Comparison of Common Encryption Algorithms

Matúš Mrekaj

November 23, 2021

Contents

List of Figures

1 Symmetric ciphers

Symmetric ciphers[1] are algorithms where both the encryption and decryption methods use the same key. Thus also sometimes reffered to as symmetric-key algorithms. The interface of such algorithms could be described as follows

$$\text{encrypt(plaintext, key)} \longrightarrow \text{ ciphertext}$$

$$\text{decrypt(ciphertext, key)} \longrightarrow \text{ plaintext}$$

The whole idea behind symmetric ciphers is that, if two parties want to exchange data via an insecure channel without a third party being able to read it, they first have to agree on a secret key and then exchange that key through a different communication channel[2] than the one being used for data exchange (as that channel is insecure and the key would be easily compromised).

Sharing the secret key may seem as a simple task at first but if done naively it may lead to the whole conversation being compromised. For

[1]cipher is an algorithm used for encryption and decryption, `https://en.wikipedia.org/wiki/Symmetric-key_algorithm`

[2]a channel is reffered to as a means of data exchange, a concrete example can be email, internet, verbal communication etc...

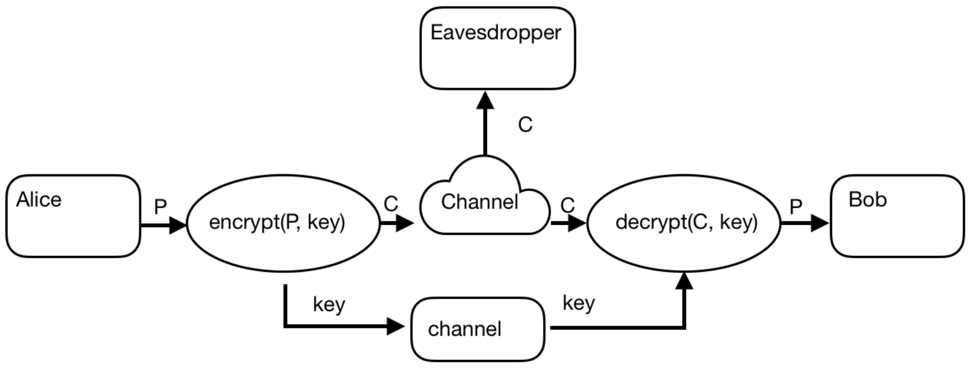

Figure 1.1: system for symmetric algorithms[1]

example if an channel used for the key exchange isn't as secure as thought of, an eavesdropper on that channel could easily listen for the key exchange after which he would be able to decrypt the whole communication between the two parties. What it means for a channel to be insecure or secure is out of scope for this paper. Therefore when we further mention a channel for a key exchange in this chapter we will assume a secure channel[3] is used (i.e a channel where the key is being exchanged securely without the worry of someone eavesdropping[4]).

We distinguish between two types of symmetric ciphers. Namely **Block ciphers** and **Stream ciphers**.[2] Due to the limited scope of this paper we will be only concerned with a few selected ciphers from these types. From the Block ciphers we selected **AES**, as of now it's one of the most broadly used ones, and from Stream ciphers we decided to select **Salsa20/ChaCha** as it's in the eSTREAM portfolio and one of the more studied ones.[27, 28]

[3]a secure channel is a channel where it is safe to transfer confidential data

[4]an eavesdropper is someone who is secretly listening to a private communication, `https://en.wikipedia.org/wiki/Eavesdropping`

1.1 Block ciphers vs Stream ciphers

We won't get much into the details and will only scratch the surface of the differences between these two types of symmetric ciphers, though enough to grasp the core concepts.

1.1.1 Block ciphers

As the name may already suggest, block ciphers operate on blocks[5] of plaintext data. The size of the block (the number of bits) depends on the algorithm being used, though as of now blocks of 64,128,256,512-bits are common. The way block ciphers operate is that given some plaintext data the algorithm will process a single block at a time with the key. Meaning the whole plaintext will be encrypted gradually block by block and each block uses the same key for the encryption. The decryption process is anologous. The ciphertext is processed block by block and being decrypted with the same key.

But there is more to block ciphers than just the data being encrypted/decrypted. Encrypting/Decrypting data block by block with the same key is a naive and insecure method also known as ECB. In ECB mode, information could easily be extracted from the ciphertext as depicted by the following figure. Thus, over the years other modes of operation were invented, namely **CBC, CFB, OFB and CTR**.[18] Though we won't get into the details about any of the modes of operations and thus it is up to the reader to understand the difference between the modes of operation.

[5]a block is a group of bits

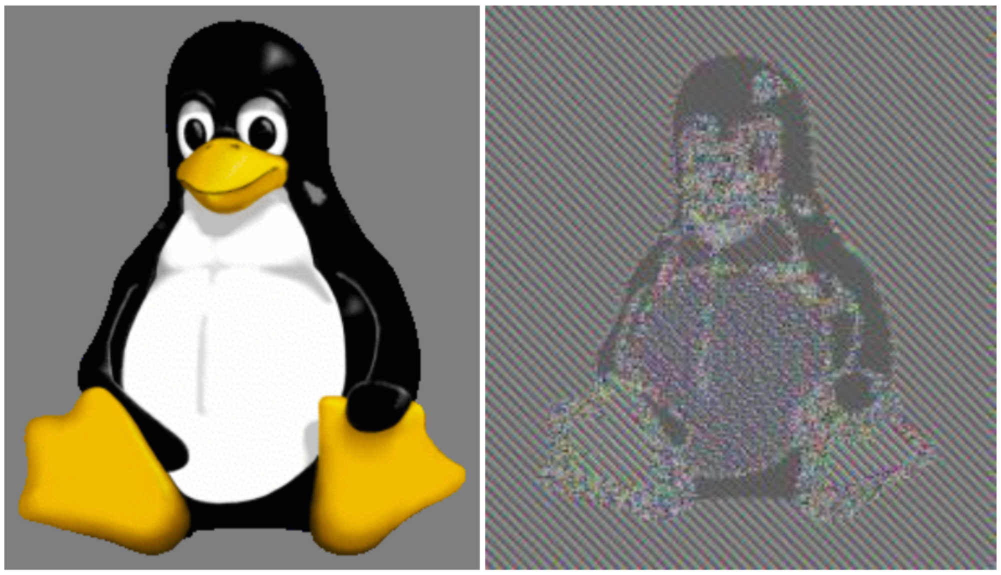

Figure 1.2: block cipher in ECB mode of operation[17]

1.1.2 Stream ciphers

Whereas block ciphers work with blocks of plaintext/ciphertext, stream ciphers work with individual bits hence the name stream ciphers. The notion of a key is also different from the one used in block ciphers. As we are encrypting/decrypting individual bits using the whole key on encrypting a single bit would be rather inefficient and naive. Thus, instead the key is used as a input into the *key stream generator*, where a single bit from the key stream is used to encrypt/decrypt a single bit of data.

In general there exist two types of stream ciphers. Namely synchronous and asynchronous.[34] In synchronous stream ciphers the keystream only depends on a single input, the key as already mentioned earlier. Though in asynchronous stream ciphers, in addition to the key as a input to the keystream, it also depends on the generated ciphertext.[3]

The core part of the stream cipher is its *key stream generator*. The generated bits need to appear to be completely random so that an attacker wouldn't be able to extract any information.[4] Usually the internals of the generator is made of some form of *shift registers* [6], but there also exists other techniques.

LFSR is an example of a shift register commonly mentioned in literatury about stream ciphers but you don't want to use a plain **LFSR** as a keystream. Why you don't want to just use a plain **LFSR**, is because it's insecure due to its linearity and as a result of that known attacks are already known. Thus **LSFRs** are not used by themselfs alone and usually mixed with other techniques.[5, 14]

We will not get into any details about the techniques used in stream ciphers as it is out of the scope of this paper, thus as mentioned in earlier chapters it's up to the reader to research these topics.

[6]a shift register is a circuit using flip-flops where outputs are connected as inputs to the next, `https://en.wikipedia.org/wiki/Shift_register`

1.2 Salsa20

Engineered by Daniel J. Bernstein in 2005 and later submitted to the eSTREAM validation processl. Salsa20 is a stream cipher that at its core uses a hash function in counter mode to produce a keystream that is further XORed with the plaintext to produce the ciphertext. The decryption process is anologous where the only difference is that we use the keystream on the ciphertext to produce the plaintext[29, 30]

1.2.1 The Hash Function

The Salsa20 hash function at its core uses only three operations, namely **addition, xor, rotation** also known as ARX operations. The reason for chosing only these three operation is that a smaller set of operations is able to simulate a bigger, more sophisticated, amount of operations and reaching the same security level.[31]

The input into the hash function are a 256-bit key, 64-bit nonce[7] and a 64-bit block counter. The hash function performs 16 invertible operations over 20 rounds, 320 invertible operations in total.[32]

Algorithm 1: Salsa20(in [16]WORD, out [16]WORD)

```
K ⟵ CLONE(in[..])
currentRound ⟵ 0, maxRounds ⟵ 20
Loop
    if currentRound >= maxRounds then
        break
    end
    Quarterround(K[0], K[4], K[8], K[12])
```

[7]nonce is an abbreviation for number used only once, `https://en.wikipedia.org/wiki/Cryptographic_nonce`

```
    Quarterround(K[5], K[9], K[13], K[1])
    Quarterround(K[10], K[14], K[2], K[6])
    Quarterround(K[15], K[3], K[7], K[11])

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

    currentRound ⟵ currentRound + 2
EndLoop
for i in 0..16 do
    out[i] ⟵ K[i] + in[i]
end
```

Figure 1.3: salsa20 hashing function

1.2.2 The Quarterround

The quarterround function is the core part of the hash function, the part which actually performs the invertible operations with addition, xor and rotation. The function is defined as follows:

Algorithm 2: Quarterround(q_1, q_2, q_3, q_4)

```
$q_2 = q_2 \oplus ((q_1 + q_4) <<< 7)$
$q_3 = q_3 \oplus ((q_2 + q_1) <<< 9)$
$q_4 = q_4 \oplus ((q_3 + q_2) <<< 13)$
$q_1 = q_1 \oplus ((q_4 + q_3) <<< 18)$
```

Figure 1.4: the quarterround function

1.3 ChaCha

ChaCha is also a stream ciher engineered by the same author as Salsa20, Daniel J. Bernstein, back in 2008. [19]

ChaCha isn't a completely different stream cipher, actually it's a variant of the Salsa20. The main purpose of this cipher was to increase the diffusion per each round to decrease the chances for breaking the cipher by cryptanalysis.[20]

We won't get into much detail about this cipher and only mention some differences from the Salsa20 cipher. More in depth explanation can be found in the original paper here [20].

1.3.1 The Quarterround

The main difference is that the quarterround function in ChaCha performs the operations in a completely different order and updates each input twice instead of once. Furthermore also the constant rotation distances changed from 7,9,13,18 to 16,12,8,7 which didn't compromise on security of the cipher and also made it faster on some platforms.[21]

Algorithm 3: Quarterround(q_1, q_2, q_3, q_4)

```
$q_1 += q_2$; $q_4 \oplus= q_1$; $q_4 <<<= 16$
$q_3 += q_4$; $q_2 \oplus= q_3$; $q_2 <<<= 12$
$q_1 += q_2$; $q_4 \oplus= q_1$; $q_4 <<<= 8$
$q_3 += q_4$; $q_2 \oplus= q_3$; $q_2 <<<= 7$
```

Figure 1.5: the quarterround function

1.3.2 The Hash Function

The order of the inputs passed into the quarterround function has also changed as follows:

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

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

Figure 1.6: chacha quarterrounds

I'd like to add that there are far more things to these two ciphers than presented here and these pseudocodes absolutely do not represent the whole cipher. You almost never shouldn't implement a cipher from scratch. Instead look for a library that is being used that already implements these ciphers. The reason being that even a slight implementation error could make the cipher insecure.

1.4 AES

AES is an abbreviation for *Advanced encryption standard.* AES is a block cipher block cipher with support for different kinds of mode of operations. It is a variant of the Rijindael cipher, designed by belgian cryptographers

Vincent Rijmen and Joan Daemen back in 1998 and later submitted to NIST[8]. Furthermore down the line the cipher also became a U.S. federal standard.[16]

Today AES is one of the most popular and widely adopted block ciphers replacing its predecessor DES. DES shouldn't be used at all today mainly due to its 56-bit key, which is ridiculously short for todays standard, neither it's variantion 3DES should be used, due to their insecurity and since already known attacks exists to break DES, 3DES.

1.4.1 Internals of The Algorithm

Rijndael operates on key sizes of 128, 192, 256 bits. Correlated with the key sizes is also the number of rounds the algorithm will perform.[16]

Size	Rounds
128	10
192	12
256	14

While DES is based upon Feistel networks, the core of the algorithm is based upon substitution-permutation networks. In each round of substitution-permutation networks the input is transformed via S-boxes (substitution boxes) and then permutations that results in output bytes. The process is repeated x-rounds after which the ciphertext is reached.

The whole AES internals are depicted by the following figure taken from *Understanding Cryptography* by *Ch. Paar*, *J. Pelz*, which I've included 'cause I think it's a great visualization and I've couldn't it done better myself.

[8]NIST is the abbreviation for National Institute of Standards and Technology

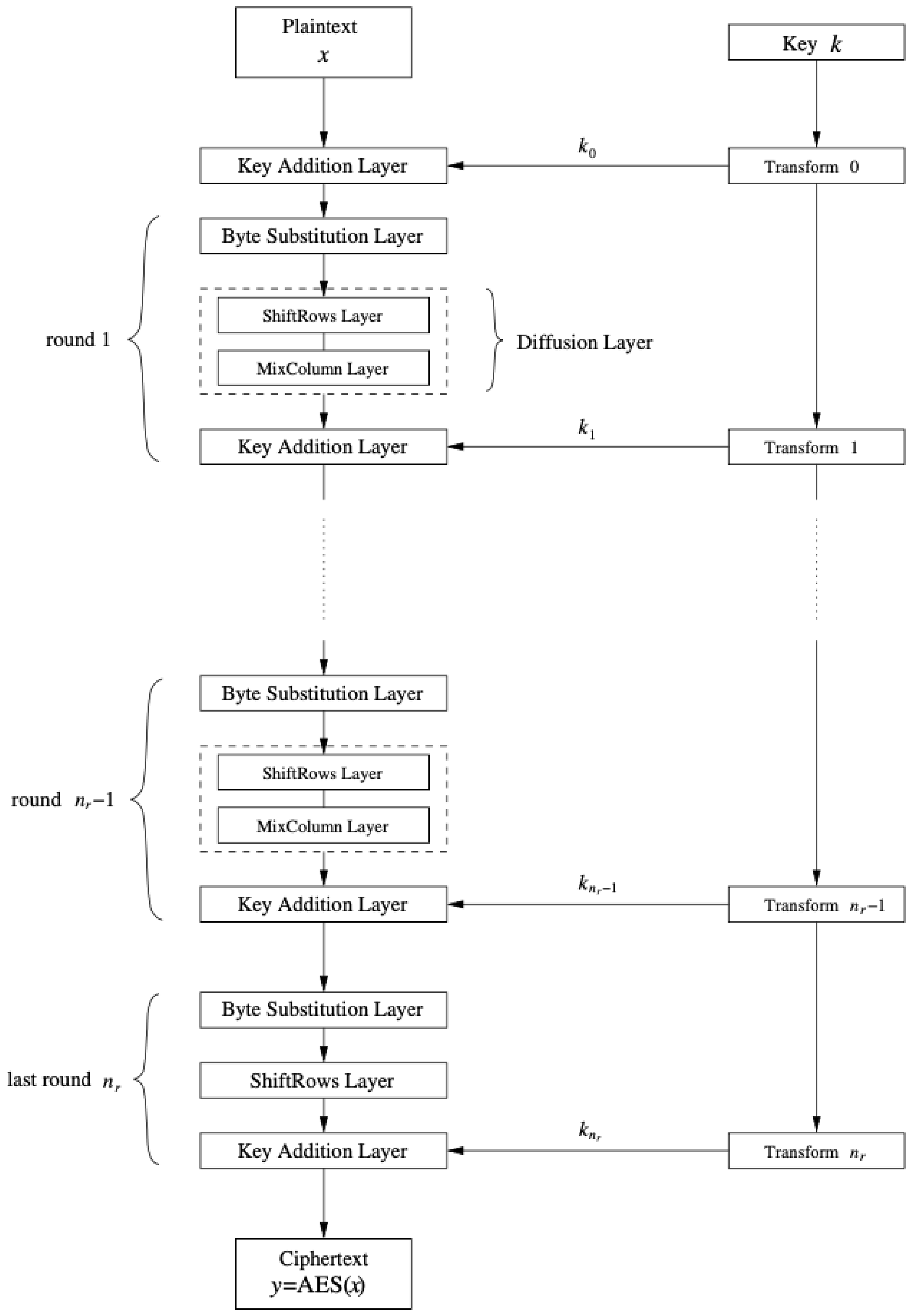

Figure 1.7: AES rounds[6]

The substitution-permutation network works on the so called internal state of AES. The algorithm processes block of 128-bits. Contrary to other algorithms which work with bits, AES works with bytes. Meaning the 128-bits are interpreter as 16 bytes which will be worked with, and can be visualized as in the follwing table:

B_0	B_4	B_8	B_{12}
B_1	B_5	B_9	B_{13}
B_2	B_6	B_{10}	B_{14}
B_3	B_7	B_{11}	B_{15}

1.4.2 Substitution

This would be called S-boxes in other ciphers. Each byte out of the 16 bytes is fed into an S-box or lookup table, depicted in the 2.9 figure and 2.1 equation, which outpus a substitution.

$$Lookup(B_{\mathrm{x}}) = Y_{\mathrm{x}} \tag{1.1}$$

This part of the algorithm provides the non-linear part. The lookup table isn't random and does have a structure in it, though we won't get into the details explaining how the table can be computed as we would also need to cover Galois fields and this is simply out of the scope of this paper.

1.4.3 Permutations

There are two parts to the permutation layer, namely **ShiftRows, Mix-Columns**. If we take back into context the 4x4 matrix, which was mentiond just earlier, ShiftRows just shifts the rows following this formula:

$$ith\_row \longleftarrow ShiftBy(i) \tag{1.2}$$

$$for\ i\ in\ 0..3 \tag{1.3}$$

In the MixColumns part, which is done after the rows have been shifted, we apply a transformation for each column of the matrix. The transformation doesn't changed based on the column and stays same. Thus, each column is multiplied with the following matrix in Galois Fields [16]:

$$\begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \tag{1.4}$$

1.4.4 Key Layer

The inputs into the Key Layer, as depicted in figure 2.7, are the result of the substitution-permunations and the key derived by the *Key Schedule*, which we won't get into in this paper. The result is simply XORed with the current derived key of the round.

1.4.5 Decyption

The decyption process simply applies the inverse of each operation, the permunations, substitution and key addition, to obtain the original data. The whole decryption process is described by the following figure, which again is taken from the book *Understanding Cryptography* by *Ch. Paar*, *J. Pelz*, as I think the visualization is amazing and I couldn't it done better.

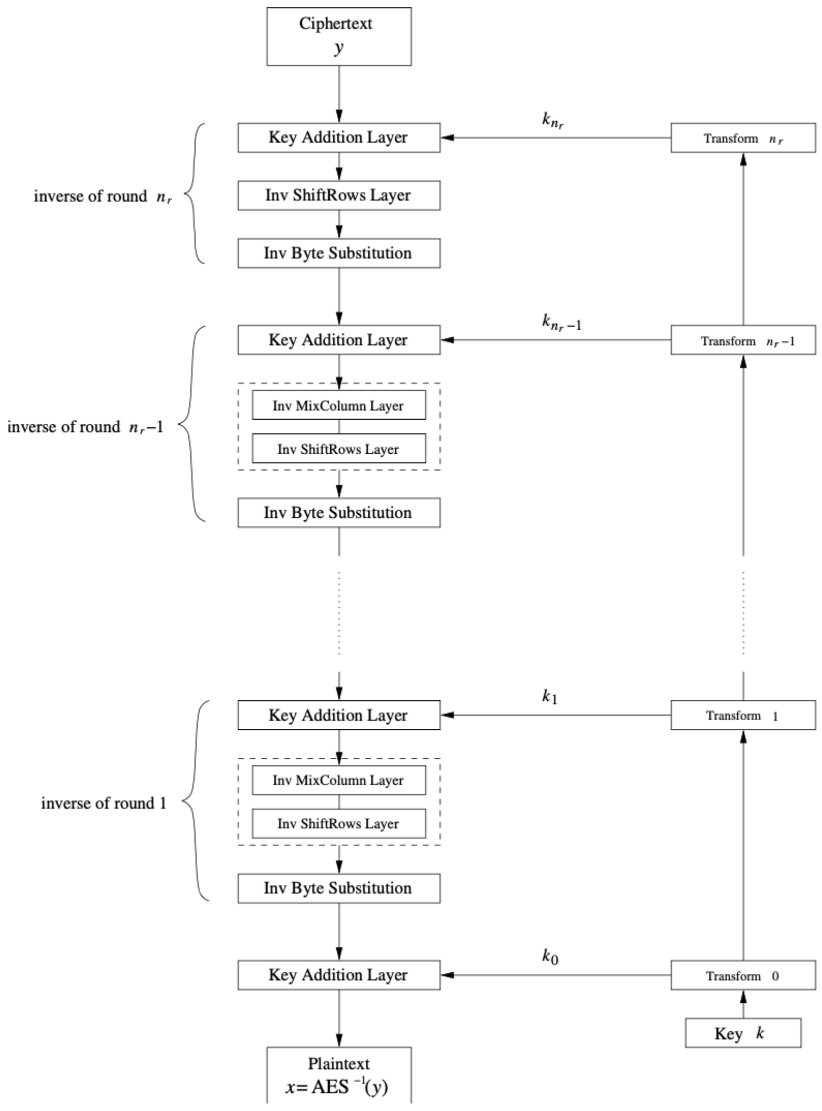

Figure 1.8: AES encryption S-boxes[7]

Starting off with the Key Layer there isn't any change there, as the XOR is the same when encrypting or decrypting. Though, I'd like to emphasize that the key order is reversed in this case. To inverse the MixColumns part of the algorithm we are going to apply the inverse matrice to the one in equation 2.4. Just as when encrypting we are going to apply the inverse to each column of the 4x4 internal AES state. Also just like when encrypting the operations are done in Galois Fields[8, 15].

$$\begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} \tag{1.5}$$

Next, reversing the ShiftRows operation is as simple as when encrypting. We simply just shift the rows in the opposite direction and with the exact amount of places as we shifted them when encrypting.

$$ith\_row \longleftarrow ShiftBy(-i) \tag{1.6}$$

$$for\ i\ in\ 0..3 \tag{1.7}$$

To inverse the substitution layer each byte of the 4x4 internal state matrix is fed into an S-box, just as when we are encrypting. The difference is in the S-box. The lookup in the decryption table is the inverse of the encryption thus yielding the original data. The decryption lookup table is shown in figure 2.10. As calculating the values of the table also involes operations in Galois Fields, same as when encrypting, we won't get into the details of those computations as it is out of the scope for this paper.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Figure 1.9: AES encryption S-boxes[9]

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
A	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
D	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
E	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
F	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

Figure 1.10: AES decryption S-boxes[10]

2 Asymmetric ciphers

Asymmetric ciphers or also known as public-key cryptography work quite different from symmetric ciphers. Contrary to the symmetric ciphers, asymmetric use two keys. One of them is called the private key or secret key and the other one is the public key. The private key must be kept secret at all times and the public key is ment to be shared. These two keys have two different applications. The secret key is used for decryption, while the public key is used for encryption. Thus an interface of such algorithms could be described as follows:

encrypt(plaintext, public_key) $\longrightarrow$ ciphertext

decrypt(ciphertext, secret_key) $\longrightarrow$ plaintext

The idea of asymmetric ciphers is that, if two parties wish to start communicating over an insecure channel they can do so by exchanging their public keys via some asymmetric cipher, thus being able to encrypt and send the ciphertext over an insecure channel and then being able to decrypt the ciphertext with their private keys.

To give some application of public key cryptography. This can be a way to establish a secure channel to exchange the private key between two parties for symmetric ciphers.

While on paper it looks like public key cryptography solves the problem of symmetric keys in exchanging a key securely over an unknown channel in practices it's not that simple. How can we be sure that if we want to establish a message exchange with someone, that their public key is truly theirs and not a forged one? Without getting into the details this problem is solved by something called a $certificate$.[11] Though, certicifates create a whole bunch of other problems, such as their distribution, but we won't get into the details as it is out of the scope of this paper.

Futhermore, public key cryptography also allows the use of digital signatures, identification and other functions.

Digital signatures allow for a third party to verify the authenticity of the data, document or message. You could sign a message with your private key and a third party could verify that that message was sent by you via your public key.

While public key cryptography brings many function into the space it also has downfalls. Mainly that it's slow compared to symmetric ciphers. Most symmetric ciphers, whether block or stream ciphers, outperform asymmetric ciphers on encrypting/decrypting a message.

As of now, there are 3 main mathematical problems on which public-key algorithms are based upon. Namely integer factorization, dicrete logarithms and elliptic curves.[12] In this paper we will only cover **RSA**, which is based upon integer factorization and also it's widely spread in the industry, and we also choose an elliptic curve algorithm namely **Secp256k1**, as I found it interesting mainly due to its usage in Bitcoin.

2.1 RSA

Engineered by Ron Rives, Adi Shamir and Leonard Adleman back in 1977. The name RSA is a short combination of their names.[22] RSA supports encryption/decryption of data and also digital signatures. The problem, or one way function, on which RSA is based upon is integer factorization, as already mentioned.

RSA works with modular arithmetic. Thus, the public key is actually composed of two integers, namely e and n. Whereas the private key has only 1 part the exponent d. N is the divisor (the product of two secret large prime numbers), whereas e is the public exponent to be able to encrypt a message. Since the divisor is publicly available and it's a product of two large prime numbers the security of the algorithm depends on the hardness of the factorization of these two prime numbers.[23] It is thus required for the numbers to be large so it will be computationally infeasible to determine them.

2.1.1 Encryption and Decryption Methods

Actually both methods aren't complicated. Starting off with the encryption, which is just raising to the power e modulus n.

$$\textit{ciphertext} = \textit{Encrypt}(x) = x^{\mathrm{e}} \pmod{n} \tag{2.1}$$

The decryption process is actually the same we just raise the ciphertext to the power of d to obtain the plaintext.

$$\textit{plaintext} = \textit{Decrypt}(y) = y^{\mathrm{d}} \pmod{n} \tag{2.2}$$

With the just above mentioned methods in order to encrypt/decrypt a message we would need to represent is as a number between 1 and $n-1$. In the paper [24] there is a mention that longer messages should be broken up into smaller blocks and processed individually. You shouldn't do this or come up with a clever way to encrypt longer messages as this isn't done in practice and will most likely fire in the wrong direction, we rather exchange a shared key with asymmetric algorithms which is then used to encrypt/decrypt longer messages.

At this point you may be asking yourself how can you compute the private and public keys? We first choose two large prime numbers, namely p and q, and we obtain n by multiplying them. Thus $n = p * q$. We then compute $Phi(n) = (p-1) * (q-1)$. Next. we compute d to be a large integer such that $gcd(d, Phi(n)) = 1$[1]. Atlast, we compute the public exponent e to be the inverse of d modulo $Phi(n)$.[23]

$$e * d \equiv 1 \pmod{Phi(n)} \tag{2.3}$$

In addition for keeping d private, you should also keep the primes p and q and also $Phi(n)$ private.

We won't get into the details about how the mathematics is correct for RSA. Thus for the curios reader we suggest to read the original paper [23].

How to efficiently obtain large prime numbers, or in generall the computation of public/private keys can be read in the original paper [23] and thus we won't get into the details here.

[1]gcd is the greatest common divisor

2.1.2 Signatures

Signatures provides authentication, meaning that since only one part possesses a private key if we receive a valid signature it means that the message originated from that party since no other posseses the private key.

To sign a message the reverse process is applied. To compute the signature we apply the exponent d

$$signature = Decrypt(x) = x^{\mathrm{d}} \pmod n \tag{2.4}$$

After sending the signature to the other party, it can verify that the signature is valid by using the public key

$$x = Encrypt(signature) = signature^{\mathrm{e}} \pmod n \tag{2.5}$$

A signature needs to be message dependent and signer dependent as otherwise we could forge a message or adjust the original message with the signature.[25] Thus if the validation of a signature fails, i.e. we don't get back the original message, we know that the signature is invalid.

2.1.3 Breaking RSA

The just mentioned RSA (also with signatures) is know as *Textbook RSA*, we just explained the core ideas behind the algorithm. It should in no way be used in production, as *textbook RSA* has many flaws that emerged during the years and we strongly adivice the reader to read up upon all the attacks known to *RSA* and to use an well tested library since even a little bug can cause the algorithm to be insecure.

2.2 Secp256k1

2.2.1 Ellipctic Curves

Elliptic curves, just like above mentioned RSA, also have a one way function. Elliptic curves are based upon generalized discrete logarithm problem. And as a result of that discrete logarithm protocols apply to elliptic curves.[13]

Ellipctic curves are defined over finite fields, mostly prime fields. Stricly speaking an ellipctic curve is a polynomial and thus can be expressed by an equation. Current algorithms satisfy the following equation[26]:

$$y^2 \equiv x^3 + a * x + b \pmod{p} \tag{2.6}$$

Tough we'd like to add that this is just a stripped definition, choosing a curve is a hard problem and you most likely shouldn't choose an arbitrary curve as there are requirments to fullfill and should therefore problably used a well known and tested one.

A geometric interpretation of an ellipctic curve is depicted in the following figure

The DL problem upon which ECC[2] is based is really just finding the points on the curve. We can express it with the following equation:

$$P + P + P + ... + P = k * P = T \tag{2.7}$$

The equation means that given a starting point P we add P to itself k times and we obtain the point T. I'd like to note that this is not simple arithmetic addition but addition on the elipctic curve, which is different. The number k is used as a private key and the resulting point on the curve

[2]ECC is short for elliptic curve cryptography

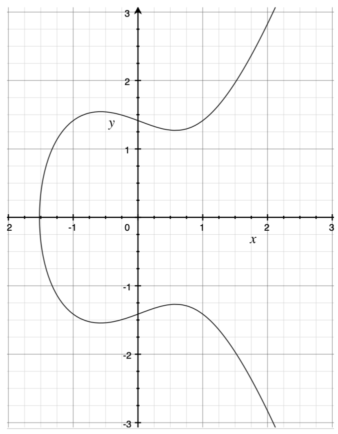

Figure 2.1: elliptic curve with parameters $y^2 = x^3 - x + 2$

T is used as a public key.

To Explain the DL problem in simple terms, we are walking along the curve for an certain amount of steps evantually ending up at a point. And the problem is that reversing the process from the end point is hard.

2.2.2 Encryption and Decryption Methods

Elliptic curves don't have encryption/decryption methods on their own, like RSA. Instead if we want to use ECC algorithms with encryption/decryption we have to rely on protocols like Elliptic Curve Integrated Encryption Scheme (which is a hybrid scheme mixing with symmetric encryption), Elgamal encryption or other protocols, which we won't get into in this paper.

2.2.3 Signatures

The case for signatures is very similar to the encryption/decryption methods in that elliptic curve algorithms don't have signatures on their own and also have to rely on protocols to achieve them. The protocol used for signatures with ECC algorithms is called **Elliptic Curve Digital Signature Algorithm** or ECDSA in short, though it's not the only one there also exists **Edward-curve Digital Signature Algorithm** though as with encryption/decryption methods we won't get into the details in it's up to the reader to reaserch up on these topics.

2.2.4 Secp256k1

With the brief introduction to ECC we come back to the secp256k1 curve. The name secp256k1 refers to just the parameters used for an elliptic curve defined in [33] This curve became popular mainly due to bitcoin using it. It is defined over an finite prime field.

The equation for the secp256k1 curve is as follows[33]:

$$y^2 = x^3 + 7 \tag{2.8}$$

Geometrically interpreted as:

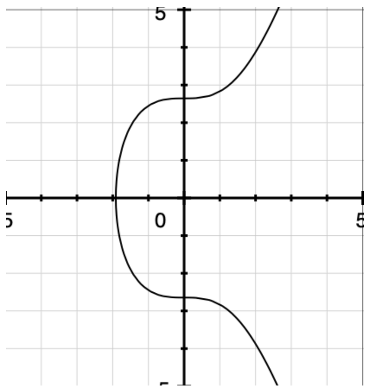

Figure 2.2: secp256k1 elliptic curve

3 Experiments

We decided to test these algorithms in the **Rust** programming language. We only tested the base encryption/decryption operations on sizes 12MB, 56MB, 128MB for symmetric algorithms and encrypting/decrypting 256bits with asymmetric algorihms, as they are mainly used for key exchanges and shouldn't be used for encrypting long files/messages. For RSA only the encryption method was tested as that would be used to encrypt a symmetric key before sending it to the other party. For Secp256k1 we tested creating a shared key via ECDH[1] which can either be used as a key or futher be used as input into a key deriviation function to establish a key for symmetric encryption.

For SECP256K1 we decided to use the `https://crates.io/crates/secp256k1` rust implementation. The implementation for **AES**, **RSA** and **Chacha20** is taken from the **RustCrypto** project `https://github.com/RustCrypto`.

All experiments were performed on a 15-inch, Mid 2015 Macbook Pro with the following specs:

RAM - 16GB 1600MHz DDR3
CPU - 2.5 GHz Quad-Core Inter Core i7

[1]ECDH stands for elliptic curve diffie hellman

CPU Caces:
- L1 Data 32K (x4)
- L1 Instruction 32K (x4)
- L2 Unified 262K (x4)
- L3 Unified 6291K (x1)

3.0.1 benchmarks

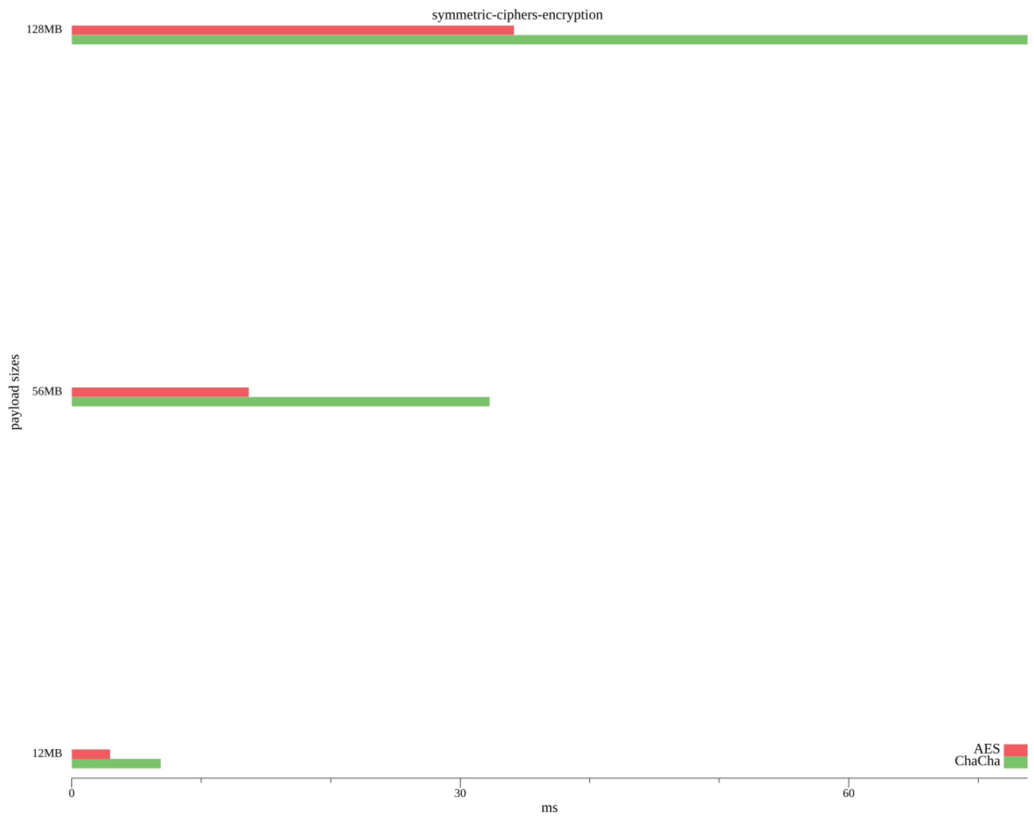

Figure 3.1: symmetric encryption

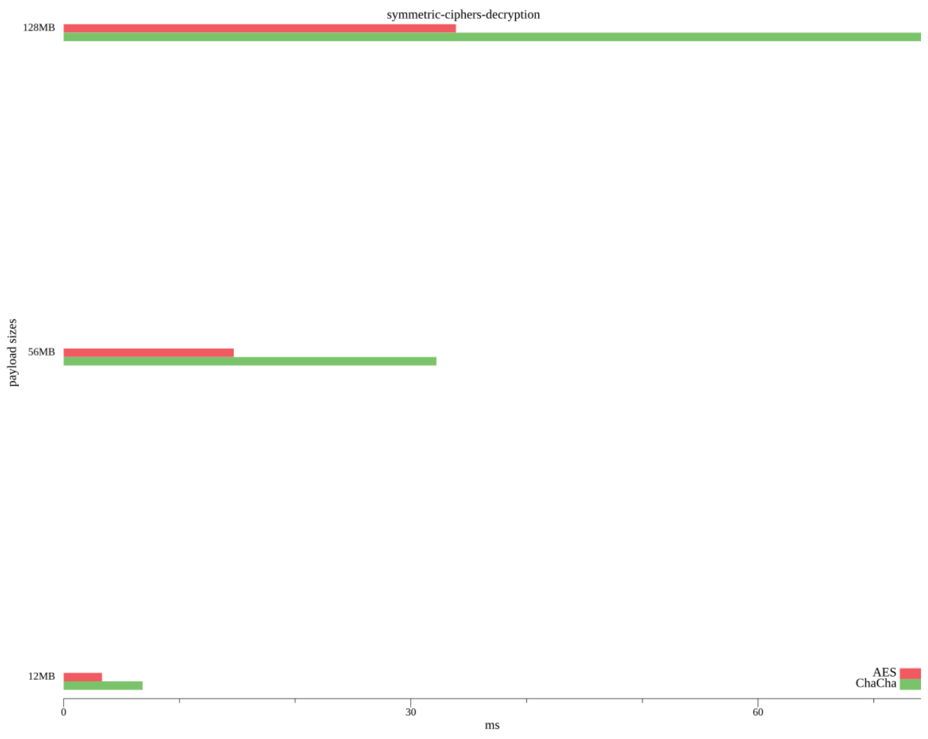

Figure 3.2: symmetric decryption

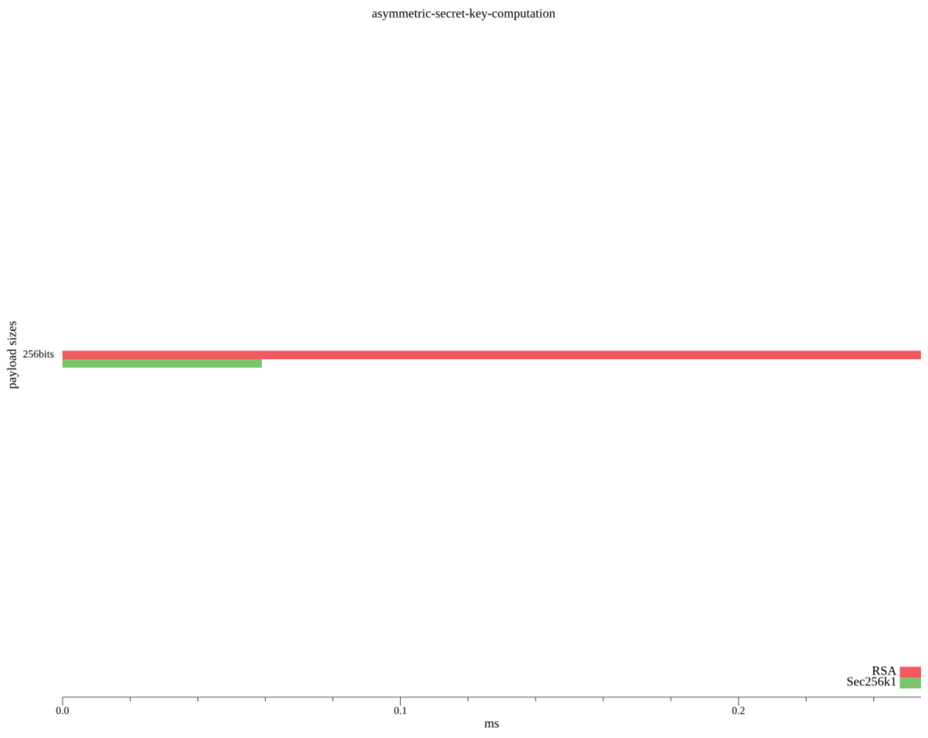

Figure 3.3: asymmetric secret key

4 Conclusion

From the figures we can see that AES is crealy two times faster than chacha. The main reason for this is due to varios hardware optimalizations and that most modern CPUs support native AES instructions, therefore it's expected that AES run on this CPUs will be significantly faster than on others.

Regarding asymmetric ciphers, RSA is roughly 4x slower than ECDH. This was expected as the RSA algorithm involves heavier calculations than ECDH.

Bibliography

[1] Jan Pelzl Christof Paar. *Understanding Cryptography*. Figure 1.5. Springer, 2010, page 5. ISBN: 978-3-642-04100-6.

[2] Jan Pelzl Christof Paar. *Understanding Cryptography*. Springer, 2010, page 29. ISBN: 978-3-642-04100-6.

[3] Jan Pelzl Christof Paar. *Understanding Cryptography*. Stream ciphers paragraph. Springer, 2010, page 30. ISBN: 978-3-642-04100-6.

[4] Jan Pelzl Christof Paar. *Understanding Cryptography*. What Exactly is the Nature of the Key Stream? Springer, 2010, page 34. ISBN: 978-3-642-04100-6.

[5] Jan Pelzl Christof Paar. *Understanding Cryptography*. Springer, 2010, pages 41–46. ISBN: 978-3-642-04100-6.

[6] Jan Pelzl Christof Paar. *Understanding Cryptography*. Figure 4.2. Springer, 2010, page 91. ISBN: 978-3-642-04100-6.

[7] Jan Pelzl Christof Paar. *Understanding Cryptography*. Figure 4.8. Springer, 2010, page 111. ISBN: 978-3-642-04100-6.

[8] Jan Pelzl Christof Paar. *Understanding Cryptography*. Springer, 2010, pages 110–114. ISBN: 978-3-642-04100-6.

[9] Jan Pelzl Christof Paar. *Understanding Cryptography*. Table 4.3. Springer, 2010, page 101. ISBN: 978-3-642-04100-6.

[10] Jan Pelzl Christof Paar. *Understanding Cryptography*. Table 4.4. Springer, 2010, page 114. ISBN: 978-3-642-04100-6.

[11] Jan Pelzl Christof Paar. *Understanding Cryptography*. Springer, 2010, pages 150–155. ISBN: 978-3-642-04100-6.

[12] Jan Pelzl Christof Paar. *Understanding Cryptography*. Springer, 2010, page 155. ISBN: 978-3-642-04100-6.

[13] Jan Pelzl Christof Paar. *Understanding Cryptography*. Springer, 2010, page 239. ISBN: 978-3-642-04100-6.

[14] Jean-Philippe Aumasson. *Serious Cryptography: A Practical Introduction to Modern Encryption*. Feedback Shift Registers. No Starch Press, 2017, pages 136–145. ISBN: 978-1593278267.

[15] Jean-Philippe Aumasson. *Serious Cryptography: A Practical Introduction to Modern Encryption*. Deryption. No Starch Press, 2017, page 111. ISBN: 978-1593278267.

[16] *AES*. URL: `https://en.wikipedia.org/wiki/Advanced_Encryption_Standard` (visited on 11/13/2021).

[17] *Block ciphers ECB mode*. URL: `https://en.wikipedia.org/wiki/File:Tux_ecb.jpg` (visited on 11/08/2021).

[18] *Block ciphers modes of operation*. URL: `https://en.wikipedia.org/wiki/ISO/IEC_10116` (visited on 11/08/2021).

[19] *ChaCha20*. ChaCha variant. URL: `https://en.wikipedia.org/wiki/Salsa20#cite_note-6` (visited on 11/10/2021).

[20] *ChaCha20*. URL: `https://cr.yp.to/chacha/chacha-20080128.pdf` (visited on 11/10/2021).

[21] *ChaCha20*. ChaCha quarter-round. URL: `https://cr.yp.to/chacha/chacha-20080128.pdf` (visited on 11/10/2021).

[22] *RSA*. URL: `https://en.wikipedia.org/wiki/RSA_(cryptosystem)` (visited on 11/17/2021).

[23] *RSA*. URL: `https://people.csail.mit.edu/rivest/Rsapaper.pdf` (visited on 11/17/2021).

[24] *RSA*. Our Encryption and Decryption Methods. URL: `https://people.csail.mit.edu/rivest/Rsapaper.pdf` (visited on 11/19/2021).

[25] *RSA*. Signatures. URL: `https://people.csail.mit.edu/rivest/Rsapaper.pdf` (visited on 11/19/2021).

[26] *RSA*. URL: `https://en.wikipedia.org/wiki/Elliptic-curve_cryptography#cite_note-7` (visited on 11/19/2021).

[27] *Salsa20*. URL: `https://cryptopp.com/wiki/Salsa20` (visited on 11/07/2021).

[28] *Salsa20*. URL: `https://en.wikipedia.org/wiki/Salsa20` (visited on 11/07/2021).

[29] *Salsa20*. first paragraph. URL: `https://cryptopp.com/wiki/Salsa20` (visited on 11/07/2021).

[30] *Salsa20*. URL: `https://cr.yp.to/snuffle/spec.pdf` (visited on 11/09/2021).

[31] *Salsa20*. Section 2. URL: `https://cr.yp.to/snuffle/design.pdf` (visited on 11/09/2021).

[32] *Salsa20*. Section 4. URL: `https://cr.yp.to/snuffle/design.pdf` (visited on 11/09/2021).

[33] *Secp256k1*. URL: `https://www.secg.org/sec2-v2.pdf` (visited on 11/20/2021).

[34] *Stream ciphers*. Types section. URL: `https://en.wikipedia.org/wiki/Stream_cipher` (visited on 11/09/2021).