk  kkkkkkkk  kkkkkkkk
bbbbbbbb  nnnnnnnn  nnnnnnnn  nnnnnnnn

c=constant k=key b=blockcount n=nonce
```

The following code is used to initialize the ChaCha20 state:

```
def init(key, nonce):
    # Declaring state
    # Here we are using fixed size int to handle overflow edge cases
    state = np.empty(shape=16, dtype=np.uint32)
    state[:4] = np.array([0x61707865, 0x3320646E, 0x79622D32,
0x6B206574], dtype=np.uint32)
    # 4..11 from the key
    state[4:12] = struct.unpack("<8L", key)
    # block counter starts from 1 and keeps incrementing for every block
    state[12]   = 1
    # nonce
    state[13:]  = struct.unpack("<3L", nonce)

    return state
```

Now we test the state initialization by using the test vector defined in [2]:

```
octet_string = """00:01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:
0e:0f:10:11:12:13:14:15:16:17:18:19:1a:1b:1c:1d:1e:1f"""
octet_string = octet_string.replace(":", "")
key = bytes.fromhex(octet_string)
octet_string = "00:00:00:09:00:00:00:4a:00:00:00:00"
octet_string = octet_string.replace(":", "")
nonce = bytes.fromhex(octet_string)

state = init(key, nonce)
for i in range(0, 4):
    for j in range(0, 4):
        print(hex(state[i * 4 + j]), "\t\t", end="")
    print()
```

We get the following output after executing the previous code snippet:

```
0x61707865    0x3320646e    0x79622d32    0x6b206574
0x3020100     0x7060504     0xb0a0908     0xf0e0d0c
0x13121110    0x17161514    0x1b1a1918    0x1f1e1d1c
0x1           0x9000000     0x4a000000    0x0
```

Which is the same result given in [2].

Now we apply the rounds on the state by calling the rounds() method, after the execution, the following output is returned:

```
0x837778ab    0xe238d763    0xa67ae21e    0x5950bb2f
0xc4f2d0c7    0xfc62bb2f    0x8fa018fc    0x3f5ec7b7
0x335271c2    0xf29489f3    0xeabda8fc    0x82e46ebd
0xd19c12b4    0xb04e16de    0x9e83d0cb    0x4e3c50a2
```

We also do a final vector addition to the initial state and the intermediary state after the 20 rounds to get the final state, as a result, we get the following matrix:

```
0xe4e7f110    0x15593bd1    0x1fdd0f50  0xc47120a3
0xc7f4d1c7    0x368c033     0x9aaa2204  0x4e6cd4c3
0x466482d2    0x9aa9f07     0x5d7c214   0xa2028bd9
0xd19c12b5    0xb94e16de    0xe883d0cb  0x4e3c50a2
```

The previous results are also identic to the results given in [2].

The block function is composed of the following steps:

- Incrementing the block counter.
- Applying the 20 rounds on the initial state.
- Matrix addition of the previous two states.

Which can be implemented like the following:

```
def block_fn(init):
    init[12] += 1
    state     = rounds(init.copy())
    state     = state + init
    return state
```

2.5 Encryption algorithm

ChaCha20 invokes the block function each time to encrypt a block, using the same key and nonce. The resulting state from the block function is then serialized by writing the 32-bit unsigned integers to buffer in little-endian order, this gives us the keystream that will be used to encrypt the plaintext by a simple XOR operation that gives us a ciphertext. If there are extra bits left from the keystream, then it is simply discarded.

In the same manner, we can do decryption, the state returned from the block function is used to be XOR-ed with the ciphertext returning the plaintext.

The encryption and decryption can be implemented in the following manner:

```
def encrypt(state, plaintext):
    # calling block function
    state      = block_fn(state)
    # serializing final state
    stream     = struct.pack("<16L", *state)
    # XOR
    ciphertext = bytes([a ^ b for a, b in zip(stream, plaintext)])
    # returning only the size of plaintext
    return ciphertext[:len(plaintext)]

def decrypt(state, ciphertext):
    return encrypt(state, ciphertext)
```

To verify that our algorithm is working, we run the same tests given in [2]:

```
First state before calling block function:
0x61707865    0x3320646e    0x79622d32  0x6b206574
0x3020100     0x7060504     0xb0a0908   0xf0e0d0c
```

```
0x13121110    0x17161514    0x1b1a1918  0x1f1e1d1c
0x1           0x0           0x4a000000  0x0

First state after calling block function:
0xf3514f22    0xe1d91b40    0x6f27de2f  0xed1d63b8
0x821f138c    0xe2062c3d    0xecca4f7e  0x78cff39e
0xa30a3b8a    0x920a6072    0xcd7479b5  0x34932bed
0x40ba4c79    0xcd343ec6    0x4c2c21ea  0xb7417df0

Second state before calling block function:
0x61707865    0x3320646e    0x79622d32  0x6b206574
0x3020100     0x7060504     0xb0a0908   0xf0e0d0c
0x13121110    0x17161514    0x1b1a1918  0x1f1e1d1c
0x2           0x0           0x4a000000  0x0

Second state after calling block function:
0x9f74a669    0x410f633f    0x28feca22  0x7ec44dec
0x6d34d426    0x738cb970    0x3ac5e9f3  0x45590cc4
0xda6e8b39    0x892c831a    0xcdea67c1  0x2b7e1d90
0x37463f3     0xa11a2073    0xe8bcfb88  0xedc49139

Ciphertext:
6e2e359a2568f98041ba0728dd0d6981e97e7aec1d4360c20a27afccfd9fae0bf91b6
5c5524733ab8f593dabcd62b3571639d624e65152ab8f530c359f0806c818e61f0c61
2402be9758cc2bab0c06a05c0850dff907869ba016e47f2c2b4ae81cbf7fed0cfeae1
286a9b07f1b0b9a175a
```

Comparing previous results with the ones given in [2] are identic, which means our algorithm is functional!

Security

Chacha20 follows the same security principles as salsa20 and has a significant security review such as [3], ChaCha20 has also proven to be more resistant to certain attacks compared to Salsa20 such as [4].

In [1], the authors describe multiple ways that compromise the key used to initiate the ChaCha state. Fault injection is a potential issue for micro-controllers as a series of fault injections on the final addition between the initial state and calculated state (specifically a fault injection on the keywords addition) with an average of 32 fault injections, can reveal the secret key, thus rendering a potential use of ChaCha on an embedded device such as encrypted communication compromised.

Some stream ciphers use a feedback mechanism where the previous ciphertext is also XOR-ed with the current plaintext and keystream to create more diffusion compared to only XOR-ing the plaintext with keystream (what ChaCha20 does actually) [5]. This feedback introduces non-linearity, making the cipher more resistant to various attacks. Using the feedback mechanism in a cipher can offer a higher level of security and be more resistant against known plaintext and chosen plaintext attacks. This is the same reason why S-Box has proven its robustness and enhancement of block-ciphers because S-Box has the ability to confuse potential attackers.

[6] identifies the need for resistance against side channel attacks (besides timing attacks because ChaCha20 is immune to this suite of potential weaknesses) in IoT devices, due to the inherent nature of how IoT devices handle sensitive and critical data. There is an incentive for such devices to be resistant to attacks such as power monitoring which can compromise the key without any signs of physical tempering such as fault injection.

Differential-linear attacks are possible [7] and should be considered depending on the thread model of the application where ChaCha will be deployed. Although they are still currently impractical even for ChaCha variants with reduced rounds (such as ChaCha6 and ChaCha7). To be able to crack a ChaCha6 key, a D-linear attack is defined in [8] with $2^{77.4}$ time complexity and 2^{58} data complexity. According to [9] the world's fastest computer at the time can perform at least 1 quintillion (1^{18}) operations in each second. Thus a ChaCha6 key can be cracked using the method defined in [9] in $T_s = \frac{2^{77.4}}{10^{18}} \approx 199398.39$ seconds which is roughly equal to $T \approx 138.47$ hours.

ChaCha20 like many other ciphers is prone to oracle attacks, in case an attacker gains access to a cryptographic oracle, which is a system that provides data about the encryption or decryption. By analyzing the results given by the oracle, an attacker can exploit this behavior to potentially decrypt other encrypted information, or at worse retrieving the encryption keys.
Oracle attacks are categorized into different types as the following:

- Padding Oracle Attack: In a padding oracle attack, the attacker exploits the behavior of a decryption oracle that reveals whether the padding of a decrypted message is correct or not. By sending specially crafted ciphertexts and analyzing the oracle's responses, the attacker can gradually recover the plaintext message or encryption key.
- Timing Oracle Attack: In a timing oracle attack, the attacker exploits variations in the time taken to process different inputs by the cryptographic system. By measuring the timing differences in the system's responses, the attacker can infer information about the encrypted data or encryption keys.
- Error Oracle Attack: An error oracle attack involves exploiting an oracle that provides information about errors in the decryption process. By sending manipulated ciphertexts and analyzing error messages or responses from the oracle, the attacker can gather information about the encryption scheme or recover sensitive data.
- Chosen-Ciphertext Attack (CCA): In a chosen-ciphertext attack, the attacker can interact with an encryption oracle to encrypt chosen plaintexts of their choice. By analyzing the ciphertexts generated by the oracle, the attacker can gain information about the encryption scheme or potentially recover the encryption key.

Additionally, message forgery can occur if an attacker can manipulate or create ciphertexts that decrypt to a specific chosen plaintext of their choice. This may occur because of a weak key used for encryption, the reuse of the nonce, padding oracle attacks, or cryptanalysis.

In This section, we will explore other cryptography additions that aim to prevent some of the previously mentioned weaknesses present in Chacha20.

3.1 Authentication with Poly1305

Authenticated Encryption with Associated Data (AEAD) modes are recommended approaches to prevent oracle attacks and ensure the authenticity of the encrypted data.

In the case of ChaCha20, it is combined with Poly1305 to guarantee the authenticity of the encrypted data all while preventing against previous attacks.

Poly1305 takes a 32-byte one-time key and a message and produces a 16-byte tag. This tag is used to authenticate the message. the key is divided into two parts, called "r" and "s", they should be unique, and must be unpredictable for each invocation (that is why it was originally obtained by encrypting a nonce), "r" needs to be modified as follows before being used ("r" is treated as a 16-octet little-endian number):

- r[3], r[7], r[11], and r[15] must be smaller than 16, and essential to have the top four bits clear.
- r[4], r[8], and r[12] must be divided by 4, and essential to have the bottom two bits clear.

For the case of "s", it should be random.

The previous constraint can be implemented constraint for “r” like the following:

```
key = os.urandom(32)
r = int.from_bytes(key[0:16], byteorder='little')
r &= 0x0ffffffc0ffffffc0ffffffc0fffffff
```

Poly1305 takes as input:

- 32-byte keys which are “r” and “s”.
- A message to be authenticated.

And as output:

- 16-byte tag.

The whole procedure is executed like the following:

- First, the “r” value is clamped.
- Next, set the constant prime “P” to be $2^{130} - 5$: 3fffffffffffffffffffffffffffffffb.
- An accumulator “a” is declared and set to zero.
- The message is divided into 16-byte blocks.
- Each block is read as a little-endian integer.
- Add one bit beyond the 16-byte limit.
- If the block is less than 17-bytes, we need to pad it to match the previous size.
- Add the previous number to the accumulator “a”.
- Calculate: $a = (r * a)\%P$
- Finally the value of “s” is added to “a” which is returned as the tag as a slice of 16-byte in little-endian format.

The previous algorithm can be implemented like the following:

```
P = 0x3fffffffffffffffffffffffffffffffb # 2^130-5

def ceildiv(a, b):
    return -(-a // b)

def num_to_16_le_bytes(num):
    """Convert number to 16 bytes in little endian format"""
    ret = [0]*16
    for i, _ in enumerate(ret):
        ret[i] = num & 0xff
        num >>= 8
    return bytearray(ret)

def poly1305_tag(key, msg):
    if len(key) != 32:
            raise ValueError("Key must be 256 bit long")
    a = 0
    r = int.from_bytes(key[0:16], byteorder='little')
    # clamping r
```

```
    r &= 0x0ffffffc0ffffffc0ffffffc0fffffff
    s = int.from_bytes(key[16:32], byteorder='little')
    # now we calculate the tag
    for i in range(0, ceildiv(len(msg), 16)):
        n = int.from_bytes(msg[i*16:(i+1)*16] + b'\x01',
byteorder='little')
        a += n
        a = (r * a) % P
    a += s
    return num_to_16_le_bytes(a)
```

To test the previous implementation, we run the given test vectors in [2]:

```
octet_string = """85:d6:be:78:57:55:6d:33:7f:44:52:fe:42:d5:06:a8:01:
03:80:8a:fb:0d:b2:fd:4a:bf:f6:af:41:49:f5:1b"""
octet_string = octet_string.replace(":", "")
key = bytes.fromhex(octet_string)
msg = "Cryptographic Forum Research Group"

tag = poly1305_tag(key, msg.encode())
print(binascii.hexlify(tag).decode())
```

As an output we get:

```
a8061dc1305136c6c22b8baf0c0127a9
```

which is the correct tag for the given message [2].

3.1.1 Chacha20Poly1305

To construct ChaCha20Poly1305 we need the following inputs:

- 256-bit key.
- 96-bit Nonce different for each invocation using the same key.
- A plaintext.
- Arbitrary length associated data (AAD).

The construction of the AEAD using ChaCha20Poly1305 follows the next steps for encryption:

- Generate a one time key using the key and the nonce and by using ChaCha20 itself to generate the key by encrypting a 32-byte input and using its 32-byte result as the key.
- ChaCha20 is then invoked to encrypt the plaintext using the **same key and nonce** used in the previous step. In this step, the block counter is set to 1.
- Then Poly1305 function is called using the previous Poly1305 key, and a message concatenated of the following:

- The AAD.
- Padding up to 15 bytes, which should bring the total size of the message until now to a multiple of 16. It can be zero in length if AAD is already multiple to 16.
- The ciphertext.
- Padding2 again with the same properties. It can be zero in length if ciphertext is already multiple to 16.
- The length of the additional data in octets as a 64-bit little-endian integer.
- The length of the ciphertext in octets (same format as previous).

Decryption is similar but in reverse order:

- Parsing the fields from the input (AAD, ciphertext, AAD size, ciphertext size).
- Running the Poly1305 function on the AAD and the ciphertext, and check if the tag matches the one present as input, if it does not match, the decryption must not be performed.
- Decrypt the ciphertext using ChaCha20.

The following code snippet is the implementation for ChaCha20Poly1305:

```
class ChaCha20Poly1305():
    def __init__(self, key):
        if len(key) != 32:
            raise ValueError("Key must be 256 bit long")
        self.key = key

    def poly1305_keygen(self, key, nonce):
        poly = ChaCha20(key, nonce)
        return poly.encrypt(bytearray(32))

    def pad16(self, data):
        if len(data) % 16 == 0:
            return bytearray(0)
        else:
            return bytearray(16-(len(data)%16))

    def encrypt(self, nonce, plaintext, aad=None):
        if len(nonce) != 12:
            raise ValueError("Nonce must be 96 bit long")
        if aad is None:
            aad = bytearray(0)

        auth_key = self.poly1305_keygen(self.key, nonce)

        cipher = ChaCha20(self.key, nonce)

        # always start from 1
        cipher.block_counter = 1

        ciphertext = cipher.encrypt(plaintext)
```

```
        # | aad | padding aad | ciphertext | padding cipher text | aad
len 8 byte | cipher len 8 byte | tag 16 byte

        auth_msg   = aad
        auth_msg  += self.pad16(aad)
        auth_msg  += ciphertext
        auth_msg  += self.pad16(ciphertext)
        auth_msg  += struct.pack('<Q', len(aad))
        auth_msg  += struct.pack('<Q', len(ciphertext))

        tag        = poly1305_tag(auth_key, auth_msg)

        return ciphertext + tag

    def decrypt(self, nonce, ciphertext, aad=None):
        if len(nonce) != 12:
            raise ValueError("Nonce must be 96 bit long")
        if aad is None:
            aad = bytearray(0)
        if len(ciphertext) < 16:
            # we must have at least ciphertext and tag
            return None

        msg_tag    = ciphertext[-16:]
        ciphertext = ciphertext[:-16]

        auth_key = self.poly1305_keygen(self.key, nonce)

        auth_msg   = aad
        auth_msg  += self.pad16(aad)
        auth_msg  += ciphertext
        auth_msg  += self.pad16(ciphertext)
        auth_msg  += struct.pack('<Q', len(aad))
        auth_msg  += struct.pack('<Q', len(ciphertext))
        tag        = poly1305_tag(auth_key, auth_msg)

        if tag != msg_tag:
            raise ValueError("Invalid tag")

        cipher = ChaCha20(self.key, nonce)
        cipher.block_counter = 1

        return cipher.decrypt(ciphertext)
```

3.2 XChaCha20

XChaCha20 is an IETF draft [10] that has been designed to provide better security. It uses a 192-bit nonce which allows it to be more resistant to nonce reuse attacks compared to the original ChaCha20.

Cipher	**Nonce Length**	**Description**	**Max data (same nonce)**	**Max data (random nonce with average message size of 32 bytes)**
ChaCha20	12 bytes	RFC8439 [2]	256GB	Max 4.3 billion messages
XChaCha20	24 bytes	Draft [10]	256GB	No limitations

Table 1: Comparison between ChaCha20 and it's variant XChaCha20

The following matrix resumes the state layout of HChaCha20 which a transitionary state:

```
cccccccc  cccccccc  cccccccc  cccccccc
kkkkkkkk  kkkkkkkk  kkkkkkkk  kkkkkkkk
kkkkkkkk  kkkkkkkk  kkkkkkkk  kkkkkkkk
nnnnnnnn  nnnnnnnn  nnnnnnnn  nnnnnnnn

c=constant k=key b=blockcount n=nonce
```

The only difference between HChaCha20 and Chacha20 is that the block counter is removed and the nonce size is 128 bit.

To be able to encrypt using XChaCha20 we use the following algorithm:

```
xchacha20_encrypt(key, nonce, plaintext):
    subkey = hchacha20(key, nonce[0:15])
    chacha20_nonce = "\x00\x00\x00\x00" + nonce[16:23]
    blk_ctr = 1
    return chacha20_encrypt(subkey, chacha20_nonce, plaintext, blk_ctr)
```

Comparison with other ciphers

In this section, we will compare ChaCha20 to other known ciphers.

4.1 Salsa20

ChaCha20 is closely related to Salsa20 because both algorithms were developed by Daniel J. Bernstein.

Key differences are:

- Salsa20 has a different way for the Quarter Round procedure. Which is summarized by the following pseudo-code:

```
b ^= (a + d) <<<  7;
c ^= (b + a) <<<  9;
d ^= (c + b) <<< 13;
a ^= (d + c) <<< 18;
```

which does use the same operators as we did discuss in Section 2.2.

- Salsa20 also has a slightly different layout for the initial state for 256-bit key size:

```
cccccccc  kkkkkkkk  kkkkkkkk  kkkkkkkk
kkkkkkkk  cccccccc  nnnnnnnn  nnnnnnnn
pppppppp  pppppppp  cccccccc  kkkkkkkk
kkkkkkkk  kkkkkkkk  kkkkkkkk  cccccccc

c=constant k=key p=position n=nonce
```

- Salsa20 can also operate on 128-bit keys, unlike Chacha20.

Overall, ChaCha20 and Salsa20 are exactly identical except for the differences we cited previously.

4.2 AES, Camellia and Ascon

	ChaCha20	**AES**	**Camellia**	**Ascon-128 Ascon-128a**
Keysize	256 bits	128, 192 or 256 bits	128, 192 or 256 bits	128 bits
Type	Stream cipher	Block cipher	Block cipher	AEAD
Structure	Add–rotate–XOR	Substitution–permutation network	Feistel cipher	Sponge construction
Publication date	2008	1998	2000	2014
Implementation complexity	Simple	Complex	Complex	Simple
Implementation type	Software implementation	Requires hardware	Software/ hardware	Software/ hardware
Speed*	Fast	Fastest	Slow	Medium

Table 2: Comparison between ChaCha20, AES, and ASCON

* According to the benchmarks performed in Section 5.

Benchmark

In this benchmark, we will try to compare ChaCha20 and its variants, to other well established cipher algorithms. The basis of this benchmark will be to determine encryption and decryption speeds by using varying data sizes inputs.

The goal of this benchmarking process is to assess the performance of each cipher in real-world deployments. By conducting this benchmark, we aim to:

- Measure the speed of encryption and decryption for each cipher to determine their efficiency in processing data and maintaining security.
- Identify and highlight any significant differences in processing speeds among the ciphers. This will provide insights for resource utilization and computation overhead.
- Analyze how ciphers balance security features and processing speeds. Taking into account key size, and algorithm complexity.

The following algorithms were chosen for this benchmark to be compared to ChaCha20:

- XChaCha20: the other variant of ChaCha20 with increased nonce size, although it is still an IETF draft, it is already used in popular applications such as WireGuard [11], QUIC [12] protocol used for HTTP3, and Noise protocol.
- Salsa20: An older variant to ChaCha20 which is also an established cipher.
- Rabbit: Another stream cipher introduced in 2004, known for its lightweight software implementation and high performance. It will be a good basis to compare ChaCha20 to, as it is considered one of the fastest ciphers.
- AES256-cbc / AES256-ctr: An already established cipher used in most cryptography libraries, thus most applications rely on this cipher with its different modes.
- Camellia256-cbc: a Feistel cipher that is considered modern and safe for usage in any protocol, it is integrated into the TLS cipher suite and is supported by all modern browsers.

We also decided to include AEADs in this benchmark because they are the foundation of modern cryptography protocols, be it for communication (TLS, DTLS), file-system encryption (LUKS2), etc. The following AEADs were included in this benchmark:

- ChaCha20Poly1305 / XChaCha20Poly1305.
- AES256-gcm
- Ascon128

Different implementations for some specific ciphers/AEADs will be used to check if different libraries may be optimizing their implementation for better performance, and check for potential overheads in a specific implementation.

5.1 Environment

Benchmarks were run on intel i5 9400F machine running the Linux kernel on version 6.6.32 with the following CPU information:

```
Architecture:             x86_64
  CPU op-mode(s):         32-bit, 64-bit
  Address sizes:          39 bits physical, 48 bits virtual
  Byte Order:             Little Endian
CPU(s):                   6
  On-line CPU(s) list:    0-5
Vendor ID:                GenuineIntel
  Model name:             Intel(R) Core(TM) i5-9400F CPU @ 2.90GHz
    CPU family:           6
    Model:                158
    Thread(s) per core:   1
    Core(s) per socket:   6
    Socket(s):            1
    Stepping:             10
    CPU(s) scaling MHz:   97%
    CPU max MHz:          4100.0000
    CPU min MHz:          800.0000
    BogoMIPS:             5799.77
    Flags:                fpu vme de pse tsc msr pae mce cx8 apic sep
mtrr pge mca cmov pat pse36 clflush dts
                           acpi mmx fxsr sse sse2 ss ht tm pbe syscall
nx pdpe1gb rdtscp lm constant_tsc art
                          arch_perfmon pebs bts rep_good nopl xtopology
nonstop_tsc cpuid aperfmperf pni pclm
                          ulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3
sdbg fma cx16 xtpr pdcm pcid sse4_1 s
                          se4_2 x2apic movbe popcnt tsc_deadline_timer
aes xsave avx f16c rdrand lahf_lm abm
                          3dnowprefetch cpuid_fault pti ibrs ibpb stibp
tpr_shadow flexpriority ept vpid ept_
                          ad fsgsbase tsc_adjust bmi1 avx2 smep bmi2
erms invpcid mpx rdseed adx smap clflush
                          opt intel_pt xsaveopt xsavec xgetbv1 xsaves
dtherm ida arat pln pts hwp hwp_notify
                          hwp_act_window hwp_epp vnmi
Virtualization features:
  Virtualization:         VT-x
Caches (sum of all):
  L1d:                    192 KiB (6 instances)
  L1i:                    192 KiB (6 instances)
  L2:                     1.5 MiB (6 instances)
  L3:                     9 MiB (1 instance)
NUMA:
  NUMA node(s):           1
  NUMA node0 CPU(s):      0-5
```

```
Vulnerabilities:
  Gather data sampling:   Vulnerable: No microcode
  Itlb multihit:          KVM: Mitigation: VMX disabled
  L1tf:                   Mitigation; PTE Inversion; VMX conditional
cache flushes, SMT disabled
  Mds:                    Vulnerable: Clear CPU buffers attempted, no
microcode; SMT disabled
  Meltdown:               Mitigation; PTI
  Mmio stale data:        Vulnerable: Clear CPU buffers attempted, no
microcode; SMT disabled
  Reg file data sampling: Not affected
  Retbleed:               Mitigation; IBRS
  Spec rstack overflow:   Not affected
  Spec store bypass:      Vulnerable
  Spectre v1:             Mitigation; usercopy/swapgs barriers and
__user pointer sanitization
  Spectre v2:             Mitigation; IBRS; IBPB conditional; STIBP
disabled; RSB filling; PBRSB-eIBRS Not af
                          fected; BHI Not affected
  Srbds:                  Vulnerable: No microcode
  Tsx async abort:        Not affected
```

Here we also include the different software versions that were used to build the benchmark suite:

- Rust stable toolchain version 1.79.0 (129f3b996 2024-06-10).
- OpenSSL version 3.3.1 4 Jun 2024 (using the OpenSSL Rust bindings library version 0.10.64).
- Glibc version 2.39.
- RustCrypto submodule versions: chacha20 0.9.1, chacha20poly1305 0.10.1, salsa20 0.10.2, rabbit 0.4.1.

5.2 Results

After running the benchmark suite we get the following results.

Input size / Cipher	100B	10KB	1MB	100MB	1GB
OpenSSL chacha20	597.64 ns (159.57 MiB/s)	3.5289 µs (2.7025 GiB/s)	315.58 µs (3.0945 GiB/s)	39.699 ms (2.4599 GiB/s)	396.92 ms (2.5194 GiB/s)
RustCrypto chacha20*	152.74 ns (624.38 MiB/s)	4.0728 µs (2.3416 GiB/s)	419.75 µs (2.3266 GiB/s)	42.905 ms (2.2761 GiB/s)	431.24 ms (2.3189 GiB/s)
RustCrypto xchacha20*	152.09 ns (627.04 MiB/s)	4.0706 µs (2.3428 GiB/s)	415.83 µs (2.3485 GiB/s)	42.413 ms (2.3025 GiB/s)	430.14 ms (2.3248 GiB/s)
RustCrypto salsa20*	178.26 ns (534.98 MiB/s)	15.255 µs (640.17 MiB/s)	1.5507 ms (644.88 MiB/s)	157.05 ms (636.76 MiB/s)	1.5898 s (644.10 MiB/s)
RustCrypto rabbit*	119.08 ns (800.84 MiB/s)	11.550 µs (845.48 MiB/s)	1.1793 ms (847.98 MiB/s)	119.95 ms (833.71 MiB/s)	1.2111 s (845.55 MiB/s)
OpenSSL aes256-cbc	597.28 ns (159.67 MiB/s)	9.8754 µs (988.89 MiB/s)	969.63 µs (1.0072 GiB/s)	102.00 ms (980.36 MiB/s)	1.0378 s (986.75 MiB/s)
OpenSSL aes256-ctr	556.41 ns (171.40 MiB/s)	2.8969 µs (3.2920 GiB/s)	246.77 µs (3.9574 GiB/s)	31.227 ms (3.1273 GiB/s)	311.71 ms (3.2081 GiB/s)
OpenSSL camellia256-cbc	1.2817 µs (74.407 MiB/s)	57.198 µs (170.73 MiB/s)	5.7874 ms (172.79 MiB/s)	585.62 ms (170.76 MiB/s)	5.9899 s (170.95 MiB/s)
OpenSSL chacha20poly1305	963.48 ns (98.982 MiB/s)	5.1423 µs (1.8546 GiB/s)	450.10 µs (2.1697 GiB/s)	54.167 ms (1.8029 GiB/s)	549.60 ms (1.8195 GiB/s)
RustCrypto chacha20poly1305*	1.4375 µs (66.344 MiB/s)	8.9903 µs (1.0608 GiB/s)	790.21 µs (1.2358 GiB/s)	79.393 ms (1.2300 GiB/s)	814.67 ms (1.2275 GiB/s)
RustCrypto xchacha20poly1305*	1.6453 µs (57.962 MiB/s)	9.1585 µs (1.0413 GiB/s)	795.92 µs (1.2270 GiB/s)	79.673 ms (1.2257 GiB/s)	815.08 ms (1.2269 GiB/s)
OpenSSL aes256-gcm	717.52 ns (132.91 MiB/s)	3.0873 µs (3.0890 GiB/s)	245.73 µs (3.9741 GiB/s)	31.017 ms (3.1485 GiB/s)	311.09 ms (3.2145 GiB/s)
RustCrypto ascon128*	532.64 ns (179.05 MiB/s)	31.605 µs (308.99 MiB/s)	3.2175 ms (310.80 MiB/s)	321.89 ms (310.67 MiB/s)	3.2961 s (310.67 MiB/s)

Table 3: Encryption benchmark results.

Input size / Cipher	100B	10KB	1MB	100MB	1GB
OpenSSL chacha20	600.23 ns (158.89 MiB/s)	3.4476 µs (2.7662 GiB/s)	321.70 µs (3.0357 GiB/s)	40.208 ms (2.4288 GiB/s)	407.22 ms (2.4557 GiB/s)
RustCrypto chacha20*	175.36 ns (543.83 MiB/s)	4.1683 µs (2.2879 GiB/s)	415.81 µs (2.3486 GiB/s)	42.480 ms (2.2989 GiB/s)	430.45 ms (2.3232 GiB/s)
RustCrypto xchacha20*	177.43 ns (537.50 MiB/s)	4.0959 µs (2.3284 GiB/s)	420.52 µs (2.3223 GiB/s)	41.970 ms (2.3268 GiB/s)	431.40 ms (2.3180 GiB/s)
RustCrypto salsa20*	210.28 ns (453.52 MiB/s)	15.199 µs (642.52 MiB/s)	1.5484 ms (645.81 MiB/s)	155.28 ms (643.99 MiB/s)	1.5876 s (644.99 MiB/s)
RustCrypto rabbit*	119.00 ns (801.42 MiB/s)	11.501 µs (849.09 MiB/s)	1.1871 ms (842.42 MiB/s)	118.17 ms (846.26 MiB/s)	1.2096 s (846.56 MiB/s)
OpenSSL aes256-cbc	588.61 ns (162.02 MiB/s)	2.9674 µs (3.2139 GiB/s)	247.43 µs (3.9468 GiB/s)	31.272 ms (3.1228 GiB/s)	309.27 ms (3.2334 GiB/s)
OpenSSL aes256-ctr	553.71 ns (172.23 MiB/s)	2.8993 µs (3.2893 GiB/s)	245.75 µs (3.9738 GiB/s)	31.122 ms (3.1378 GiB/s)	309.27 ms (3.2334 GiB/s)
OpenSSL camellia256-cbc	1.3024 µs (73.223 MiB/s)	56.772 µs (172.01 MiB/s)	5.7646 ms (173.47 MiB/s)	583.34 ms (171.43 MiB/s)	5.9587 s (171.85 MiB/s)
OpenSSL chacha20poly1305	995.62 ns (95.787 MiB/s)	5.2413 µs (1.8195 GiB/s)	452.56 µs (2.1579 GiB/s)	54.806 ms (1.7819 GiB/s)	556.95 ms (1.7955 GiB/s)
RustCrypto chacha20poly1305*	1.4466 µs (65.927 MiB/s)	7.2337 µs (1.3184 GiB/s)	630.92 µs (1.5478 GiB/s)	79.905 ms (1.2222 GiB/s)	841.57 ms (1.1882 GiB/s)
RustCrypto xchacha20poly1305*	1.6821 µs (56.696 MiB/s)	7.4667 µs (1.2772 GiB/s)	642.46 µs (1.5200 GiB/s)	79.805 ms (1.2237 GiB/s)	857.94 ms (1.1656 GiB/s)
OpenSSL aes256-gcm	727.75 ns (131.05 MiB/s)	3.1201 µs (3.0566 GiB/s)	247.24 µs (3.9498 GiB/s)	31.320 ms (3.1180 GiB/s)	321.43 ms (3.1111 GiB/s)
RustCrypto ascon128*	435.97 ns (218.75 MiB/s)	31.203 µs (312.97 MiB/s)	3.2231 ms (310.26 MiB/s)	334.14 ms (299.28 MiB/s)	3.4992 s (292.64 MiB/s)

Table 4: Decryption benchmark results.

* Cipher implementation uses encryption in-place without any need for allocating a new memory region.

To simplify the analysis of the previous results, we also decided to represent the encryption and decryption throughput by input size, this will give us a clear vision of the performance for each cipher and AEAD.

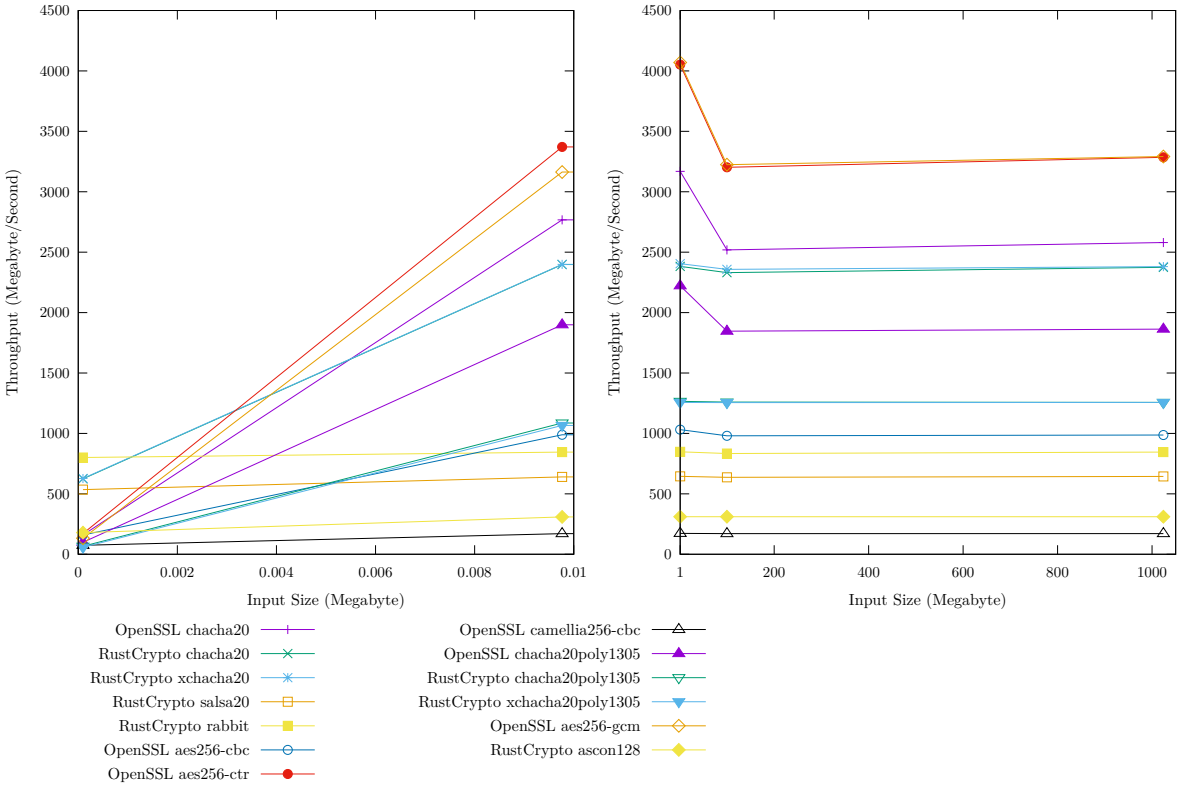

Figure 1: Encryption throughput by input size.

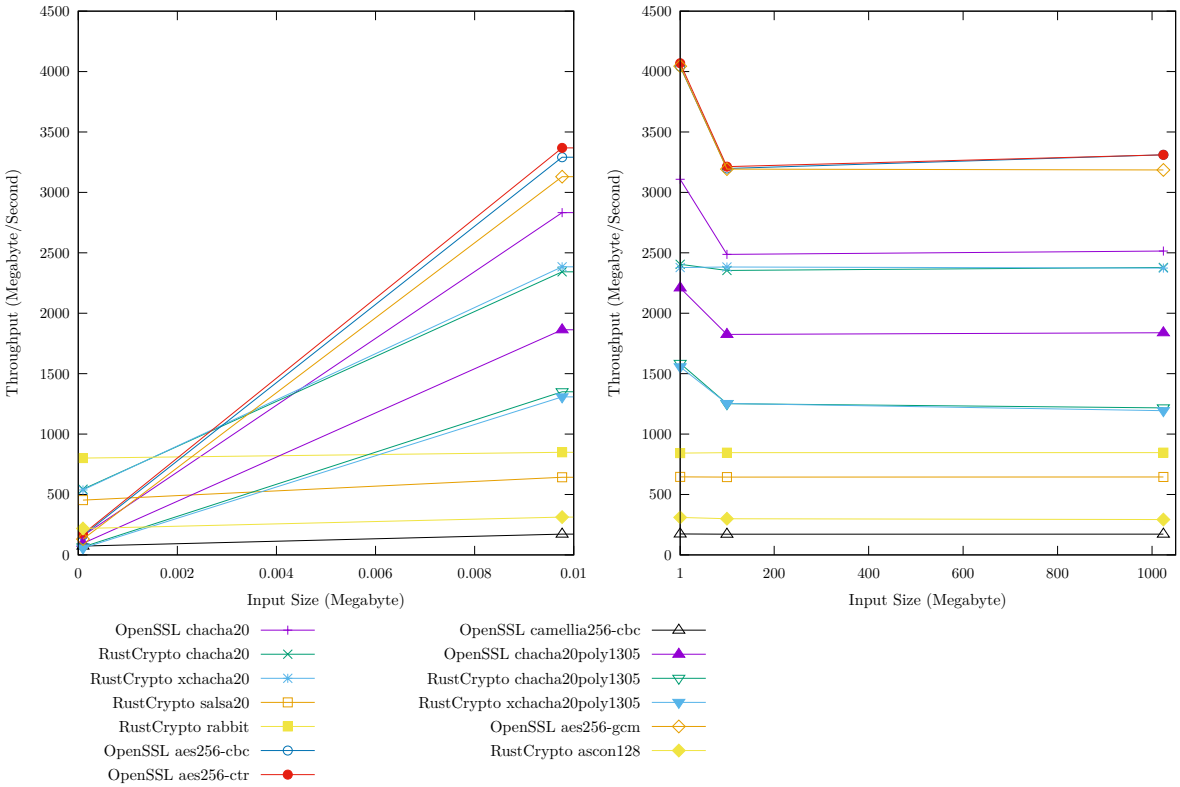

Figure 2: Decryption throughput by input size.

In this benchmark, for each cipher, we tested its encryption and decryption speed and throughput (how much input is processed in a given time) using different input sizes (100B, 10KB, 1MB, 100MB, and 1G).

From the previous graphs, we notice that AES with GCM and CTR modes are the fastest, this is mostly because of the hardware acceleration that AES has by default on the x86_64 platform, which ChaCha20 and its variants do not have.

We also do notice that although ChaCha20 and XChaCha20 (this includes the RustCrypto and OpenSSL implementations) are not the fastest compared to AES, they are still performing better than other stream ciphers like Salsa20 and Rabbit. Salsa20 on the other hand seems to perform worse than other stream ciphers, the only possible explanation is that the current implementation is not optimized enough as its counterpart for ChaCha20.

What is a surprise, is that Ascon-128 is one of the worst performers in this benchmark, due to its lightweight nature, we were expecting higher results for this cipher, however, the results suggest that Ascon-128 although lightweight, would not have the same performance profile like other top performers in this benchmark. And its deployment would be strictly limited to constrained environments without any high performance requirements.

And finally, Camellia has the worst performance in the whole benchmark, it is already standardized to the TLS protocol, and in the same way as AES it can be hardware

accelerated, but most implementations (OpenSSL included) do not support it, thus it can be slow which did lead to these results.

We also include a 100-byte input size to measure overhead (additional resources or processing time required to perform encryption or decryption beyond the actual cryptographic operations), and from the results, we can conclude that it is mostly dependent on the implementation as most of the RustCrypto ciphers seems to have smaller encryption and decryption times compared to OpenSSL ciphers, on the other hand AEADs in RustCrypto seem to have bigger overhead compared to the ones present in OpenSSL.

From all the results, we can conclude that security and performance are independent, ciphers such as AES, ChaCha20, and Camellia (to a lesser degree) are the defacto-standard in security protocols because of their high level of security, and others such as Rabbit and Salsa20 which are older ciphers with at least some theoretical possible attacks and are also known for their high performance still perform worse compared to the previously cited ciphers.

Conclusion

In this report, we explored the implementation of ChaCha20, saw potential security weaknesses and their respective remediation, and reviewed other competing ciphers and their advantages. We finally performed a benchmark to assess the performance of ChaCha20 with other established ciphers.

Overall ChaCha20 stands as a robust cipher that strikes a balance between security, performance, and reliability. It has proven its reliability by the mass adoption it received in different applications, with no known weaknesses at the time of writing.

Benchmarking suite can be found in [13] and our own implementation of ChaCha20 with related demonstration can be found in [14].

Resources

[1] S. Dilip Kumar *et al.*, "A Practical Fault Attack on ARX-Like Ciphers with a Case Study on ChaCha20," in *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2017, pp. 33–40. doi: 10.1109/FDTC.2017.14.

[2] "ChaCha20 and Poly1305 for IETF Protocols." [Online]. Available: https://www.rfc-editor.org/rfc/rfc8439

[3] "Salsa20 security," [Online]. Available: https://cr.yp.to/snuffle/security.pdf

[4] "New Features of Latin Dances: Analysis of Salsa, ChaCha, and Rumba," [Online]. Available: https://eprint.iacr.org/2007/472.pdf

[5] V. R. Kebande, "Extended-Chacha20 Stream Cipher With Enhanced Quarter Round Function," *IEEE Access*, vol. 11, no. , pp. 114220–114237, 2023, doi: 10.1109/ACCESS.2023.3324612.

[6] M. Aamir, S. Sharma, and A. Grover, "ChaCha20-in-Memory for Side-Channel Resistance in IoT Edge-Node Devices," *IEEE Open Journal of Circuits and Systems*, vol. 2, no. , pp. 833–842, 2021, doi: 10.1109/OJCAS.2021.3127273.

[7] N. Ghafoori and A. Miyaji, "Higher-Order Differential-Linear Cryptanalysis of ChaCha Stream Cipher," *IEEE Access*, vol. 12, no. , pp. 13386–13399, 2024, doi: 10.1109/ACCESS.2024.3356868.

[8] C. Beierle *et al.*, "Improved Differential-Linear Attacks with Applications to ARX Ciphers," *Journal of Cryptology*, vol. 35, 2022, doi: 10.1007/s00145-022-09437-z.

[9] "Exascale computers: 10 Breakthrough Technologies 2024." [Online]. Available: https://www.technologyreview.com/2024/01/08/1085128/exascale-computing-breakthrough-technologies/

[10] "XChaCha: eXtended-nonce ChaCha and AEAD_XChaCha20_Poly1305." [Online]. Available: https://datatracker.ietf.org/doc/draft-irtf-cfrg-xchacha/

[11] "WireGuard." [Online]. Available: https://www.wireguard.com/

[12] "QUIC." [Online]. Available: https://en.wikipedia.org/wiki/QUIC

[13] [Online]. Available: https://github.com/m1tk/symmetric_crypto_bench

[14] [Online]. Available: https://github.com/m1tk/chacha20family