Chapter 2
Pratical Examples of Security Techniques

In this chapter, we are presenting security algorithms and solutions which are currently considered as secure. All functions, algorithms, and computations assume Big-Endian byte order, unsigned integer usage (size depends on the algorithm) and standard wrapping behavior of unsigned integer numbers. All arrays and tables are indexed from 0.

2.1 Advanced Encryption Standard

Advanced Encryption Standard (AES) is a symmetric block cipher announced by National Institute of Standards and Technology in 2001 [34]. It can be used as a *CIPHER* algorithm presented in symmetric block ciphers modes in sec.(1.2.3). Operations are made on 128 bits blocks and possible key sizes are 128, 192 or 256 bits [34]. AES is considered as a secure standard which can be used for network communication. Before presenting the algorithm itself, we are introducing appropriate notation and information in sec.(2.1.2).

2.1.1 AES in Clouds

AES is one of the most efficient symmetric algorithms [73]. Thus authors in [98] claimed, that Advanced Encryption Standard is very suitable for Cloud environments. Authors point out that [98]:

- AES can perform efficiently on software platforms as well as on hardware platforms (including 8 and 64 bits).

- Using AES results in very high performance in software usage because of ease of parallelization of computations and parallelization of instructions.

- AES is suitable for restricted-space environments due to its less memory allocation than other ciphering schemes.

- No proved cryptoanalysis attacks on AES cryptosystem were made. What's more, AES potentially supports any key and block size greater than 128 bits.

Authors in [73] performed tests on 128 bit AES used under simulated Cloud environment. Their conclusion was: *The performance evaluation shows that AES cryptography can be used for data security*.
Authors in [13] present complex studies on how AES can be used for secure data storage in Cloud environments. They investigated security, speed, and delay which appears when AES is used. They conclude that usage of AES allows to increase confidentiality, authenticity and access control. The only drawback was that larger files to encrypt were increasing the delay.
AES is also used in commercial Clouds:

- Google: *Data stored in Google Cloud Platform is encrypted at the storage level using either AES256 or AES128* [42].

- Amazon Web Services: *AWS Key Management Service uses the Advanced Encryption Standard (AES) algorithm in Galois/Counter Mode (GCM) with 256-bit secret keys* [12].

- Microsoft OneDrive: *While BitLocker encrypts all data on a disk, per-file encryption goes even further by including a unique encryption key for each file. Further, every update to every file is encrypted using its own encryption key. Before they're stored, the keys to the encrypted content are stored in a physically separate location from the content. Every step of this encryption uses Advanced Encryption Standard (AES) with 256-bit keys...* [82].

2.1.2 AES notation and basic information

- M - is a message which will be ciphered and $len(M) \pmod{16} = 0$. If the message length do not fulfill this equation, appropriate padding method have to be used.
- State - 4x4 matrix (see tab. (2.1)) which is used for ciphering operations [34]. Firstly it contains message, than ciphering operations can be done. $s_{i,j}$ are denoting bytes, so as it can be seen $4*4B = 16B = 16*8b = 128b$. As we mentioned earlier, at first step message is put into a State array. It is done by taking byte of the message after byte, and putting them into rows.
  Example: let's consider message given in a hex form:
  $M = 0xa1b1c1d1e1f1a2b2c2d2e2f2a3b3c3d3$,

$s_{0,0}$	$s_{0,1}$	$s_{0,2}$	$s_{0,3}$
$s_{1,0}$	$s_{1,1}$	$s_{1,2}$	$s_{1,3}$
$s_{2,0}$	$s_{2,1}$	$s_{2,2}$	$s_{2,3}$
$s_{3,0}$	$s_{3,1}$	$s_{3,2}$	$s_{3,3}$

Table 2.1 AES State array

than State will look as presented in tab.(2.2). Note that technically each row is

0xa1	0xb1	0xc1	0xd1
0xe1	0xf1	0xa2	0xb2
0xc2	0xd2	0xe2	0xf2
0xa3	0xb3	0xc3	0xd3

Table 2.2 AES State array example

32 bits word (it is also classic integer size). It implicates the fact, that State rows can be interpreted as integer numbers which can speed up computations. In such case, State contains 4 elements and in our example $S(0) = 0xa1b1c1d1$, etc.

- Nb - is equal to four, because it is the number of 32 bits words stored in a State array [34].

- Nk - as we mentioned earlier, possible key sizes are 128, 192 or 256 bits. However, the key is also stored as 32 bits words. Thus appropriate Nk values are: 4, 6 or 8 [34].

- K - ciphering and deciphering key. Contains 32 bits words.

- Nr - represent number of rounds. When $Nk = 4$, $Nr = 10$. When $Nk = 6$, $Nr = 12$ and when $Nk = 8$, $Nr = 14$ [34].

For ciphering and deciphering purposes we also have to define 5 tables containing constant values: *Sbox* (see tab.(2.3)), *InvSbox* (see tab.(2.7)), *MixingVars* (see tab.(2.5)) and *InvMixingVars* (see tab.(2.6)) [34].

2.1.3 AES Ciphering algorithm

The ciphering algorithm presented by NIST looks as follows [34]:

Step 0 $K = ExpandKey(K)$
Step 1 $State = M$
Step 2 $State = AddRoundKey(State, K[0, ..., Nb-1])$
Step 3 $i = 1, ..., Nr-1$ do Steps 4, 5, 6, 7

0x63	0x7C	0x77	0x7B	0xF2	0x6B	0x6F	0xC5	0x30	0x01	0x67	0x2B	0xFE	0xD7	0xAB	0x76
0xCA	0x82	0xC9	0x7D	0xFA	0x59	0x47	0xF0	0xAD	0xD4	0xA2	0xAF	0x9C	0xA4	0x72	0xC0
0xB7	0xFD	0x93	0x26	0x36	0x3F	0xF7	0xCC	0x34	0xA5	0xE5	0xF1	0x71	0xD8	0x31	0x15
0x04	0xC7	0x23	0xC3	0x18	0x96	0x05	0x9A	0x07	0x12	0x80	0xE2	0xEB	0x27	0xB2	0x75
0x09	0x83	0x2C	0x1A	0x1B	0x6E	0x5A	0xA0	0x52	0x3B	0xD6	0xB3	0x29	0xE3	0x2F	0x84
0x53	0xD1	0x00	0xED	0x20	0xFC	0xB1	0x5B	0x6A	0xCB	0xBE	0x39	0x4A	0x4C	0x58	0xCF
0xD0	0xEF	0xAA	0xFB	0x43	0x4D	0x33	0x85	0x45	0xF9	0x02	0x7F	0x50	0x3C	0x9F	0xA8
0x51	0xA3	0x40	0x8F	0x92	0x9D	0x38	0xF5	0xBC	0xB6	0xDA	0x21	0x10	0xFF	0xF3	0xD2
0xCD	0x0C	0x13	0xEC	0x5F	0x97	0x44	0x17	0xC4	0xA7	0x7E	0x3D	0x64	0x5D	0x19	0x73
0x60	0x81	0x4F	0xDC	0x22	0x2A	0x90	0x88	0x46	0xEE	0xB8	0x14	0xDE	0x5E	0x0B	0xDB
0xE0	0x32	0x3A	0x0A	0x49	0x06	0x24	0x5C	0xC2	0xD3	0xAC	0x62	0x91	0x95	0xE4	0x79
0xE7	0xC8	0x37	0x6D	0x8D	0xD5	0x4E	0xA9	0x6C	0x56	0xF4	0xEA	0x65	0x7A	0xAE	0x08
0xBA	0x78	0x25	0x2E	0x1C	0xA6	0xB4	0xC6	0xE8	0xDD	0x74	0x1F	0x4B	0xBD	0x8B	0x8A
0x70	0x3E	0xB5	0x66	0x48	0x03	0xF6	0x0E	0x61	0x35	0x57	0xB9	0x86	0xC1	0x1D	0x9E
0xE1	0xF8	0x98	0x11	0x69	0xD9	0x8E	0x94	0x9B	0x1E	0x87	0xE9	0xCE	0x55	0x28	0xDF
0x8C	0xA1	0x89	0x0D	0xBF	0xE6	0x42	0x68	0x41	0x99	0x2D	0x0F	0xB0	0x54	0xBB	0x16

Table 2.3 AES Sbox constants

0x01	0x00	0x00	0x00
0x02	0x00	0x00	0x00
0x04	0x00	0x00	0x00
0x08	0x00	0x00	0x00
0x10	0x00	0x00	0x00
0x20	0x00	0x00	0x00
0x40	0x00	0x00	0x00
0x80	0x00	0x00	0x00
0x1b	0x00	0x00	0x00
0x36	0x00	0x00	0x00

Table 2.4 AES Rcon Constants

0x02	0x03	0x01	0x01
0x01	0x02	0x03	0x01
0x01	0x01	0x02	0x03
0x03	0x01	0x01	0x02

Table 2.5 AES MixingVars Constants

0x0e	0x0b	0x0d	0x09
0x09	0x0e	0x0b	0x0d
0x0d	0x09	0x0e	0x0b
0x0b	0x0d	0x09	0x0e

Table 2.6 AES InvMixingVars Constants

0x52	0x09	0x6A	0xD5	0x30	0x36	0xA5	0x38	0xBF	0x40	0xA3	0x9E	0x81	0xF3	0xD7	0xFB
0x7C	0xE3	0x39	0x82	0x9B	0x2F	0xFF	0x87	0x34	0x8E	0x43	0x44	0xC4	0xDE	0xE9	0xCB
0x54	0x7B	0x94	0x32	0xA6	0xC2	0x23	0x3D	0xEE	0x4C	0x95	0x0B	0x42	0xFA	0xC3	0x4E
0x08	0x2E	0xA1	0x66	0x28	0xD9	0x24	0xB2	0x76	0x5B	0xA2	0x49	0x6D	0x8B	0xD1	0x25
0x72	0xF8	0xF6	0x64	0x86	0x68	0x98	0x16	0xD4	0xA4	0x5C	0xCC	0x5D	0x65	0xB6	0x92
0x6C	0x70	0x48	0x50	0xFD	0xED	0xB9	0xDA	0x5E	0x15	0x46	0x57	0xA7	0x8D	0x9D	0x84
0x90	0xD8	0xAB	0x00	0x8C	0xBC	0xD3	0x0A	0xF7	0xE4	0x58	0x05	0xB8	0xB3	0x45	0x06
0xD0	0x2C	0x1E	0x8F	0xCA	0x3F	0x0F	0x02	0xC1	0xAF	0xBD	0x03	0x01	0x13	0x8A	0x6B
0x3A	0x91	0x11	0x41	0x4F	0x67	0xDC	0xEA	0x97	0xF2	0xCF	0xCE	0xF0	0xB4	0xE6	0x73
0x96	0xAC	0x74	0x22	0xE7	0xAD	0x35	0x85	0xE2	0xF9	0x37	0xE8	0x1C	0x75	0xDF	0x6E
0x47	0xF1	0x1A	0x71	0x1D	0x29	0xC5	0x89	0x6F	0xB7	0x62	0x0E	0xAA	0x18	0xBE	0x1B
0xFC	0x56	0x3E	0x4B	0xC6	0xD2	0x79	0x20	0x9A	0xDB	0xC0	0xFE	0x78	0xCD	0x5A	0xF4
0x1F	0xDD	0xA8	0x33	0x88	0x07	0xC7	0x31	0xB1	0x12	0x10	0x59	0x27	0x80	0xEC	0x5F
0x60	0x51	0x7F	0xA9	0x19	0xB5	0x4A	0x0D	0x2D	0xE5	0x7A	0x9F	0x93	0xC9	0x9C	0xEF
0xA0	0xE0	0x3B	0x4D	0xAE	0x2A	0xF5	0xB0	0xC8	0xEB	0xBB	0x3C	0x83	0x53	0x99	0x61
0x17	0x2B	0x04	0x7E	0xBA	0x77	0xD6	0x26	0xE1	0x69	0x14	0x63	0x55	0x21	0x0C	0x7D

Table 2.7 AES InvSBox constants

Step 4 $State = SubWords(State)$
Step 5 $State = ShiftRows(State)$
Step 6 $State = MixColumns(State)$
Step 7 $State = AddRoundKey(State, K[i*Nb, ..., (i+1)*Nb-1])$
Step 8 do Steps 4, 5
Step 9 $State = AddRoundKey(State, K[Nr*Nb, ..., (Nr+1)*Nb-1])$

Note that putting message to the state presented in Step 1 is described in sec. (2.1.2). After all those operations $State$ contains the cryptogram.
Deciphering algorithm presented by NIST [34]:

Step 0 $K = ExpandKey(K)$
Step 1 $State = AddRoundKey(State, K[Nr*Nb, ..., (Nr+1)*Nb-1])$
Step 2 $i = Nr-1, ..., 1$ do Steps 3, 4, 5, 6
Step 3 $State = InvSubWords(State)$
Step 4 $State = InvShiftRows(State)$
Step 5 $State = AddRoundKey(State, K[i*Nb, ..., (i+1)*Nb-1])$
Step 6 $State = InvMixColumns(State)$
Step 7 do Steps 3, 4
Step 8 $State = AddRoundKey(State, K[0, ..., Nb-1])$

After all those operations, $State$ contains original, deciphered message. All ciphering and deciphering functions are described in the next sections.

2.1.4 SubWords(State) and InvSubWords(State)

This function uses *SBox* (see tab.(2.3)) for changing bytes values stored in the $State$ during ciphering, and *InvSBox* (see tab, 2.7) during deciphering.

Lets assume, that $State(0,0) = 0xa1$. $0xa1$ is indicating which value from *Sbox* (during ciphering) will replace value currently stored in $State(0,0)$. In particular, 4 most significant bits indicate row in *SBox* and 4 least significant bits indicate column number of *SBox* [34]. Thus in our example: $State(0,0)' = SBox(0xa, 0x1) = SBox(10,1) = 0x32$ where $State(0,0)'$ is the new value of $State(0,0)$.
Function *SubWords* is performing this substitution with usage of *Sbox* on all *State* values during ciphering (see eq. (2.1)) and function *InvSubWords* is performing this substitution with usage of *InvSBox* on all *State* values during deciphering [34].

$$State'(i,j) = SBox\left(\left\lfloor \frac{State(i,j)}{16} \right\rfloor, State(i,j) - \left\lfloor \frac{State(i,j)}{16} \right\rfloor * 16\right), i,j = 0,...,3 \tag{2.1}$$

Note, that eq. (2.1) is also equivalent to eq. (2.2).

$$State'(i,j) = SBox(ShiftRight(State(i,j),1), State(i,j) \wedge 0xF), i,j = 0,..,3 \tag{2.2}$$

2.1.5 ExpandKey(K)

As we mentioned earlier, AES key can contain 4, 6 or 8 32-bit words, and the key size is determining the number of rounds (Nr). In fact, in each round we are operating on 4 32-bit words of the key. Practically input key words are used as a seed for generation of another $Nr * Nb$ different 32-bit words of K. The final size of K is equal to $(Nr+1) * Nb$ which produces $Nr+1$ vectors, each containing four 32-bit words. Those vectors can be also presented as 4x4 matrices containing bytes values (if we divide each 32-bit word into 4 bytes).
to expand values of K, the following algorithm should be used [34]:

Step 1 $i = Nk, ..., (Nr+1) * Nb - 1$ do Steps 2-7
Step 2 $tmp = K(i-1)$
Step 3 if $i \pmod{Nk} = 0$ do Step 4
Step 4 $tmp = RotateLeft(SubWord(tmp),1) \oplus Rcon\left(\left\lfloor \frac{i}{Nk} \right\rfloor - 1\right)$
Step 5 else if $Nk > 6 \wedge i \pmod{Nk} = 4$ do Step 6
Step 6 $tmp = SubWord(tmp)$
Step 7 $K.append(K(i-Nk) \oplus tmp)$

Note, that this algorithm assumes that initial Nk 32-bit words of key K were generated, and $SubWords(tmp)$ is the same function as described in sec. (2.1.4) performed on one vector containing 4 bytes. $Rcon(i)$ denotes 32-bit word stored in $i-th$ row of Rcon constants table (see tab. 2.4).

2.1.6 ShiftRows(State) and InvShiftRows(State)

ShiftRows operation can be formally represented as in eq. (2.3) and InvShiftRows as in eq. (2.4) [34].

$$State^{T\prime}(i,j) = State^{T}(i,(i+j) \pmod 4), i,j = 0,...,3 \tag{2.3}$$

$$State^{T\prime}(i,j) = State^{T}(i,(4-i+j) \pmod 4), i,j = 0,...,3 \tag{2.4}$$

In fact, ShiftRows is performing left binary rotation by a row number and InvShiftRows is performing right binary rotation by a row number. If we represent state as a 4 32-bit words, ShiftRows can be represented as in eq.(2.5) and InvShiftRows as in eq.(2.6) equivalently [34].

$$State^{T\prime}(i) = RotateLeft(State^{T}(i),i), i = 0,...,3 \tag{2.5}$$

$$State^{T\prime}(i) = RotateRight(State^{T}(i),i), i = 0,...,3 \tag{2.6}$$

After using ShiftRows or InvShiftRows $State'$ is becoming new $State$. Note, that operations are done on $State$ transposition.

2.1.7 MixingColumns(State) and InvMixingColumns(State)

MixingColumns operation is based on a matrix multiplication and is presented in eq. (2.7). InvMixingColumns is also a matrix multiplication, however constant values are different, see eq. (2.8).

$$State^{T\prime} = MixingVars \times State^{T} \tag{2.7}$$

$$State^{T\prime} = InvMixingVars \times State^{T} \tag{2.8}$$

MixingVars are presented in tab. (2.5) and InvMixingVars in tab. (2.6). The important thing is that MixingVars and InvMixingVars represents polynomials coefficients in $GF(2^8)$ and all calculations are done under this field [34]. Thus multiplication of byte $a = [b_7,b_6,b_5,b_4,b_3,b_2,b_1,b_0]$ by $0x02$ can be done as presented in eq. (2.9) [108].

$$a * 0x02 = \begin{cases} if \quad b_7 = 1 \; ShiftLeft(a,1) \oplus 0x1b \\ else \; b_7 = 0 \; ShiftLeft(a,1) \end{cases} \tag{2.9}$$

Plus operator is represented by XOR: $a+b = a \oplus b$. All multiplications must be done with usage of appropriate combinations of adding and multiplications by $0x02$. For example: if $a = 0xaa$ and $b = 0x09$ then $a*b = 0xaa*0x09 = 0xaa*(0x02*0x02*0x02+0x01) = 0xaa*0x02*0x02*0x02+0xaa = 0x4f*0x02*0x02+0xaa = 0x9e*0x02+0xaa = 0x27+0xaa = 0x8d$.
Note that operations are done on $State$ transposition.

2.1.8 AddRoundKey(State, RoundKey)

This function is performing XOR operation on RoundKey words and corresponding State words (see eq.(2.10)) [34].

$$State'(i,j) = State(i,j) \oplus RoundKey(i,j), i,j = 0,...,3 \tag{2.10}$$

$State(i,j)$ denotes j-th byte in $i-th$ 32-bit word of State and $RoundKey(i,j)$ denotes j-th byte in i-th 32-bit word of RoundKey. Note, that:

- RoundKey is created by four 32-bit words taken from key K after key extension. Ciphering and deciphering algorithms presented in sec. (2.1.3) are indicating which words should be taken for a given round.

2.2 Secure Hash Algorithm 2

In this chapter, we will describe two classic hashing algorithms which are current Secure Hash Standards (SHS): SHA-256 and SHA-512. Before we will describe algorithms itself, we are presenting additional functions which will be useful [90]:

- S function [90]:

$$S(x,a,b,c) = RotateRight(x,a) \oplus RotateRight(x,b) \oplus RotateRight(x,c) \tag{2.11}$$

- s function [90]:

$$s(x,a,b,c) = RotateRight(x,a) \oplus RotateRight(x,b) \oplus ShiftRight(x,c) \tag{2.12}$$

- Maj function [90]:

$$Maj(a,b,c) = (a \wedge b) \oplus (a \wedge c) \oplus (b \wedge c) \tag{2.13}$$

- Ch function [90]:

$$Ch(a,b,c) = (a \wedge b) \oplus (not(a) \wedge c) \tag{2.14}$$

For both of hashing algorithms mentioned earlier, messages have to have appropriate byte length. Let's define the padding algorithm also [90]:

1. m - byte array representing message
2. n - block size

Step 0 $pad(m,n)$:
Step 1 $tmp = len(m) * 8$
Step 2 $m.append(0x80)$