# 50.020 Security
# Lecture 11: Block Ciphers

# Status update for 50.020

- Outlook for rest of term:
  - Block Ciphers (AES)
  - Number theory, Finite Field, Primes,...
  - Asymmetric Cryptography
  - Digital Signatures
  - Network Security
  - Digital cash/ Bitcoin
- Some content in these slides is based on slide set of "Understanding Cryptography" by C. Paar and J. Petzl

# Review Ciphers so far



- Simple ciphers:
  - Substitution ciphers
  - Transposition ciphers
  - Problem: small keyspace, frequency analysis, linearity
- Stream Ciphers
  - Problem: key stream generation
- We will now discuss AES block cipher
  - Built on substitution, transposition

# Why Block Ciphers?



- Stream ciphers are
  - Easy to implement
  - Have low latency
  - Have relatively low throughput
  - Suffer from problem of key stream generation
- We need more efficient algorithms to encrypt large data sets
  - Leverage large word width of modern CPUs
  - Parallelize parts in cipher

# Design principles

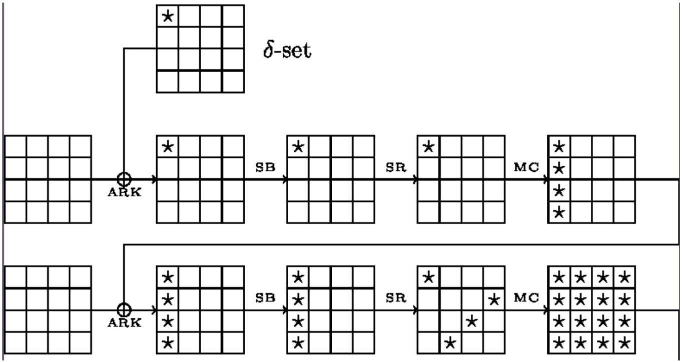


- Confusion: Relation between key and ciphertext should be obscured
  - Can be achieved with substitution ciphers
  - On example, the non-linear S-box helps AES (to discuss later) achieve confusion
- Diffusion: Any change in the plaintext should change 50% of ciphertext bits
  - Can be achieved with transpositions/ permutations
- Both elements are usually combined in block ciphers
  - Several iterations, with one confusion function followed by a diffusion function
  - This is also called a substitution-permutation network

# Diffusion example



In AES, changing a single byte (look at the $*$ in the picture below) in the input will change the entire block (all the other bytes in the output) after 2 rounds. In other words, a difference in a single byte will be propagated into the full block after 2 rounds.



# DES: Data Encryption Standard

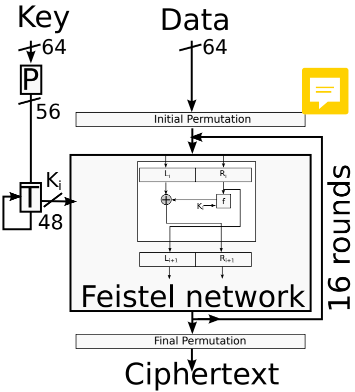


- The predominant block cipher before replaced by AES
- Developed in the 70's
- Block size 64 bit, 64 bit key
  - 8 bit of the key are used as parity (56-bit key)
- Due to small keyspace, brute force attacks are now feasible
- No relevant other attacks have been found
- 3DES (Triple DES) is used as drop-in replacement
  - $E(E^{-1}(E(m,k_1),k_2),k_3)=c$
  - Effective key length increased to 112[1]
  - Backward compatible: with k1=k2=k3, same as DES

[1] And not $3 \cdot 56 = 168$, due to a meet-in-the-middle attack

# DES: Overview

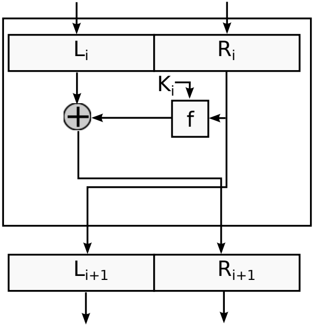





- Main part: 16 rounds of Feistel network
- P is a parity check
- T is a key transformation

# Feistel networks

- Symmetric structure for en-/decryption
- Only the key scheduling differs between both
- Each round in DES is one Feistel iteration
- Function f does diffusion and confusion



# AES: Advanced Encryption Standard



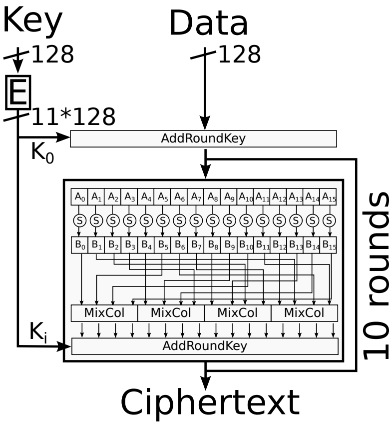
- Result of NIST design competition in 2000
- Designed to operate on 128-bit block size
- Three modes with different key length:
  - AES-128 (10 rounds)
  - AES-192 (12 rounds)
  - AES-256 (14 rounds)
- No efficient attacks are known
  - Apart from side-channel attacks on implementation

# AES: Basic Structure



- 10/12/14 rounds
- Round keys are derived from key
  - 11 roundkeys for 10 rounds
  - Each roundkey is $4 \cdot 32$ bit long
  - Rijndael key schedule is used to derive keys (details are omitted here)
- In each round
  - Substitution layer (SubBytes)
  - Diffusion layer
    - Shiftrows
    - MixColumns
  - Key addition (AddRoundKey)
- All operations operate on Bytes

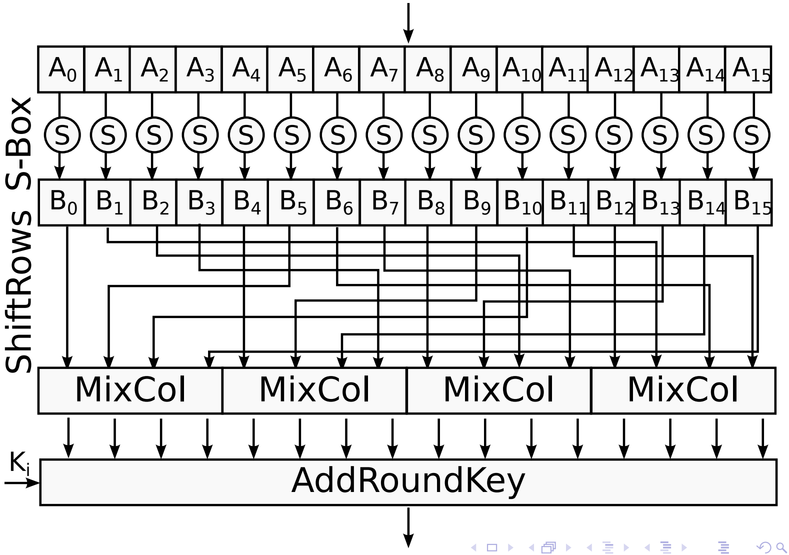
# AES-128: Overview




Note: the last round does not use the mixCol function

# AES: Overview single round





# Substitution layer (S-Boxes)



- Input: 128-bit state (16 Bytes)
- Each Byte is translated by *S-Box*
- S-Boxes are used to substitute Byte input with fixed Byte output
- The mapping is bijective (256 individual mappings)
- Similar to constant key substitution cipher
  - Non-linear if constructed correctly (not just Caesar's shift)
  - So $S(A) + S(B) \neq S(A + B)$

# S-Box implementations



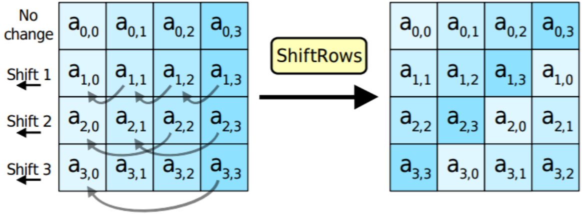
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 63 | 7c | 77 | 7b | f2 | 6b | 6f | c5 | 30 | 01 | 67 | 2b | fe | d7 | ab | 76 |
| 1 | ca | 82 | c9 | 7d | fa | 59 | 47 | f0 | ad | d4 | a2 | af | 9c | a4 | 72 | 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 |

**Example:** S(b7) = a9

- S-Boxes can be implemented
  - As complete lookup tables (see picture above)
  - As logic circuits
- On standard architectures, tables are used
  - 256 Bytes per table

# Diffusion layer

- Shiftrows simply changes order of Bytes (see picture below)



# Diffusion layer



- MixColumn computes a linear combination of the input Bytes (to explain next week)

  $\theta : M_{4\times 4}[\mathbb{F}_{2^8}] \to M_{4\times 4}[\mathbb{F}_{2^8}]$ by

$$\theta(a) = b \Leftrightarrow \begin{bmatrix} b_{0j} \\ b_{1j} \\ b_{2j} \\ b_{3j} \end{bmatrix} = T \cdot \begin{bmatrix} a_{0j} \\ a_{1j} \\ a_{2j} \\ a_{3j} \end{bmatrix}$$

  where $T \in M_{4\times 4}[\mathbb{F}_{2^8}]$ is fixed as

$$T = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}$$

# Decryption



- For the decryption process, each operation is inverted and applied in reverse order
- Round keys are derived in the original way, but applied in inverse order
- XOR of addRoundKey is easy to invert
- Diffusion layer is inverted using the *multiplicative inverse* of the coefficients in $GF(2^8)$
  - In the end, very similar operation as original mixCol and shiftRow
- S-Boxes can also be inverted easily
- Overall, same effort for de- and encryption

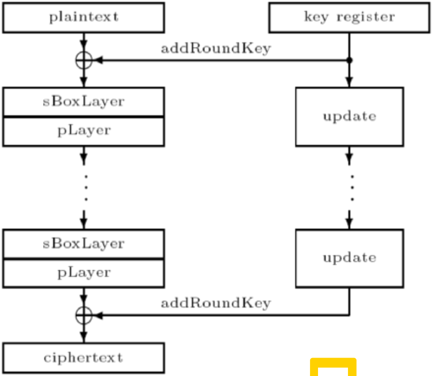
# Present



- Present is an (academic) blockcipher
- Optimised for low cost hardware
- 64-bit block size
- 80/128 bit key
- few rounds
- Efficient S-Box implementation in logic

# Present Overview

generateRoundKeys()
**for** $i = 1$ to 31 **do**
  addRoundKey(STATE,$K_i$)
  sBoxLayer(STATE)
  pLayer(STATE)
**end for**
addRoundKey(STATE,$K_{32}$)




Overview Present cipher
Figure by Axel Poschmann, CHES 2007 talk

# How to encrypt big datasets?

- So far, we only consider input text of block size (e.g., 128 bit)
- How can we use a block cipher to encrypt larger data sets?
- Any ideas?

# How to encrypt big datasets?



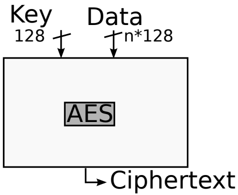
- So far, we only consider input text of block size (e.g., 128 bit)
- How can we use a block cipher to encrypt larger data sets?
- Any ideas?

- Fragment big message into blocks
- Apply the block cipher individually to each block
- This operating mode is called Electronic Codebook mode (ECB)

# Block cipher modes



- To encrypt plaintext longer than block size, it needs to be cut into block-sized chunks
- Padding is used for the last block if required
- The *block cipher mode* determines how the plaintext is fragmented, and the key is used



# How to improve ECB mode?



- With fixed key, ECB mode always encrypts same input to same ciphertext
- Any easy ideas how to fix this?
- You are allowed to exchange 128 bit $n$ (nonce) over a public channel . . .

# How to improve ECB mode?



- With fixed key, ECB mode always encrypts same input to same ciphertext
- Any easy ideas how to fix this?
- You are allowed to exchange 128 bit $n$ (nonce) over a public channel ...

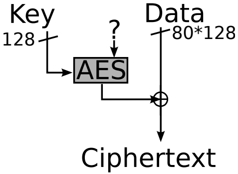
---

- We can $n \oplus k$ to randomize things
- $n$ should change often, but can be public
- More on nonces later ...

# Block ciphers properties



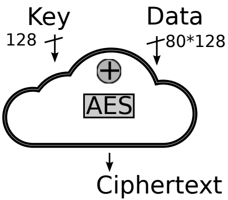
- Ciphertext $c$ depends on plaintext $p$ and key $k$
  - Without $p$ AND $k$, $c$ is unpredictable to attacker
  - Without $c$ AND $k$, $p$ is unpredictable to attacker
- Unpredictability $->$ uniform distribution, relations between plaintext is not preserved in ciphertext
- These useful properties can be used for more than just encryption
  - Which ones can you think of?
    - Key stream generation
    - Cryptographic Hashing
    - Message authentication codes



# Design problem



- Alice and Bob share a short key $k$ (128 Bit)
- They want to exchange 80*128 bit data burst
  - Re-using the key is a bad idea (why?)
  - Data has to be sent with minimal delay
- They have AES, but cannot use it directly
  - Only have few cycles to encrypt the data
  - AES encryption of data takes too long
- Can they use the AES to solve problem?
  - $\oplus$ and some memory can also be used

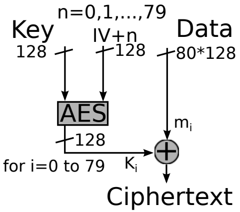


# Counter-Mode CTR



A possible solution:

- Use AES to (pre) compute key stream of 80*128 bit
    - Use $k$ as key, and IV+counter as "message"
    - For next round key, encrypt incremented counter
    - So first input to AES is IV+0
    - Next input is IV+1, ...
- Use key stream for $\oplus$ with data
    - Just like stream cipher
- Decryption is done the same way
- This is called Counter-Mode (CTR) for the block cipher
- Can be easily parallelized

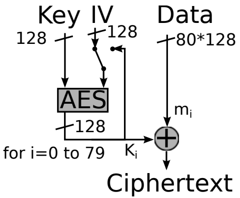


# Output-Feedback-Mode OFB



Another possible solution:

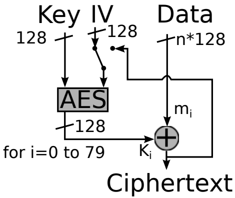
- Use AES to (pre) compute key stream of 80*128 bit
  - Use $k$ as key, initial "message" IV
  - To update round key, encrypt last round's key
- Use key stream for $\oplus$ with data
  - Just like stream cipher
- Decryption is done the same way
- This is called Output-Feedback-Mode (OFB) for the block cipher



# Cipher-Feedback-Mode CFB

A "stream cipher" version of CBC

- Use AES to compute key stream, update with ciphertext
  - Use $k$ as key, initial "message" IV
  - To update round key, encrypt last round's ciphertext
- Use key stream for $\oplus$ with data
  - Just like stream cipher
- Decryption is done in similar same way

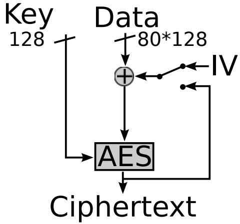


# Cipher-Block-Chaining mode CBC



Lets retrun to "normal" use of AES again

- AES itself will encrypt the messages
  - Example: ECB mode discussed earlier
  - ECB is somewhat "predictable"
- How to make the encryption unpredictable?
- Use ciphertext to mask next encryption
- Plaintext $m_i$ is $\oplus$'ed with $c_{i-1}$ before encryption
- IV is random and non-secret *nonce*
- Called Cipher-Block-Chaining (CBC) mode
- Parallel encryption is not possible since every encryption requires previous cipher.



# Nonces



- Nonce = "Number used Once"
- Nonces are used in CBC mode, but why can they be public?
  - They just introduce "randomness"
  - If they were secret, they would be *shared keys*...

Nonces:

- Should ideally never[2] repeat for a given setting
- Should be generated randomly
- Are used in many different protocols
  - Provides freshness of messages and sessions
- Sounds like salt? Similar, difference is:
  - Salt may repeat *relatively* often
  - Nonces should not repeat in given setting

[2] Never as in "with probability lower than $\epsilon$"

# Design Challenge 2



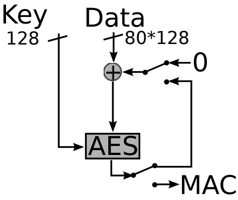
- Alice wants to send Bob a message $m$ of 80*128 bit
- Both share a 128 bit key and have AES
- How can Alice generate a message authentication code?
  - i.e. some value MAC(m,k) to detect changes to m
- CBC encryption does note provide integrity...
  - E.g. if last $c_i$ contains only 128 bit of data
    - Attacker could flip a bit in $c_i$
    - Decryption would result in different $m_i$
- Ideally, MAC validation would also be optional
  - $m$ should be readable for people without $k$

## CBC-MAC



One solution: CBC MAC

- Use AES in CBC mode, but only send the last encrypted chunk (together with *m*)!
- Note: if using CBC AES to encrypt m as well, do NOT use the same key for encryption and MAC computation!



# Conclusion

- Block ciphers such as AES are versatile
  - Can be used to generate key streams
  - Can be used to compute MACs
  - Beware: devil is often in the detail
    - Example: key reuse for MAC and encryption
- Different operating modes can lead to big security problems
  - You need to understand them, before using APIs!