CSCI361

Computer Security

Advanced Encryption Standard (AES)

Outline

- The development of the Advanced Encryption Standard (AES).
  - Selection criteria.
  - Finalists.
- Rijndael.
  - Broad structure.
  - Detailed structure.
    - Round operations.

The development of AES

- The National Institute of Standards and Technology (NIST) worked with the international cryptographic community to develop an *Advanced Encryption Standard* (AES).

- The overall aim:
  - To develop a Federal Information Processing Standard (FIPS) for an encryption algorithm for protecting sensitive (unclassified) government information well into the twenty-first century.
  - The standard was planned for use by the U.S. Government and, on a voluntary basis, by the private sector.

- NIST selected *Rijndael* at the end of a very long and complex evaluation process:
  - January 2, 1997. NIST announced the initiation of the project.
  - September 12, 1997. NIST made a formal call for algorithms.

  NIST specified minimum requirements:
  - Implement symmetric key cryptography as a block cipher.
  - Block size of 128 bits.
  - Key sizes of 128, 192, and 256 bits.

- August 1998: NIST announced 15 AES candidate algorithms at the First AES Candidate Conference (AES1) and solicited public comments.
- March 1999: A Second AES Candidate Conference (AES2) was held to discuss the results of the analysis.
- August 1999: NIST announced its selection of five finalist algorithms from the fifteen candidates:
  - **MARS** (IBM; USA): Modified Feistel rounds in which one fourth of the data block is used to alter the other three fourths of the data block.
  - **RC6** (RSA Labs; USA): Feistel structure; 20 rounds.
  - **Rijndael** (Daemen, Rijmen; Belgium): Substitution-linear transformation network with 10, 12 or 14 rounds, depending on the key size.
  - **Serpent** (Anderson, Biham, Knudsen; UK, Israel, Norway): Substitution-linear transformation network consisting of 32 rounds.
  - **Twofish** (Counterpane; USA): Feistel network with 16 rounds, key-dependent S-boxes.

The finalists

- The five finalists are iterated block ciphers.
  - They specify a transformation that is iterated a number of times on the data block to be encrypted or decrypted.
- Each finalist also specifies a *key scheduling* algorithm for the production of sub-keys.
- NIST gave evaluation criteria to compare the candidate algorithms:
  1. Security.
  2. Cost.
  3. Algorithm and Implementation Characteristics.

- **Security:**
  - Resistance to cryptanalysis.
  - Soundness of the mathematical basis.
  - Randomness of the algorithm output.
  - Relative security as compared to other candidates.
- **Cost:**
  - Computational efficiency (speed) on various platforms,
    - In Round 1, the speed associated with 128-bit keys (the primary key size).
    - In Round 2, hardware implementations and the speeds associated with the 192 and 256-bit key sizes were addressed.
  - Memory requirements:
    - Memory requirements and software implementation constraints for software implementations.

- No attacks were reported against any of the full versions of the finalists.

- The only known attacks are against simplified variants of the algorithms: the number of rounds is reduced or simplified in other ways.
  - It is difficult to assess the significance of the attacks on reduced-round versions.
  - On the other hand, it is standard practice to try to build upon attacks on reduced-round variants.
- The *Security margin* is the degree by which the full number of rounds of an algorithm exceeds the largest number of rounds that have been attacked.
- However, not too much weighting was to be put on any single figure of merit for the security:
  - Security margin would tend to favour novel techniques but wouldn't be a good index to the resistance to new techniques.
  - There is no natural definition for the number of analyzed rounds:
    - MARS has 16 unkeyed mixing rounds and 16 keyed core rounds: is MARS a 16 round or a 32 round algorithm, or something in between?

Other areas of assessment by NIST

- Design Paradigms and Ancestry.
- Simplicity.
- Statistical Testing.
- Machine Word Size:
  - It appears that over the next 30 years, 8-bit, 32-bit, and 64-bit architectures will all play a significant role (128-bit architectures may well be added to the list at some point).
  - Performance with respect to wordsize separated into four groups: 8-bit, 32-bit C and assembler code, 64-bit C and assembler code, and other (Java, DSPs (digital signal processors), etc.).
- Software Implementation Languages
  - Assembler coding to evaluate the best performance on a given architecture.
  - Standard reference code for less costly development.

Rijndael

- Joan Daemen (Proton World International) and Vincent Rijmen (Katholieke Universiteit Leuven)
- The three criteria taken into account in the design of Rijndael were:
  - Resistance against all known attacks.
  - Speed and code compactness on a wide range of platforms.
  - Design simplicity.
- Based on the cipher **Square**, also developed by Daemen and Rijmen.

- The round transformation of Rijndael does not have the Feistel structure.
- The round transformation consists of three *invertible uniform transformations*, called *layers*.
- *Uniform*, means that every bit of the input is treated in a similar way.
- Every layer has its own function:
  - The **linear mixing layer** guarantees high *diffusion* over multiple rounds.
  - The **non-linear layer:** Parallel application of S-boxes that have *optimal worst-case nonlinearity properties*.
  - The **key addition layer:** A simple XOR of the Round Key to the intermediate State.
- The *Round Keys* (or sub-keys) are derived from the Cipher Key by means of the key schedule. This consists of two components:
  - *Key Expansion* and *Round Key Selection*.

- The basic principle for key generation is as follows:
  - The total number of Round Key bits is equal to the block length multiplied by the number of rounds plus 1. Thus for a block length of 128 bits (N_b=4 is the number of (32-bit) words in the block) and N_r=10 rounds, 1408 Round Key bits are needed.
  - The Cipher Key is expanded into an Expanded Key.
  - Round Keys are taken from this Expanded Key in the following way: the first Round Key consists of the first N_b words, the second one of the following N_b words, and so on.

The number of rounds

- The number of rounds, N_r, depends on the key size.

Key Size	Rounds
128	10
192	12
256	14

Attacks on Rijndael:

- (Fergusson et al. 2000)
- For 128-bit keys, 7 out of the 10 rounds of Rijndael have been attacked. The attack on 7 rounds requiring nearly the entire codebook.
- For 192-bit keys, 8 out of the 12 rounds have been attacked.
- For 256-bit keys, 9 out of the 14 rounds have been attacked.
- Rijndael appears to offer an adequate security margin.

The cipher

- Before we look at the specific details of the layer actions we will look at the overall cipher structure, starting with the **Round transformation**.

```
Round(State,RoundKey)
{
ByteSub(State);
ShiftRow(State);
MixColumn(State);
AddRoundKey(State,RoundKey);
}
```

- The final round of the cipher is slightly different. It is defined by:

```
FinalRound(State,RoundKey)
{
ByteSub(State) ;
ShiftRow(State) ;
AddRoundKey(State,RoundKey);
}
```

- The cipher consists of
  - An Initial Round Key addition.
  - N_r-1 Rounds.
  - A final round.

- In pseudo code the cipher can be written as:

```
Rijndael(State,CipherKey)
{
KeyExpansion(CipherKey,ExpandedKey);
AddRoundKey(State,ExpandedKey);
for ( i=1 ; i<Nr ; i++ )
    Round(State,ExpandedKey + Nb*i);
FinalRound(State,ExpandedKey + Nb*Nr);
}
```

- Alternatively, the key expansion can be done beforehand, and Rijndael can be specified in terms of the *Expanded Key*.

```
Rijndael(State,ExpandedKey)
{
AddRoundKey(State,ExpandedKey);
for ( i=1 ; i<Nr ; i++ )
    Round(State,ExpandedKey + Nb*i) ;
FinalRound(State,ExpandedKey + Nb*Nr);
}
```

The inverse cipher

- The inverse of a round is given by:

```
InvRound(State,RoundKey)
{
AddRoundKey(State,RoundKey);
InvMixColumn(State);
InvShiftRow(State);
InvByteSub(State);
}
```

- The inverse of the final round is given by:

```
InvFinalRound(State,RoundKey)
{
AddRoundKey(State,RoundKey);
InvShiftRow(State);
InvByteSub(State);
}
```

How does it work?

- See the presentation on AES with 10 rounds.
- Then you can return to this set of slide if you wish.

Evaluation

- Rijndael designers described their motivation for design choices including substitution box, mix column transformation and the shiftRow offsets.
- The Rijndael cipher can be efficiently implemented on a wide range of processors and in dedicated hardware, 8-bit processors typical for current Smart Cards, and on 32-bit processors typical for PCs.
- There is considerable parallelism in the round transformation.

Security

- Rijndael has provable security against known attacks (in particular with respect to differential and linear cryptanalysis).
- The cipher and its inverse use different components and this practically eliminates the possibility of weak and semi-weak keys, as exist in DES.
- The non-linearity of the key expansion practically eliminates the possibility of equivalent keys.
- The security of the cipher heavily depends on the choice of the S boxes.
  - If the S boxes were linear, the whole cipher would be linear, thus easily breakable.

- There is a known attack, called the *square attack.* It is based on a dedicated attack against Square, the cipher that Rijndael is based on, and that exploits the byte-oriented structure of the Square cipher.
- The attack is also valid for Rijndael, as Rijndael inherits many properties from Square.
  - If one byte is modified in the plaintext, then exactly 4 are modified after one round, and all the 16 are modified after two rounds.
  - The same property holds in the decryption direction.
  - Thus, a one-byte difference cannot lead to a one-byte difference after three rounds.
  - This observation can be used for an attack on 5 rounds of Rijndael.
  - Using a few more tricks, up to 7 rounds can be attacked.
- Although infeasible with current technology, on a 6 round cipher this attack is faster than exhaustive key search, and therefore relevant.
- 2^{32} plaintexts, 2^{72} cipher executions, 2^{32} memory.
- The best theoretic attack breaks up to 8 rounds with over 2^{120} complexity for 128-bit keys and 2^{204} for 256-bit keys.

The details

- We shall now look at the details of the functions in the round transformation.
- Let the 128-bit blocks be divided into 32-bit words, and each word be divided into bytes.
- We view the block as a square of 4 by 4 bytes, where each column has the bytes of a word.

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

- 128-bit keys are views in a similar way.
- Longer keys have additional columns, e.g. 256-bit keys.

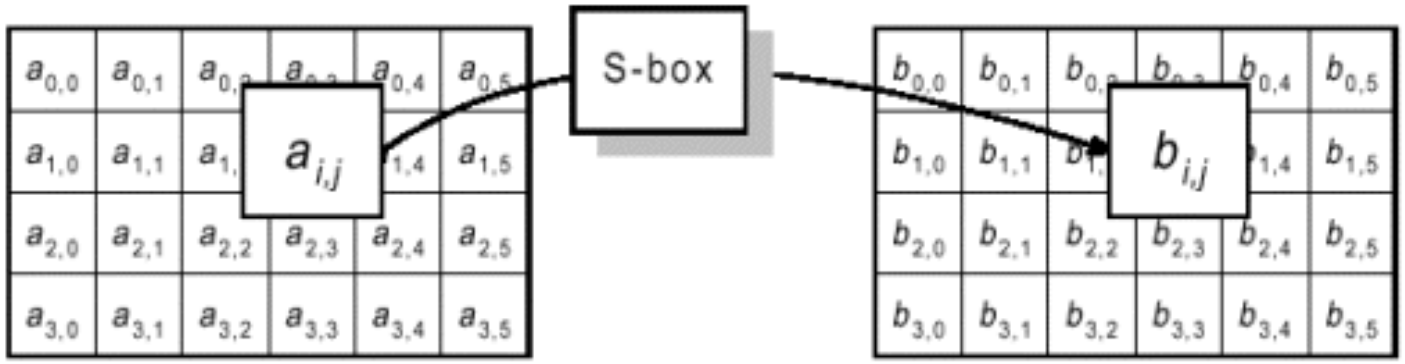
0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

- Rjindael uses four invertible operations:
  1. Byte substitution (S-box, 8-bit to 8-bit).
  2. Shift row (rotating order of bytes in each row).
  3. Mix column (linear mixing of a word column).
  4. Key mixing (addition).

Byte substitution

- The byte substitution is a fixed S box from 8 bits (**x**) to 8 bits (**y**).
- It substitutes all the bytes of the input according to the following array:

y=S[**x**] = {
99, 124, 119, 123, 242, 107, 111, 197, 48, 1, 103, 43, 254, 215, 171, 118,
202, 130, 201, 125, 250, 89, 71, 240, 173, 212, 162, 175, 156, 164, 114, 192,
183, 253, 147, 38, 54, 63, 247, 204, 52, 165, 229, 241, 113, 216, 49, 21,
4, 199, 35, 195, 24, 150, 5, 154, 7, 18, 128, 226, 235, 39, 178, 117,
9, 131, 44, 26, 27, 110, 90, 160, 82, 59, 214, 179, 41, 227, 47, 132,
83, 209, 0, 237, 32, 252, 177, 91, 106, 203, 190, 57, 74, 76, 88, 207,
208, 239, 170, 251, 67, 77, 51, 133, 69, 249, 2, 127, 80, 60, 159, 168,
81, 163, 64, 143, 146, 157, 56, 245, 188, 182, 218, 33, 16, 255, 243, 210,
205, 12, 19, 236, 95, 151, 68, 23, 196, 167, 126, 61, 100, 93, 25, 115,
96, 129, 79, 220, 34, 42, 144, 136, 70, 238, 184, 20, 222, 94, 11, 219,
224, 50, 58, 10, 73, 6, 36, 92, 194, 211, 172, 98, 145, 149, 228, 121,
231, 200, 55, 109, 141, 213, 78, 169, 108, 86, 244, 234, 101, 122, 174, 8,
186, 120, 37, 46, 28, 166, 180, 198, 232, 221, 116, 31, 75, 189, 139, 138,
112, 62, 181, 102, 72, 3, 246, 14, 97, 53, 87, 185, 134, 193, 29, 158,
225, 248, 152, 17, 105, 217, 142, 148, 155, 30, 135, 233, 206, 85, 40, 223,
140, 161, 137, 13, 191, 230, 66, 104, 65, 153, 45, 15, 176, 84, 187, 22
};



Each byte at the input of a round undergoes a non-linear byte substitution according to the following transform:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Substitution ("S")-box

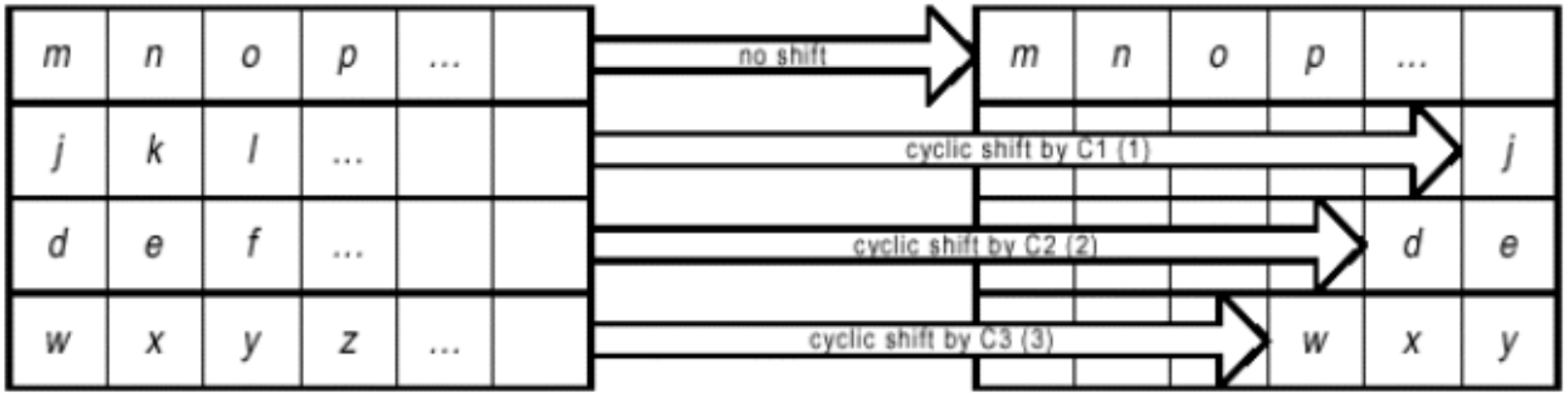
- For example: input byte **x**=0 ➔ $x_i=0$, $\forall i$ ➔ $y_6=y_5=y_1=y_0=1$ $y_7=y_4=y_3=y_2=0$

Shift row

- The Shift row operation rotates the order of bytes.

Nb	C1	C2	C3
4	1	2	3
6	1	2	3
8	1	3	4

Depending on the block length, each "row" of the block is cyclically shifted according to the above table



Mix column

- Mix column is a linear mixing of a word column (four bytes) (a_0, a_1, a_2, a_3) into a word column (b_0, b_1, b_2, b_3). The mixing is defined by operations in the field GF(2^8).
- It is equivalent to

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} f(a_0, a_1, a_2, a_3) \\ f(a_1, a_2, a_3, a_0) \\ f(a_2, a_3, a_0, a_1) \\ f(a_3, a_0, a_1, a_2) \end{pmatrix}$$

where

$$f(a_0, a_1, a_2, a_3) = (a_1 \oplus a_2 \oplus a_3) \oplus g((a_0 \oplus a_1) << 1)$$

$$g(x) = (x \,\&\, 100_x)\,?\,(x \oplus 1b_x) : x$$

The operations <<, &, ?: are left shift, logical AND, and conditional expressions, as used in C. The subscript x's are used to indicate hex.

- The left shift is multiplication by 2, and may turn the 9th bit on. The conditional XOR with $1b_x$ in the *g()* function ensures that the result always fits in a byte. (*g()* enforces the field GF(2^8) modulo the polynomial denoted by $1b_x$, that polynomial being $x^4+1=2^9$ in GF(2^8)).

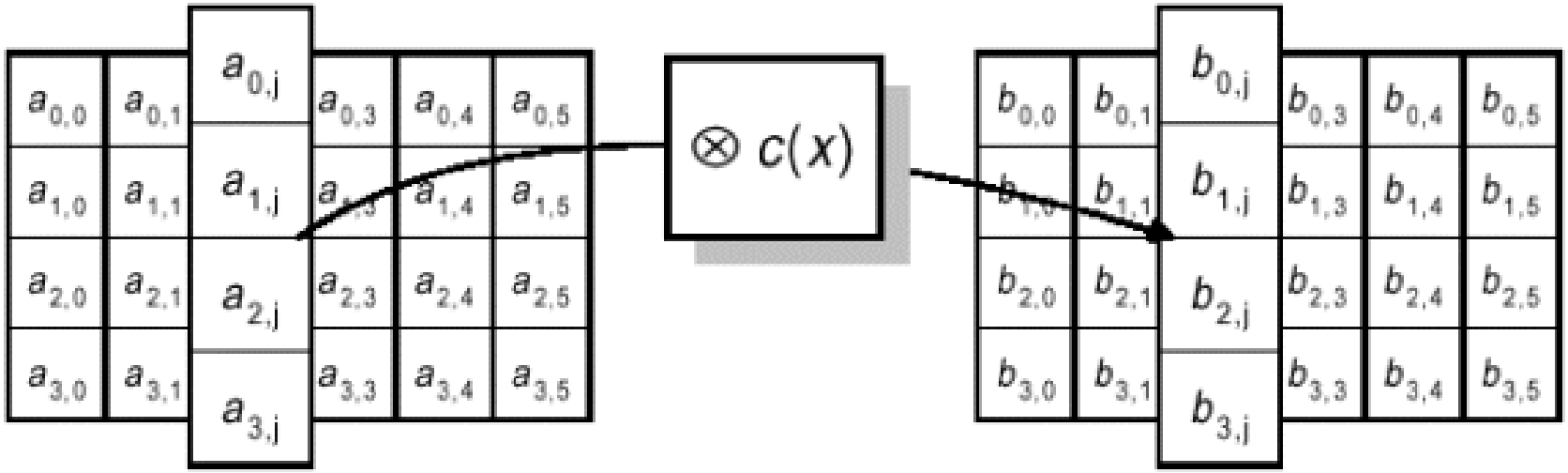
- The **inverse** of the Mix Column operation is also linear, and takes the same form, but with the *f* replaced by *d* functions where

$$d_1(a_0, a_1, a_2, a_3) = a_0 \oplus a_1 \oplus a_2 \oplus a_3$$

$$d_2(a_0, a_1, a_2, a_3) = g(d_1(a_0, a_1, a_2, a_3) << 1) \oplus (a_0 \oplus a_2)$$

$$d_3(a_0, a_1, a_2, a_3) = g(d_2(a_0, a_1, a_2, a_3) << 1) \oplus (a_0 \oplus a_1)$$

$$d_4(a_0, a_1, a_2, a_3) = g(d_3(a_0, a_1, a_2, a_3) << 1) \oplus (a_1 \oplus a_2 \oplus a_3)$$



Each column is multiplied by a fixed polynomial
$C(x) = '03'*X^3 + '01'*X^2 + '01'*X + '02'$

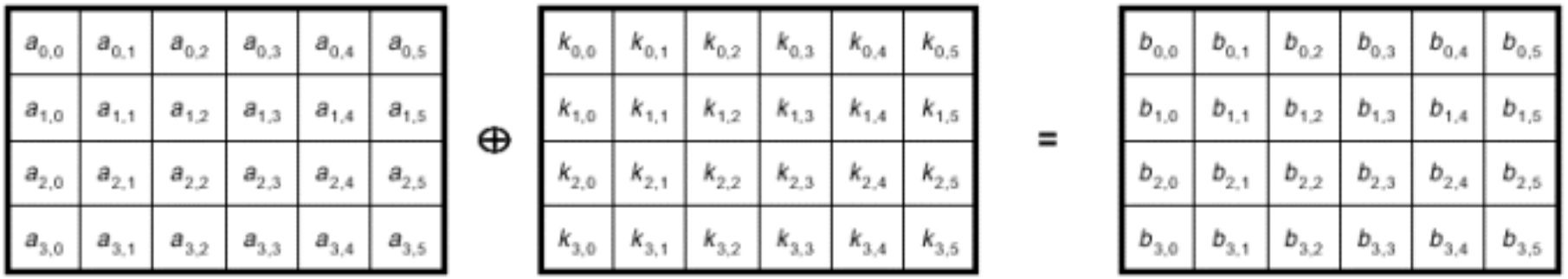
This corresponds to matrix multiplication $b(x) = c(x) \otimes a(x)$:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

Key Mixing

- The key mixing operation XORs the data with a subkey. The subkey has the same size of the blocks.

- The subkeys are computed from the key by a key scheduling algorithm, which we discuss later.



Each word is simply EXOR'ed with the expanded round key

Encryption

- Let us recap on the order of the operations for encryption.
- Encryption is performed by
  1. First key mixing using sub-key K_0.
  2. N_r rounds consisting of
     a) Byte substitution
     b) Shift row
     c) Mix column (except for the last round)
     d) Key mixing using sub-key K_i in round i $\in\{1, \ldots, N_r\}$.

Decryption

- Decryption applies the inverses of the operations in the reverse order, using the reverse order of the sub-keys.
- Decryption requires the application of the inverse S boxes, inverse Mix column, and inverse shift rows.
- The Key mixing/adding is the same operation.

Key scheduling

- The key schedule takes an N_k-word key ($32N_k$-bits) and computes the N_r+1 sub-keys of a total size of $4(N_r+1)$ words.
- Let W be a word array of size $4(N_r+1)$. We let the first N_k words be initialized by the key. Then we compute

```
for i := Nk to 4*(Nr+1)-1 do {
        temp = W[i-1];
        if ((i%Nk) == 0)
                temp = byteSubstitution(temp<<8) ^ RCON[i/Nk];
        W[i] = W[i-Nk] ^ temp;
}
```

where RCON is fixed as the array of powers of 2 in GF(2_8). The array is longer than needed in practice.

RCON[30] = {

01_x, 02_x, 04_x, 08_x, 10_x, 20_x, 40_x, 80_x,

$1b_x$, 36_x, $6c_x$, $d8_x$, ab_x, $4d_x$, $9a_x$, $2f_x$,

$5e_x$, bc_x, 63_x, $c6_x$, 97_x, 35_x, $6a_x$, $d4_x$,

$b3_x$, $7d_x$, fa_x, ef_x, $c5_x$, 91_x};

Then, the first four words of W become the subkey K_0, the next 4 words become K_1, etc.

Efficient implementations

- Efficient implementations of Rijndael combine the S boxes, Shift Rows and Mix Column operations together.
- The Mix Column operation can be performed by four table lookups (8-bit indices, 32-bit output), and XORing the results.
- The Shift row operation can be eliminated by carefully indexing the indices of the input bytes to the succeeding Mix Column operation.
- As the S boxes are also implemented as table lookups, the three operations together can be implemented by 16 table lookups in total (and 12 XORs), using 4 precomputed tables of

Mix Column(Shift Row(S boxes(.)))

CSCI361
Computer Security

Modern stream ciphers

Outline

- Stream ciphers.
- RC4.
- Others:
  - A5/1 & A5/2.
  - SEAL.
- eStream.

Stream ciphers

- In block ciphers plaintext characters are grouped in blocks and then each block is encrypted.
- In stream ciphers, characters are encrypted one at a time.
  - For example the Vigenère cipher.
  - Note that characters could themselves be a block of bits (in the sense the algorithm operates byte by byte say).
- We are now going to look at some modern stream ciphers.

RC4

- Developed by Rivest (1987) this is one of the best known, and most widely used, stream ciphers.

- It effectively acts as a pseudo-random number generator, generating a key-stream which is xor'd with the plaintext.
- Decryption and encryption operations are the same.
- Variable sized secret key up to 256 bytes.
- RC4 operates on bytes.
- Used in the WEP protocol, SSL/TLS
  - Wired Equivalent Privacy (Wireless)
  - Secure Sockets Layer/Transport Layer Security (Web browsers ←→ servers)

Initialisation

- Initialisation for key **K** of length **p**.

```
for i = 0 ... 255
    S[i] = i
for i = 0 ... 255
    j=(j + S[i]=K[i mod p]) mod 256
    swap(S[i],S[j])
```

Encryption/decryption

- Byte by byte processing.

```
i = 0
j = 0
loop until all message encrypted
    Get the next input byte
    i = (i + 1) mod 256
    j = (j + S[i]) mod 256
    swap(S[i],S[j])
    k = S[(S[i] + S[j]) mod 256]
    output: XOR k with input byte
```

Attacks against RC4

- Like all stream ciphers, RC4 is easily broken if a key is used twice.
  - Weaknesses in WEP due to key re-use.
- There is a need to discard the first outputs (1024 bytes) of the keystream.
  - The statistics of first bytes are non-random.
- Theoretical breaks may be possible if gigabytes of (plaintext, ciphertext) is available.
  - This is not usually a problem in practice.
- RC4 is safe for use in some existing applications, but isn't recommended for use to new applications. It doesn't meet various standards and will be phased out.

A5/1 and A5/2

- A5/1 was designed to provide voice privacy for GSM (Global System for Mobile Communications) cellular phones. A5/2 is a weaker version, designed for use in telecommunication environments where A5/1 couldn’t be used.
- A5/1 has a 64-bit key.
- A5/2 is weak.
- A5/1 is severely broken.

SEAL

- **S**oftware **O**ptimized **E**ncryption **A**lgorithm.
- Rogaway and Coppersmith (1994).
- This is a very fast stream cipher, which is optimized for machines with 32-bit architecture and lots of RAM.

Others

- FISH → Broken.
- Pike (from FISH and A5) → Okay.
- ISAAC (**I**ndirection, **S**hift, **A**ccumulate, **A**dd, **C**ount) (RC4 inspired)→ Attack $4.67*10^{1240}$ instead of 10^{2466}. ☺

- Note that while stream ciphers are closely tied with pseudo-random number generates, the properties for a good pseudo-random number generator do not always coincide with good cryptographic properties.

eStream

- This is a project, undertaken in the European Union ECRYPT network to develop a stream cipher standard.
- It is somewhat similar to the AES development procedure, but some improvements have been made to the process.
- Submissions were called for November 2004 and it is due to be completed January 2008.
- There are basically two classes: software and hardware, although some of the ciphers are for both.
- The website for the project is http://www.ecrypt.eu.org/stream/
- Progress is also recorded at http://en.wikipedia.org/wiki/ESTREAM

eStream Result

Profile 1 (SW)	Profile 2 (HW)
HC-128	Grain v1
Rabbit	MICKEY v2
Salsa20/12	Trivium
SOSEMANUK	

New types of attacks

- Fault analysis attacks

CSCI361
Computer Security

Message Integrity

Outline

- What is integrity?
- Message integrity.
- Message authentication codes (MAC).
- Unconditionally secure MAC.
- MAC based on block ciphers.

What is integrity?

- Integrity is both *assurance* that the content of the message has not been altered during transmission, and a mechanism for *detecting* if a message has been altered during transmission.

Message Integrity

- We distinguish two types of threats against the integrity of a message.
  1. **Un-intentional** due to noise:
     - Protection is provided by *error detecting/correcting codes*.
     - For example, the parity bit in the ASCII code is for error detection. (8 bits to represent 7-bits of information).
  2. **Intentional** in which an active spoofer tries to *modify* a message or *generate* a false message.
     - Protection in this case is provided by **Message Authentication Codes** (MAC).

- One common mechanism of implementing protection against either threat is to append to the message a *tag* or *checksum.* Such a tag is a string of bits, in a binary representation.

Error detection: An example

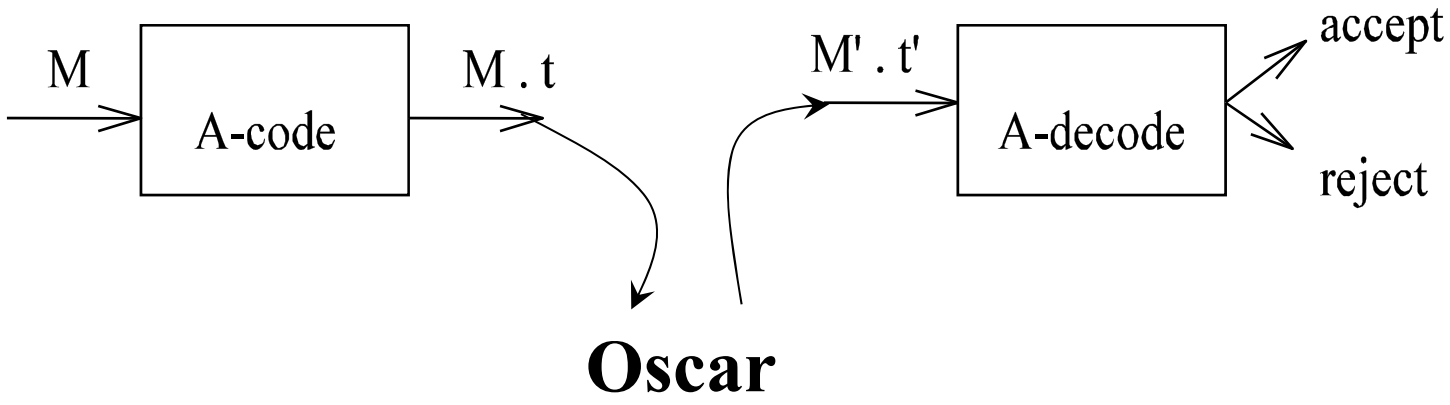
- Algorithms for error detection do not need a secret key. Consider the following example:
- An algorithm for generating parity bits:
  - Consider the message is a block of 8 bits, or, equivalently, an integer in the range 0-255.
  - We attach three parity bits to each block by finding the remainder of the integer divided by 7.
- Consider the message 01111101=125.

  The parity bits are calculated as 125 mod 7=6=110.

  So we send the block 01111101 **110**.
- If one bit error occurs, for example the receiver gets 01**0**11101 110, they detect the error as follows:

  01011101=93 mod 7=2=010 ≠110

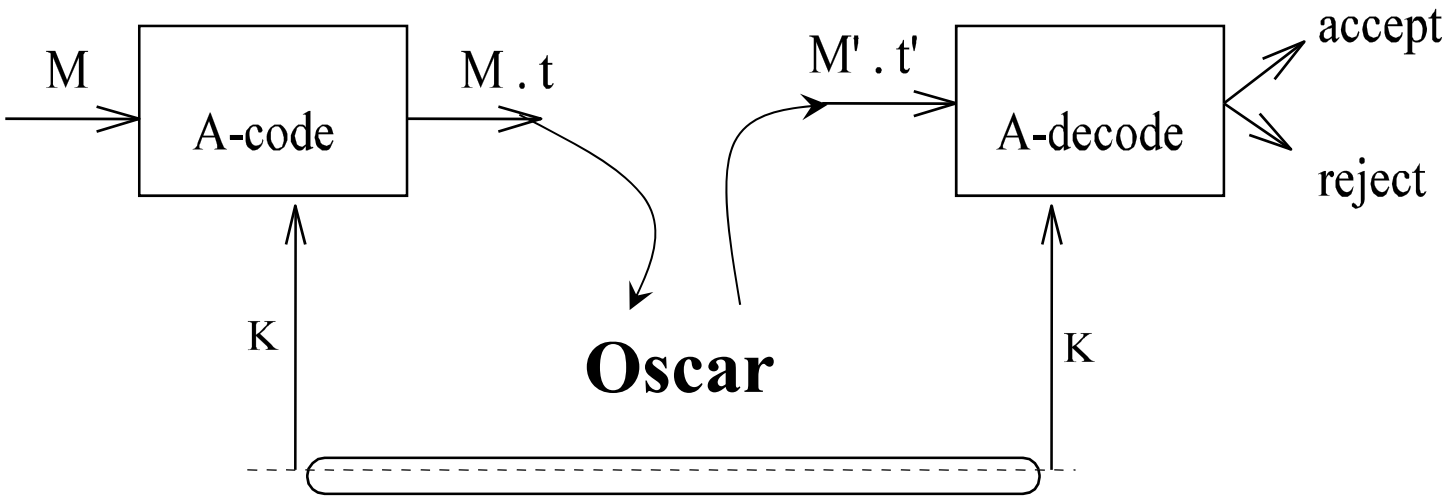
- Error-correcting/detecting codes do not provide protection against spoofing, i.e. against deliberate modification or impersonation attacks.



- Nor does encryption by itself.

Message Authentication Codes

- Transmitter and receiver share a secret key **K**. To transmit **M**, the transmitter calculates a **MAC** and appends it to **M**, thus **(M.t=MAC$_K$(M))**.



- The receiver receives a message **(M.X)**. It uses the key **K** and **M** to calculate **MAC$_K$(M)** and compare it with **X**. If the two match, the received message is accepted as authentic.

- The **MAC** is also called a **cryptographic checksum**.

- A *MAC is secure* if forging **(M,MAC$_K$(M))**, i.e. modifying it without being detected, is hard.

- We can construct a **MAC** with either **unconditional security** or with **practical security**.

- A MAC with unconditional security requires many key bits. Unconditional security in this case means that the chance of success of the enemy is always less than 1.

An unconditionally secure MAC

- Suppose our message consists of a number between *0* and *p-1*, where *p* is a prime.
- Any such message can be broken into blocks of size $log_2 p$.

- The transmitter and the receiver choose/establish/exchange a key **K** which is a pair of numbers; **K**=(a,b) where $0 \leq a, b \leq p-1$.
- Then

$$MAC_K(M) = a\,M + b \bmod p$$

- The enemy observes **(M, MAC_K(M))**, but without knowing **K** cannot change **M** and recalculate its cryptographic checksum.
- This is an example of an Authentication-code (A-code).

- Example: **p=11, K=(a,b)=(2,3).**

- To find the **MAC** for a message 5, we calculate

  $MAC_K(5)= 2*5 + 3 \bmod 11 = 2$

- The enemy doesn't know **K,** but does know a*5 + b mod 11 = 2.

- For an arbitrary choice of 'a' (11 values) the enemy can calculate a corresponding value for 'b'. So the possible keys are (0,2), (1,8), (2, 3), …

- So the chance of correctly guessing the key is 1/11.
- The tag is always one of the 11 elements.

- If the enemy receives another message **M'** whose MAC is calculated using the same key; say M'=7 with $MAC_K(7)=5$, then the enemy can solve the following equations:

  $a*5 + b \bmod 11 = 2 \qquad a * 7 + b \bmod 11 = 6$

  to find the key **K=(2,3)**.

- Thus the key must be changed with every message.

- In general it can be proven that:

Unconditionally secure authentication systems require a large amount of key amount and so are impractical.

- **Example:** Design a MAC system with unconditional security such that the chance of success for an intruder is $P \leq 0.01$.
- Let **p=101, and K=(a,b)= (5,11)**.

- A message is an integer in the range 0-100; that is it requires 7 bits to represent it. The key is 14 bits long. The transmitted message is 14 bits long.

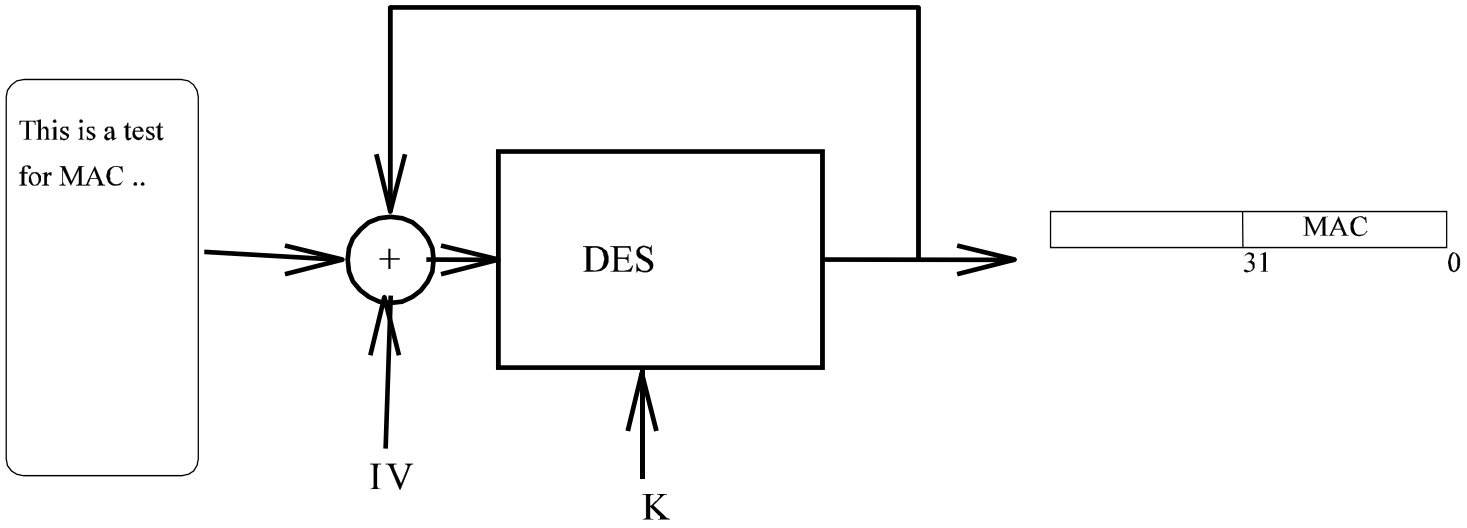
  $MAC_K(90)= 5 * 90 +11 \bmod 101 = 0111001$.
  So the message sent is 1011010 **0111001**.

The chance of the enemy's success is P=1/101 <1/100.

- We see that this MAC is very inefficient:
  - In terms of the number of bits appended to the message
  - In terms of the amount of key information, especially when we consider it requires a different key for every message.
- Essentially this is analogous to the one-time pad in encryption.
  - Recall that system is unconditionally secure encryption.
  - The key information required therein equals the information in the message to be transmitted.
- In a MAC with *practical security* it is always possible to forge a message, but the amount of computation required to do so is large, to the extent it is impractical for the attacker to do so.

Block cipher based MAC

- A block encryption algorithm in CBC (chain-block cipher) mode can be used to generate a checksum. Remember in CBC mode the ciphertext gets “feed” back through the encryption algorithm.
- **Example**: Consider a 32-bit MAC generated using CBC mode of DES. For an arbitrary length message M, $MAC_K(M)$ is the right 32 bits of the output of the algorithm.



Attacking block based MAC

- Question One: What is the cost/chance of the enemy successfully forging a message?

- Attack 1:
  - Enemy uses an exhaustive key search to find the key.
  - Then modifies the message as desired.
  - Then calculates the MAC for the new message using the key.
- The cost something like 2^{64} operations.

- Attack 2:
  - The enemy chooses any message.
  - The enemy randomly selects a 32-bit string and attaches it to the message.

- The chance of the enemy succeeding is $\geq 1/2^{32}$.

- Question Two: Is the enemies cost/chance of success changes by using triple-DES?

- It is more difficult to find the key but the answer is **no**, because the best attack is NOT finding the key of the block cipher algorithm.

- Question 3: How can we reduce enemy's chance of success using DES in CBC mode then?

- Increase the size of the tag: use a 64 bit tag.