# Lightweight Cryptography

TBC, Edinburgh Napier University, Edinburgh, UK.

**Abstract**

The use of cryptography in communication has been around long before computers. Cryptography is used to hide a message being sent from one person to another or from one device to another. Today we use cryptography to protect our privacy and data. Over the years, computing power has increased, as per Moore's law, computing power doubles nearly every two years.[1] This causes our devices to be susceptible to various attacks. With the rise of *Internet of Things* (IoT) devices, these devices are communicating with each other over the internet, many of these devices are constrained because of their computing power or because of their power consumption. In this paper, ChaCha20Poly1305 as a lightweight cryptography method has been reviewed.

## 1. Introduction

Lightweight cryptography is the combination of multiple factors: data and privacy protection, low cost, computing power and battery drain.

The main focus is to secure and protect the data and the privacy, this can't be easily achieved without powerful computers.

Servers, desktops, laptops and smart phones have sufficient computing power, RAM and storage to easily use the conventional cryptography such as AES, ECDH and RSA.

Lightweight cryptography was needed to replace conventional cryptography for IoT devices, in [2], in lightweight cryptography a smaller block size is usually used (64 bits or 80 bits), keys are also smaller (less than 90 bits) and less complex rounds (S-Boxes of 4 bits).

To address the issues mentioned previously, many cryptographers use lightweight stream cipher, lightweight block cipher and lightweight hashing. Cryptographers need to take into consideration the software weight, hardware weight and power consumption.

Lightweight cryptography is mainly used in IoT devices, these devices often have very small computing power or processing speed, small memory (RAM and ROM) and low average power consumption.

These constraints can be identified as software (Time), hardware (Gate equivalence) and power consumption. Some of the devices such as *Radio-Frequency Identification* (RFID) are powered using magnetic fields and some other devices such as sensors are battery operated.

The lightweight cryptography requires a method that is as secure as conventional cryptography. To achieve that, some companies created their own secure hardware and some others developed their own authenticated encryption algorithm. But there is always a compromise, that is why it is best to find a good balance between performance, power consumption/drain and gate equivalence.

IoT devices are susceptible to various attacks since they are connected to the internet, attacks such as Denial of Service (DoS), eavesdropping and man-in-the-middle (MITM).

## 2. Related work

In [3] National Institute of Standards and Technology (NIST) outlined some methods which can be used in lightweight cryptography. Some of these methods, are mentioned below along with other known methods, which will be reviewed in this paper. Mainly ChaCha20 as a cipher stream and Poly1305 as a message authentication code (MAC).

- Block ciphers: LED, PRESENT, SKINNY, LEA and TWINE.
- Stream ciphers: ChaCha, Grain, Mickey, SNOW 3G and Trivium.
- Hash functions and MACs: Poly1305, QUARK, SPONGENT, PHOTON, GLUON and Chaskey.

The table below represents the weaknesses or attacks that can be performed on some of the methods:

Table 1: Weaknesses/Attacks :

| | |
|---|---|
| LED | Ad Hoc (12 rounds of LED-64, 32 rounds of LED-128) [4] |
| LEA | None |
| ChaCha | Differential attack (6-7 rounds) [5] |
| Grain | Linear approximations [6] and Dynamic cube testers[7] |
| QUARK | None |
| SPONGENT | Linear Distinguishers (23 rounds)[8] |

## 3. Implementation

### *3.1. Stream cipher*

A stream cipher encrypts one bit or byte of plaintext at a time by using an algorithm.

It generates a keystream from a given key and an initialisation vector (IV).

The key and IV are random, unpredictable and should never be reused.

The stream is XORed (Exclusive OR) with the plaintext to generate the ciphertext.

ChaCha20 is a stream cipher and Poly1305 is a MAC, the combination of the two algorithms forms an Authenticated Encryption with Associated Data (AEAD) algorithm.[9]

AES-GCM is another common AEAD which is used for Transport Layer Security (TLS) connections. In AES-GCM, the cipher in use is Advanced Encryption Standard (AES) and in Chacha20-Poly1305 the cipher is ChaCha20.

### *3.1.1. ChaCha20*

ChaCha is a variant of the stream cipher Salsa, the number 20 after the name is the number of rounds.

AES is the main algorithm used for most of secret key encryption. Presently AES is secure, if this changes in the future, there will be no alternative for AES. In the Transport Layer Security (TLS) protocol version 1.3, ChaCha20-Poly1305 AEAD will be used as a cipher suite. [10]

Not so long ago Chacha20-Poly1305 AEAD was only supported by Chrome browser, today it is being adopted on the web, for example by Firefox and OpenSSL.

AES is fast especially on modern systems and on dedicated hardware. In software implementation, ChaCha is considered faster than AES and it is three times faster with in devices that do not have a dedicated AES hardware.[9]

AES suffers from side channel attacks and timing attacks, this has been demonstrated in[11]. ChaCha was proposed by Google as an alternative to AES.

ChaCha20 uses a 256-bit key, a 96-bit nonce and a 32-bit initial counter parameter with the plaintext, the ciphertext gets generated.

Quarter rounds are primarily used by ChaCha20. Specifically, ChaCha20 uses 80 quarter rounds or 20 rounds. Each quarter round operates four 32-bit integers:

Table 2: Quarter round operation:

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

By iterating the input block with several rounds, defines the core function. Each round consists of four quarter round operations. Where "+" means addition modulo $2^{32}$, "^" means XOR, and "<<< n" means an n-bit left bitwise roll.[9]

ChaCha20 state gets converted from the input by the ChaCha20 block operation. ChaCha20

state consist of a 256-bit key, a 86-bit nonce and a 32-bit initial counter parameter. By operating an XOR of a key stream, little-endian format is used to get the subsequent state of ChaCha20 serialized so it can compose the numbers along with the plaintext. Any generated extra bit are padded with the plaintext.

Table 3: Input Words

| | | | |
|---|---|---|---|
| C | C | C | C |
| K | K | K | K |
| K | K | K | K |
| B | N | N | N |

Table 4: Column quarter round

| | | | |
|---|---|---|---|
| C | C | C | C |
| K | K | K | K |
| K | K | K | K |
| B | N | N | N |

Table 5: Diagonal quarter round

| | | | |
|---|---|---|---|
| C | C | C | C |
| K | K | K | K |
| K | K | K | K |
| B | N | N | N |

Where: C= Constant, K= Key, B= Block Counter of ChaCha20, N= Initialization Vector.[12]
C = [0x61707865, 0x3320646e, 0x79622d32, 0x6b206574]

Listing 1: Quarter Round [13]

```
1  _round_mixup_box = [(0, 4, 8, 12),
2                      (1, 5, 9, 13),
3                      (2, 6, 10, 14),
4                      (3, 7, 11, 15),
5                      (0, 5, 10, 15),
6                      (1, 6, 11, 12),
7                      (2, 7, 8, 13),
8                      (3, 4, 9, 14)]
```

The operation of ChaCha20 summarizes as follows: The formation of a 16 word input matrix requires:

- C has a constant value of 4 words (Block 0 to 3).
- 8 words of key (Block 8 to 11). The key is interpreted as a little-endian number, the key size 256-bit and it is processed in 4-byte chunks.
- 1 word of block counter (Block 12).
- 3 words of nonce (Block 13 to 15), the same key can not has the same nonce.

Each block is 64-byte. 20 rounds are performed for every new 16 words of plaintext. (10 rounds of column and 10 of diagonal rounds). Each round performs the quarter round operation mentioned previously on 4 words.

### *3.1.2. Poly1305*

Poly1305 is a message-authentication code (MAC).

Poly1305 produces a 16-byte tag out of a 32-byte one time key and a message. This one-time authenticator uses the tag to authenticate the message.

D. J. Bernstein the designer of Poly1305, used AES to for the encryption of the nonce. Since AES was only used for that function, then the usage of another function was possible without affecting the functionality of Poly1305.[14]

The generated key is partitioned into two unique parts which are called "r" and "s", this pair should also be unpredictable.

The following is required:
r[3], r[7], r[11], and r[15] are smaller than 16, and the top 4 bits of each mentioned byte are set to 0.

r[4], r[8], and r[12] can be divided by 4, and the bottom 2 bits of each mentioned byte are set to 0.

"r" is a 16-octet little-endian number, it can also be a constant.

"r" and "s" are each of 128 bits, both are unpredictable and unique. They can be generated using pseudorandom.

If the size of the message is 16 bytes, a $17^{th}$ byte of the value of 1 is added, it is treated as little-endian number.

If the size of the message is less than 16 bytes, 0s are added as padding until it is 16 bytes and then the $17^{th}$ byte is added of the of 1.

A constant prime number P and a variable "accumulator" which is set to 0 are needed.

The little-endian number that was got after adding the $17^{th}$ byte, is saved in the accumulator and then the number is multiplied by "r", then the result is stored again in the accumulator.

Each of the operation is mod P, where P equals $2^{130} - 5$

The last 16 bytes are then converted to a little-endian number and then multiplied with the accumulator mod $2^{128}$. The output will be a 16 byte tag of little-endian number.

The code below summarizes the previously mentioned operation of Poly1305.

Listing 2: Tag Creation [13]

```
def create_tag(self, data):
    for i in range(0, divceil(len(data), 16)):
        n = self.le_bytes_to_num(data[i*16:(i+1)*16] + b'\x01')
        self.acc += n
        self.acc = (self.r * self.acc) % self.P
    self.acc += self.s
    return self.num_to_16_le_bytes(self.acc)
```

Since the purpose of using ChaCha20 with Poly1305, the Poly1305 key is generated using ChaCha20. A 512-bit state is the result of the block function with an input of 256-bit key and 96-bit nonce. The key generated will be the same key that will be used for encryption. the first 256 bits of the 512-bit state is used to form the "r" and "s" of the Poly1305 key. The first 128

bits are "r" and the second 128 bits are "s", the remainder of the 512-bit which are 256 bits are discarded.

Listing 3: Using ChaCha20 to generator a Poly1305 key.[13]

```
def poly1305_key_gen(key, nonce):
    """Generate the key for the Poly1305 authenticator"""
    poly = ChaCha(key, nonce)
    return poly.encrypt(bytearray(32))
```

### *3.2. AEAD*

Chacha20-Poly1305 is an authenticated encryption with additional data algorithm (AEAD). AEAD supports two operations: The "seal" and the "open".

- **Seal (Encryption)**

  The input for the seal operation are:

  - A plaintext of an arbitrary length.
  - A 256-bit secret key.
  - Unique 96-bit IV.
  - Arbitrary length additional authenticated data (AAD).[9]

  The key and the IV are used in the "seal" operation to encrypt a plaintext into a cipher of equal length. After the encryption of the data, a secondary key is generated by using the key and optionally the IV. To generate a keyed hash of the individual lengths of each of the AD and the ciphertext, the secondary key is used. Then the hash value is encrypted and appended to the ciphertext. The AD and the ciphertext are each treated as a 64-bit little-endian integer. If the AAD is of a length where it is an integral multiple of 16 bytes, there will be no padding, otherwise up to 15 zero bytes gets added as padding to make the total length so far to an integral multiple of 16.[9]

Listing 4: Seal [13]

```
def seal(self, nonce, plaintext, data):
    if len(nonce) != 12:
        raise ValueError("Nonce must be 96 bit large")

    otk = self.poly1305_key_gen(self.key, nonce)

    ciphertext = ChaCha(self.key, nonce, counter=1).encrypt(plaintext)

    mac_data = data + self.pad16(data)
    mac_data += ciphertext + self.pad16(ciphertext)
    mac_data += struct.pack('<Q', len(data))
    mac_data += struct.pack('<Q', len(ciphertext))
    tag = Poly1305(otk).create_tag(mac_data)

    return ciphertext + tag
```

- **Open (Decryption)**

  The "open" operation is the reverse of "seal".The MAC of the ciphertext and the AD is generated by using the same key and IV. If the value of the appended MAC after the ciphertext does not match, this means the integrity of the ciphertext and the AD can't be verified and it should be discarded. If there is a match, the ciphertext gets decrypted.

Listing 5: Open [13]

```
def open(self, nonce, ciphertext, data):
    if len(nonce) != 12:
        raise ValueError("Nonce must be 96 bit long")

    if len(ciphertext) < 16:
        return None

    expected_tag = ciphertext[-16:]
    ciphertext = ciphertext[:-16]

    otk = self.poly1305_key_gen(self.key, nonce)

    mac_data = data + self.pad16(data)
    mac_data += ciphertext + self.pad16(ciphertext)
    mac_data += struct.pack('<Q', len(data))
    mac_data += struct.pack('<Q', len(ciphertext))
    tag = Poly1305(otk).create_tag(mac_data)

    if not ct_compare_digest(tag, expected_tag):
        raise TagInvalidException

    return ChaCha(self.key, nonce, counter=1).decrypt(ciphertext)
```

## 4. Evaluation

### *4.1. Test Vectors*

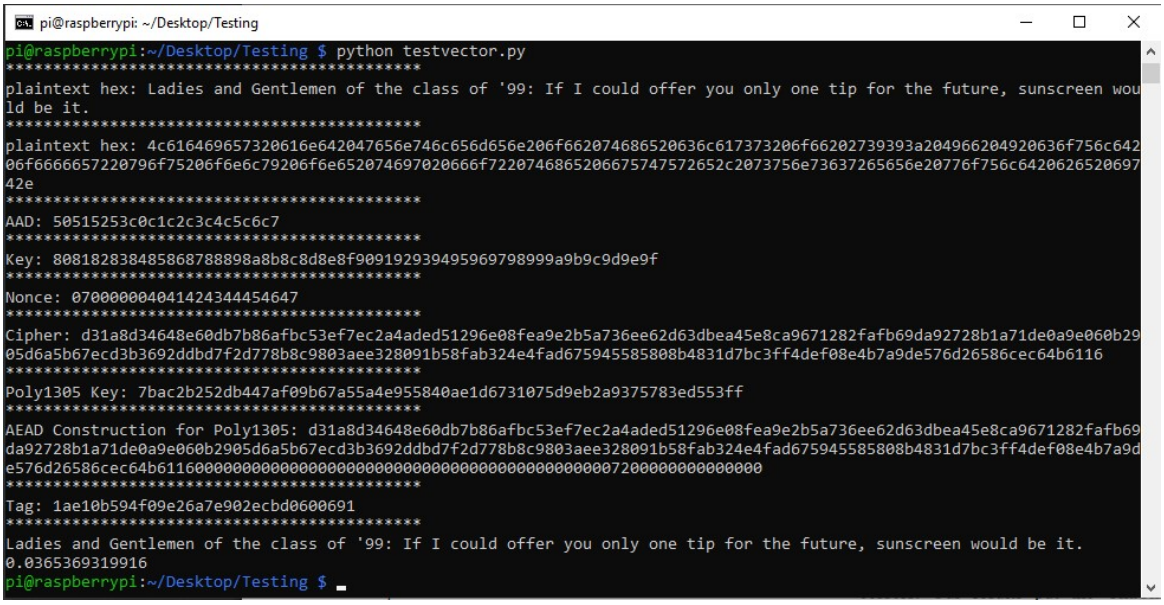


Figure 1: Test Vectors

As we see in Figure 1 the test vectors are identical to the test vectors in [9].

*4.2. Comparison*

Running Chacha20Poly1305 is not enough to evaluate the method. A comparison with another Stream cipher won't be accurate because of the usage of Poly1305. RABBIT[15] is a stream cipher that is being used to be compared with Chacha20 and ChaCha20Poly1305.

Running these methods[16] 1000 times on a Raspberry-Pi while the CPU in standby is at 2% and the memory is at 16.7%. and running the same methods 1000 times on a Linux virtual machine, the CPU in standby is at 1.3% and the memory at 26.2%. The time below is in seconds, to calculate the average value of each encryption and decryption, the total time needs to be divided by 1000.

Table 6: Raspberry-Pi Specifications

| Architecture: | armv7l |
|---|---|
| CPU(s): | 4 |
| Thread(s) per core: | 1 |
| Core(s) per socket: | 4 |
| Socket(s): | 1 |
| CPU max MHz: | 1400.0000 |

Table 7: Linux Virtual Machine Specifications

| Architecture: | x86 / x64 |
|---|---|
| CPU(s): | 2 |
| Thread(s) per core: | 1 |
| Core(s) per socket: | 1 |
| Socket(s): | 2 |
| CPU MHz: | 3200.000 |

Table 8: Results from running the code on the Raspberry-Pi.

| | | | |
|---|---|---|---|
| RABBIT encrypt | Time: 1.9054 | CPU usage: 46.5% | RAM usage: 21.8% |
| RABBIT decrypt | Time: 1.9639 | CPU usage: 45.7% | RAM usage: 21.5% |
| ChaCha20 encrypt | Time: 3.2236 | CPU usage: 46.4% | RAM usage: 21.9% |
| ChaCha20 decrypt | Time: 3.3618 | CPU usage: 45.6% | RAM usage: 22.0% |
| ChaCha20Poly1305 encrypt | Time: 7.6659 | CPU usage: 44.8% | RAM usage: 22.0% |
| ChaCha20Poly1305 decrypt | Time: 7.7284 | CPU usage: 44.7% | RAM usage: 22.1% |

Table 9: Results from running the code on the virtual machine.

| | | | |
|---|---|---|---|
| RABBIT encrypt | Time: 0.1084 | CPU usage: 61.9% | RAM usage: 33.8% |
| RABBIT decrypt | Time: 0.1085 | CPU usage: 63.2% | RAM usage: 33.7% |
| ChaCha20 encrypt | Time: 0.1029 | CPU usage: 56.3% | RAM usage: 33.1% |
| ChaCha20 decrypt | Time: 0.1013 | CPU usage: 66.7% | RAM usage: 33.1% |
| ChaCha20Poly1305 encrypt | Time: 0.2189 | CPU usage: 62.5% | RAM usage: 32.3% |
| ChaCha20Poly1305 decrypt | Time: 0.2239 | CPU usage: 62.5% | RAM usage: 33.7% |

As we can see in the table 8 and 9, ChaCha20Poly1305 is slower than ChaCha20 which in turn is slower than RABBIT. Subsequently these results show that ChaCha20Poly1305 can perform well on a device with limited resources.

## 5. Challenges

There are three main challenges for lightweight cryptography.

- **Hardware:**

  In hardware, the most important metrics are the number of logic gates needed to run the primitive, this will be reflected on how much memory (RAM) is needed. The time of processing data per second is another important metric, also the latency must be taken into consideration, the latency should be less than 50 cycles.

- **Software:**

  In software, the time it takes to run an algorithm is important, this is defined by the number of clock cycles to tread one byte.[17] The amount of memory needed to run the software needs to be considered because after the data is processed, it will be stored in the ROM where there is usually limited space.

- **Power Consumption:**

  Devices are running on battery or powered by electromagnetic energy transmitted from a reader.

## 6. Conclusion

These three challenges combined will make it difficult to strike the right balance between them. If the right balance is not available this may leave the lightweight cryptography weak and vulnerable. By using smaller block and key size this will not be the solution or the "go to" option.

Finding the best lightweight cryptography method will be defined by how secure and resilient this is to known attacks. In the present there is no algorithm which is commonly used on IoT devices. RC5 and PRESENT are two algorithms that have great potential and have been discussed in [18]. ChaCha20Poly1305 can be used on IoT devices to keep the connection between the devices and the internet secure. This method can't be used for every single IoT and other methods will need to be considered depending on the requirements of each device.

## References

## References

[1] G. E. Moore *et al.*, "Cramming more components onto integrated circuits," 1965.

[2] W. J. Buchanan, S. Li, and R. Asif, "Lightweight cryptography methods," *Journal of Cyber Security Technology*, vol. 1, no. 3-4, pp. 187–201, 2017.

[3] K. McKay, L. Bassham, M. Sönmez Turan, and N. Mouha, "Report on lightweight cryptography," National Institute of Standards and Technology, Tech. Rep., 2016.

[4] I. Dinur, O. Dunkelman, N. Keller, and A. Shamir, "Key recovery attacks on 3-round even-mansour, 8-step led-128, and full aes 2," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2013, pp. 337–356.

[5] J.-P. Aumasson, S. Fischer, S. Khazaei, W. Meier, and C. Rechberger, "New features of latin dances: analysis of salsa, chacha, and rumba," in *International Workshop on Fast Software Encryption*. Springer, 2008, pp. 470–488.

[6] C. Berbain, H. Gilbert, and A. Maximov, "Cryptanalysis of grain," in *International Workshop on Fast Software Encryption*. Springer, 2006, pp. 15–29.

[7] I. Dinur and A. Shamir, "Breaking grain-128 with dynamic cube attacks," in *International Workshop on Fast Software Encryption*. Springer, 2011, pp. 167–187.

[8] M. A. Abdelraheem, "Estimating the probabilities of low-weight differential and linear approximations on present-like ciphers," in *International Conference on Information Security and Cryptology*. Springer, 2012, pp. 368–382.

[9] Y. Nir and A. Langley, "Chacha20 and poly1305 for ietf protocols," Internet Requests for Comments, RFC Editor, RFC 8439, June 2018.

[10] E. Rescorla, "The transport layer security (tls) protocol version 1.3," Internet Requests for Comments, RFC Editor, RFC 8446, August 2018.

[11] D. J. Bernstein, "Cache-timing attacks on aes," 2005. [Online]. Available: https://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[12] M. M. Islam, S. Paul, and M. M. Haque, "Reducing network overhead of iotdtls protocol employing chacha20 and poly1305," in *2017 20th International Conference of Computer and Information Technology (ICCIT)*, Dec 2017, pp. 1–7.

[13] D. Klinec, "ph4r05/py-chacha20poly1305," Jun 2018. [Online]. Available: https://github.com/ph4r05/py-chacha20poly1305

[14] D. J. Bernstein, "The poly1305-aes message-authentication code," in *International Workshop on Fast Software Encryption*. Springer, 2005, pp. 32–49.

[15] B. Buchanan, "Rabbit." [Online]. Available: https://asecuritysite.com/encryption/rabbit2

[16] joehad83, "joehad83/coursework," May 2019. [Online]. Available: https://github.com/joehad83/coursework

[17] M.-J. O. Saarinen and D. W. Engels, "A do-it-all-cipher for rfid: Design requirements," *IACR Cryptology EPrint Archive*, vol. 2012, p. 317, 2012.

[18] N. Gunathilake, W. Buchanan, and R. Asif, "Next generation lightweight cryptography for smart iot devices: Implementation, challenges and applications," 04 2019.