Advanced Encryption Standard and Modes of Operation
AES

- Advanced Encryption Standard (AES) is a symmetric cryptographic algorithm
- AES has been originally requested by NIST for replacing DES
- A long and open selection process has chosen one algorithm to become AES: Rijndael
- Rijndael was designed by the two Belgian cryptographers: Vincent Rijmen and Joan Daemen
Difference of AES and Rijndael

• AES is a subset of the functions of Rijndael:
  – has a fixed block size of 128 bits
  – and a secret key of either 128, 192 or 256 bits
• Rijndael can work with any combination of key and data block length, from a minimum of 128 to a max of 256 bits, with a step of 32 bits
• this is the only difference between AES and Rijndael, the basic structure is essentially the same for both
Cipher Structure

• consider first the version with a secret key of 128 bits and then explain the difference in the other two cases
• the cipher is divided into two parts:
  – key schedule
  – data path
as it is customary for symmetric algorithms
Cipher Structure

- data path consists of the round function, repeated for 10 times
- at the beginning the plaintext is XORed with the secret key (operation is called Initial KeyAddition)
- operation MixColumns is missing in the last round
Cipher Structure

- the plaintext to encrypt is represented as a matrix of bytes, called state or S
- the state matrix S is a square matrix of $4 \times 4 = 16$ bytes
- after 10 rounds the state matrix S contains the ciphertext
AES Structure

encryption algorithm

PLAINTEXT

KeyAddition

ROUND 1

ROUND 9

ROUND 10

SECRET KEY

KEY SCHEDULE

ROUND KEY 0

ROUND KEY 1

ROUND KEY 9

ROUND KEY 10

CRYPTED DATA

structure of a generic round

INPUT DATA

SUBBYTES

SHIFTROWS

MIXCOLUMNS

ADDITION

OUTPUT DATA
AES – Encryption

SubBytes

S-BOX

state array

ShiftRows

state array

rotation of

Foundations of Cryptography - AES pp. 8 / 50
**AES – Encryption**

### MixColumns

\[
\begin{array}{cccc}
  s'_0 & s'_4 & s'_8 & s'_{12} \\
  s'_1 & s'_5 & s'_9 & s'_{13} \\
  s'_2 & s'_6 & s'_{10} & s'_{14} \\
  s'_3 & s'_7 & s'_{11} & s'_{15} \\
\end{array}
\]

\[
\begin{array}{cccc}
  02 & 03 & 01 & 01 \\
  01 & 02 & 03 & 01 \\
  01 & 01 & 02 & 03 \\
  03 & 01 & 01 & 02 \\
\end{array}
\]

field GF(2^8)

### AddRoundKey

\[
\begin{array}{cccc}
  s'_0 & s'_4 & s'_8 & s'_{12} \\
  s'_1 & s'_5 & s'_9 & s'_{13} \\
  s'_2 & s'_6 & s'_{10} & s'_{14} \\
  s'_3 & s'_7 & s'_{11} & s'_{15} \\
\end{array}
\]

\[
\begin{array}{cccc}
  s_0 & s_4 & s_8 & s_{12} \\
  s_1 & s_5 & s_9 & s_{13} \\
  s_2 & s_6 & s_{10} & s_{14} \\
  s_3 & s_7 & s_{11} & s_{15} \\
\end{array}
\]

\[
\begin{array}{cccc}
  k_0 & k_4 & k_8 & k_{12} \\
  k_1 & k_5 & k_9 & k_{13} \\
  k_2 & k_6 & k_{10} & k_{14} \\
  k_3 & k_7 & k_{11} & k_{15} \\
\end{array}
\]

**bit-wise XOR**

**polynomial multiplications modulo x^4+1**
SBOX

• SubBytes transformation is the application of a SBOX to the 16 bytes of the state matrix

• SBOX consists of two transformations: an inversion in $\text{GF}(2^8)$ and an affine function

• motivations of such a structure are:
  – non linearity
    • correlation between input-output is minimum
    • max difference propagation probability is minimized
  – algebraic complexity
SBOX - Inversion

- finite field $\text{GF}(2^8)$ is represented using:
  
  $$G(x) = x^8 + x^4 + x^3 + x + 1$$

  as irreducible generator polynomial
  
  (no. irr. pol. in $\text{GF}(2^8)$: $N_8(2) = 30$)

- the first SBOX transformation is inversion

- element 0 (which is not invertible in any field) is mapped to itself
SBOX – Affine Transformation

- inversion is followed by an affine transformation operating byte-wise through in GF(2) computations

\[
\begin{pmatrix}
c_0 \\
c_1 \\
c_2 \\
c_3 \\
c_4 \\
c_5 \\
c_6 \\
c_7
\end{pmatrix}
= \begin{pmatrix}
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
b_0 \\
b_1 \\
b_2 \\
b_3 \\
b_4 \\
b_5 \\
b_6 \\
b_7
\end{pmatrix}
+ \begin{pmatrix}
0 \\
1 \\
1 \\
0 \\
0 \\
0 \\
1 \\
1
\end{pmatrix}
\]

- affine transformation does not alter the non-linear behavior of inversion, but increases the difficulty of interpolation
SBOX – Affine Transformation

- affine transformation consists first of a multiplication by a constant matrix and then of the addition of a constant vector
- basic operations are executed in GF(2)
- constant matrix is invertible in GF(2), thus the affine transformation is a invertible
InvSubBytes

- InvSubBytes is the application of the inverse SBOX to the state matrix
- inverse SBOX is obtained by applying first the inverse affine transformation and then inversion in GF($2^8$)
ShiftRows

• the purpose of this transformation is to introduce diffusion and to minimize the cost of the operation
• it consists of rotating the rows of the state matrix (see previous figure)
• transformation is easily invertible, just shift to the opposite direction
MixColumns

• The transformation manages each column, \( w_i \), of the state matrix \( S \) as an element of the polynomial ring

\[
A[x] = F_{2^8}[x] / <n(x)>, \quad n(x) = x^4 + 1
\]

\[e.g.: \ w_0 = [s_0, s_1, s_2, s_3] = s_3x^3 + s_2x^2 + s_1x + s_0\]

• design criteria are:
  – diffusion through the column
  – high performance on 8 bit processors
  – linearity for simplicity
MixColumns

- The MixColumns operation multiplies each $w_i$ by a fixed unitary element $a(x) = 03_{\text{hex}} x^3 + 01_{\text{hex}} x^2 + 01_{\text{hex}} x + 02_{\text{hex}}$ which is invertible in the ring $A[x]$
- the coefficients are chosen to facilitate multiplication
- inverse MixColumns is obtained by taking the inverse: $a^{-1}(x) \mod (x^4+1)$:
  $$a^{-1}(x) = 0B_{\text{hex}} x^3 + 0D_{\text{hex}} x^2 + 09_{\text{hex}} x + 0E_{\text{hex}}$$
MixColumns

Reducing $x^4, x^5, x^6$ modulo $x^4+1$, the result of transforming each column of the state matrix is expressed as:

\[
\begin{bmatrix}
  s'_{0,c} \\
  s'_{1,c} \\
  s'_{2,c} \\
  s'_{3,c}
\end{bmatrix}
= \begin{bmatrix}
  02 & 03 & 01 & 01 \\
  01 & 02 & 03 & 01 \\
  01 & 01 & 02 & 03 \\
  03 & 01 & 01 & 02
\end{bmatrix}
\begin{bmatrix}
  s_{0,c} \\
  s_{1,c} \\
  s_{2,c} \\
  s_{3,c}
\end{bmatrix}
\]

$(0 \leq c \leq 3)$
Round Property

- two rounds suffice to yield a full diffusion:
  - every bit of the output depends on all the bits of the input
  - equivalently, complementing one bit of the input changes 50% of the bits of the output
Key Schedule

- secret key is expanded in 10 round keys
- \( k_j = k_{j-1} \ xor \ k_{j-4} \) if \( j \neq 0 \) mod 4; if \( j = 0 \) mod 4 see the side figure
- every \( k_i \) is a 32 bit word corresponding to a column of the unrolled key
Decryption

• decryption is obtained by applying the inverse round transformations in reverse order and by using the round keys in reverse order
• encryption round is defined as the sequence of SubBytes, ShiftRows, MixColumns and AddRoundKey
• decryption round is the sequence of InvShiftRows, InvSubBytes, AddRoundKey and InvMixColumns
Secret Key of 192 and 256 Bits

- in these cases the round function is applied 12 times for 192 bits and 14 times for 256 bits
- key schedule is slightly different, for the details see the specification by NIST
Notes

• for a complete analysis of the design choices there is a complete book:
  Joan Daemen and Vincent Rijmen
  “The Design of Rijndael”
  Ed. Springer-Verlag
Implementing AES

• there are many possibilities for implementing AES

• such a great flexibility is due to the fact that AES was explicitly designed to:
  – have both SW and HW efficient implementation
  – work well at byte, word (32 bits) or block 8128 bits) level
SW Implementation

- the inversion necessary in the SBOX is too complex to compute in software
- the alternative is to implement the SBOX as a look-up table
- the best choice is to use two tables, one for SBOX and one for InvSubBytes (inverse SBOX)
SW Implementation

• ShiftRows can be moved in front of or after SBOX

• a general solution is to integrate ShiftRows and SBOX: the bytes are output from SBOX accordingly to the ShiftRows order

• MixColumns is directly implemented as a multiplication by $a(x)$ in $A[x] = F_{28}[x] \mod <n(x)>, n(x) = x^4+1$

• AddRoundKey is just a XOR of bit sequences
Optimization

- to speed up execution it is possible to create a T table
- T table stores directly the results of the SBOX and MixColumns relative to a single byte
- the four bytes of a state column are passed through T table, rotated and added
Optimization

- to increase performances it is even possible to use four different tables storing the values already rotated
- these tables increase the memory space from 1 k Byte to 16 k Bytes
Equivalent Decryption

- decryption round can be rearranged to have the same sequence of transformations as encryption: InvShiftRows, InvSbox, InvMixColumns and AddRoundKey
- this is possible thanks to the linearity of InvMixColumns
- now it is possible to create a unique table for InvSubBytes and InvMixColumns
- but the round keys have to be processed accordingly
Equivalent Decryption

• the transformation to be applied to the round key is the InvMixColumns, as:
  \[
  \text{InvMixColumns}(\text{state} + \text{key}) = \\
  \text{InvMixColumns}(\text{state}) + \text{InvMixColumns}(\text{key})
  \]
  because MixColumns is linear

• this transformation can be applied only to the unrolled key, so it is not an overhead cost for decryption
Key Schedule

- it is generally better to schedule all the round keys in advance and store them
- thanks to the structure of the round, the key schedule can be performed “on-the-fly”: compute a round key only when needed
- this could be useful for devices subject to memory constraints
HW Implementation

• the simplest way to implement AES is to instantiate the HW circuits for one round and iterate it 10 times
• also in HW the central point is the implementation of SBOX
• SBOX can be implemented by a look-up table (LUT), but … better solutions exist!
HW Implementation

• if SBOX is implemented by a LUT, 16 SBOXes (one for each byte of the state matrix) take about 80% of the area
• if SBOX is decomposed into inversion followed by affine transformation, it is possible to compute the inverse in the composite finite field \( GF((2^4)^2)) \)
HW Implementation

• an element of $\text{GF}(2^8)$ can be viewed as:
  – a polynomial of degree seven with coefficients in $\text{GF}(2)$
  – or a polynomial of degree one with coefficients in $\text{GF}(2^4)$
• both representations are equivalent, it is only necessary to have a transformation to convert from one representation to the other one
• composite field $\text{GF}((2^4)^2))$ allows to reduce SBOX silicon area of roughly 50%
HW Implementation

• SW implementations relay on key schedule executed in advance
• in the case of HW implementations, the memory for storing all the round keys is too expensive
• key schedule is executed “on-the-fly”
HW Implementation

- if silicon area is a constraint, then it is possible to implement the AES round by using 4 SBOXes instead of 16
- this requires 4 clock cycles to execute the round function, instead of 1 cycle
- if throughput is the major issue, it is possible to pipeline the round function
Other Algorithms

• there are other symmetric algorithms:
  – Safer++ (Secure And Fast Encryption Routine), used in Bluetooth
  – Kasumi/Misty, proposed for UMTS
  – RC5 and RC6 (patented by RSA-Security)

• trend is to rely on AES for everything
Mode of Operation

• A block cipher can be used in a simple way, called **Electronic Code Book**
• plaintext is divided into blocks of the same size
• if the length of the message is not a multiple of the block size, padding is required: just add bits to reach the required length
• one of the most used paddings is 10*: concatenate at the end of the plaintext a single 1 and as many 0es as needed
  – very simple to understand where the padding ends, if it is known that padding is present
Drawback of ECB

• ECB has a drawback:
  – equal plaintext blocks are encrypted to identical ciphertext blocks
  – this gives advantage to an attacker

• alternative modes of operation have been introduced
Example of ECB

AES

AES
Cipher Text Stealing (CTS)

CTS is a tweak to an encryption mode

Length of plaintext is at least 1 block

The processing of the last two blocks of plaintext is altered resulting in a reordered transmission of ciphertext and no ciphertext expansion
CBC Cipher Block Chain

- an Initialization Vector (IV) is needed to start the “chain”
- but there is no need of keeping the IV secret
Cipher Feedback (CFB)

\[ P_1 \xrightarrow{E_k} C_1 \quad P_2 \xrightarrow{E_k} C_2 \quad P_3 \xrightarrow{E_k} C_3 \quad P_4 \xrightarrow{E_k} C_4 \]

Encryption

IV \rightarrow C_1 \quad C_2 \quad C_3 \quad C_4 \quad \text{Output IV}
Output Feedbacks (OFB)
Counter Mode

\[ \text{IV} \rightarrow \underbrace{+1}_{E_k} \rightarrow \underbrace{+1}_{E_k} \rightarrow \underbrace{+1}_{E_k} \rightarrow \underbrace{+1}_{E_k} \rightarrow C_1 \rightarrow C_2 \rightarrow C_3 \rightarrow C_4 \]
Notes on Modes

• OFB, CFB and Counter Mode do not need the decryption primitive
• CBC is the most used mode, but Counter Mode is gaining interest
• there are other modes for guaranteeing data integrity instead of confidentiality
• CBC, CFB and OFB modes can not be parallelized, while CTR and ECB modes can be
Error Propagation

• in CBC mode a bit flip in the ciphertext affects the complete deciphered block and also the next one
• In Counter Mode a bit flip affects only the specific bit affected, there is no error propagation
• remember that error injection could be an attack (fault-injection attacks)
CBC-MAC
(Message Authentication Code)

- the scheme is equal to the CBC mode, but only the last output is used as a TAG

Foundations of Cryptography - AES
pp. 48 / 50
CBC-MAC Notes

• CBC-MAC is secure only for messages with length multiple of the data block size and messages of fixed length
• for a general MAC, a derivation has been standardized recently to pad the last block in a proper way: named CMAC or OMAC
Privacy and Data Integrity

• it is possible to use modes that guarantee both confidentiality and data integrity
• one of these modes is CCM: a combination of CBC-MAC and Counter Mode
Authentication

• it is possible to create a simple authentication protocol
• devices that have to be authenticated are equipped with the same secret key
• when they need to authenticate, one device (verifier) sends a random number (challenge)
• the second device (prover) encrypts the challenge and sends it back
• the verifier decrypts the answer of the prover and checks whether it is equal to the original challenge (or encrypts the challenge and compares the result)