COSC 260: Cryptography

AES

Mahendran Velauthapillai
Linear Cryptanalysis (DES)

Given many input/output pairs, recover key in $< 2^{56}$

Suppose for random $k,m$: There is a relationship between
Subset of the message bits, subsets of the cipher text bits and
subset of the key bits:

$$\Pr \left[ m[i_1] \oplus \cdots \oplus m[i_r] \oplus c[j_k] \oplus \cdots \oplus c[j_v] = k[l_1] \oplus \cdots \oplus k[l_u] \right] = \frac{1}{2} + \varepsilon$$

For some $\varepsilon$. For DES, this exists with $\varepsilon = 1/2^{21} \approx 0.0000000477$
Thm: given $1/\varepsilon^2$ random $(m, c=\text{DES}(k, m))$ pairs then

\[ k[l_1,\ldots,l_u] = \text{MAJ} \left[ m[i_1,\ldots,i_r] \oplus c[j_j,\ldots,j_v] \right] \]

with prob \( \geq 97.7 \)

This means that with $1/\varepsilon^2$ input/output pairs can find

\[ k[l_1,\ldots,l_u] \]

in time $1/\varepsilon^2$
For DES, $\varepsilon = 1/2^{21} \Rightarrow$ with $2^{42}$ input/output pairs can find $k[l_1,\ldots,l_u]$ in time $2^{42}$

Approximately: can find 14 key bits;

This means you only have to brute force $56-14 = 42$ bits

But you need RANDOM $2^{42}$ input/output pairs
**What we have learnt from history?**

- **Observation:** If we have a cipher
  \[ C_1 = \left( P, C, K_1, E_1, D_1 \right) \]
  and a cipher
  \[ C_2 = \left( P, C, K_2, E_2, D_2 \right) . \]
- We define the product cipher as \( C_1 \times C_2 \) by the process of first applying \( C_1 \) and then \( C_2 \)

- Thus \( C_1 \times C_2 = \left( P, C, K_1 \times K_2, E, D \right) \)
- Any key is of the form: \( (k_1, k_2) \)
  and \( E = E_2 (E_1 (x, k_1), k_2) \).
  Likewise \( D = D_1 (k_1, D_2 (k_2, y)) \)

**Note that the product rule is always associative**
Question?:

- Thus if we compute product of ciphers, does the cipher become stronger?
  - The key space become larger
  - 2nd Thought: Does it really become larger.

- Let us consider the product of a two ciphers:

  1) Multiplicative cipher (M): \( y = ax \), where \( \gcd(a, 26) = 1 \)
  2) shift cipher (S): \( y = x + k \)
Is $M \times S = S \times M$?

- $M \times S$: $y = a \cdot x + k$; key = $(a, k)$. This is an affine cipher, total size of key space is 312.
- $S \times M$: $y = a \cdot (x+k) = a \cdot x + a \cdot k$
- Now, since $\gcd(a, 26) = 1$, this is also an affine cipher. key = $(a, ak)$
- Since $\gcd(a, 26) = 1$, $a^{-1}$ exists. There is a one-one relation between $ak$ and $k$. Thus the total size of the key space in $S \times M$ is still 312. Thus this is also the affine cipher

Thus $S$ and $M$ are commutative.
Observation:

- M is a permutation cipher.         - S is a substitution cipher.

- But Composed cipher has a larger key space than each of them.

- If we had computed MxM or SxS, would that have lead to the increase of key space?  

  No.

  – This is because  S x S = S  and  M x M = M
  – These are called idempotent ciphers
Observation

• Thus there is no point of obtaining products of idempotent functions.

• Rather we would get “product ciphers” from non-idempotent ciphers
  – That is by iterating them (rounds)

• How to make non-idempotent functions?
  – Compose two small different cryptosystems which do not commute
Theory

• If there are two cryptosystems which are idempotent and also commute then their product is also idempotent.

\[(S_1 \times S_2) \times (S_1 \times S_2) = S_1 \times (S_2 \times S_1) \times S_2 = S_1 \times (S_1 \times S_2) \times S_2 = (S_1 \times S_1) \times (S_2 \times S_2) = S_1 \times S_2\]

Remember: \( M \times S \) is also idempotent. Thus, composing \( M \times S \) does not help.
Why Rounds?

• Consider : \( S = f(x) \) and \( P = x + k \)

Then \( S \times P : f(x) + k \)

Then \( (S \times P) \times (S \times P) : f(f(x) + k) + k \)

– For this multiplication to increase the key length, thus \( S \times P \) should not be idempotent.

\[ f(f(x) + k) + k \neq f(x) + k \]

– This happens if \( f \) is non-linear wrt. +

– Hence we compose linear and non-linear functions to increase the security of a cipher
Triple-DES

part of FIPS 46-3 standard
encrypt: $C = E(K3, D(K2, E(K1, M)))$
decrypt: $M = D(K3, E(K2, D(K1, C))$
3-key: $K1, K2, K3$ all different
    get 168-bit key
2-key: $K1 = K3$
    get 112-bit key
    better than double encryption
1-key: $K1 = K2 = K3$ – how relate to DES?
Triple DES

plaintext

encrypt

Sender

DES/e

DES/d

DES/e

encrypt

ciphertext

k1

k2

k1

decrypt

DES/d

DES/e

DES/d

Receiver

plaintext
The AES process

• 1997: NIST publishes request for proposal


• 1999: NIST chooses 5 finalists

• 2000: NIST chooses Rijndael as AES (designed in Belgium)

Key sizes: 128, 192, 256 bits. Block size: 128 bits
Advanced Encryption Standard (AES)

FIPS Pub 197, November 26, 2001
Uses Rijndael algorithm developed by
Joan Daemen, Vincent Rijmen of Belgium
Algorithm selected by open process that started in 1997
Requirements
  block size: 128 bits
  key size: 128, 192, and 256 bits
Key sizes and rounds
  AES-128 – 10 rounds
  AES-192 – 12 rounds
  AES-256 – 14 rounds
<table>
<thead>
<tr>
<th>input bytes</th>
<th>State array</th>
<th>output bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$in_0$</td>
<td>$S_{0,0}$</td>
<td>$out_0$</td>
</tr>
<tr>
<td>$in_4$</td>
<td>$S_{0,1}$</td>
<td>$out_4$</td>
</tr>
<tr>
<td>$in_8$</td>
<td>$S_{0,2}$</td>
<td>$out_8$</td>
</tr>
<tr>
<td>$in_{12}$</td>
<td>$S_{0,3}$</td>
<td>$out_{12}$</td>
</tr>
<tr>
<td>$in_1$</td>
<td>$S_{1,0}$</td>
<td>$out_1$</td>
</tr>
<tr>
<td>$in_5$</td>
<td>$S_{1,1}$</td>
<td>$out_5$</td>
</tr>
<tr>
<td>$in_9$</td>
<td>$S_{1,2}$</td>
<td>$out_9$</td>
</tr>
<tr>
<td>$in_{13}$</td>
<td>$S_{1,3}$</td>
<td>$out_{13}$</td>
</tr>
<tr>
<td>$in_2$</td>
<td>$S_{2,0}$</td>
<td>$out_2$</td>
</tr>
<tr>
<td>$in_6$</td>
<td>$S_{2,1}$</td>
<td>$out_6$</td>
</tr>
<tr>
<td>$in_{10}$</td>
<td>$S_{2,2}$</td>
<td>$out_{10}$</td>
</tr>
<tr>
<td>$in_{14}$</td>
<td>$S_{2,3}$</td>
<td>$out_{14}$</td>
</tr>
<tr>
<td>$in_3$</td>
<td>$S_{3,0}$</td>
<td>$out_3$</td>
</tr>
<tr>
<td>$in_7$</td>
<td>$S_{3,1}$</td>
<td>$out_7$</td>
</tr>
<tr>
<td>$in_{11}$</td>
<td>$S_{3,2}$</td>
<td>$out_{11}$</td>
</tr>
<tr>
<td>$in_{15}$</td>
<td>$S_{3,3}$</td>
<td>$out_{15}$</td>
</tr>
</tbody>
</table>

Figure 3. State array input and output.
AES is a Substitution Permutation Network (Not Feistel)
AES-128 schema

10 rounds

Key expansion: 16 bytes → 176 bytes
<table>
<thead>
<tr>
<th>Code size</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-compute round functions (24KB or 4KB)</td>
<td>largest</td>
</tr>
<tr>
<td>Pre-compute S-box only (256 bytes)</td>
<td>smaller</td>
</tr>
<tr>
<td>No pre-computation</td>
<td>smallest</td>
</tr>
</tbody>
</table>
Example: Javascript AES

AES in the browser:

AES library (6.4KB)
no pre-computed tables

Prior to encryption:
pre-compute tables

Then encrypt using tables
AES in hardware

AES instructions in Intel Westmere:

- **aesenc, aesenclast**: do one round of AES
  
  128-bit registers: \( \text{xmm1} = \text{state}, \ \text{xmm2} = \text{round key} \)
  
  \text{aesenc} \text{ xmm1, xmm2} ; \ \text{puts result in xmm1}

- **aeskeygenassist**: performs AES key expansion

- Claim 14 x speed-up over OpenSSL on same hardware

Similar instructions on AMD Bulldozer
Attacks

Best key recovery attack:

four times better than ex. search \[\text{BK'11}\]

Related key attack on AES-256: \[\text{BK'09}\]

Given $2^{99}$ inp/out pairs from four related keys in AES-256

can recover keys in time $\approx 2^{99}$
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
  byte  state[4,Nb]

  state = in

  AddRoundKey(state, w[0, Nb-1])          // See Sec. 5.1.4

  for round = 1 step 1 to Nr-1
    SubBytes(state)                        // See Sec. 5.1.1
    ShiftRows(state)                       // See Sec. 5.1.2
    MixColumns(state)                      // See Sec. 5.1.3
    AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
  end for

  SubBytes(state)
  ShiftRows(state)
  AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

  out = state
end

Figure 5. Pseudo Code for the Cipher.¹
Figure 6. `SubBytes()` applies the S-box to each byte of the State.
Algorithm 4.4: SUBBYTES(a7a6a5a4a3a2a1a0)

external FIELDINV, BINARYTOFIELD, FIELDTOBINARY

z ← BINARYTOFIELD(a7a6a5a4a3a2a1a0)

if z ≠ 0
    then z ← FIELDINV(z)

(a7a6a5a4a3a2a1a0) ← FIELDTOBINARY(z)

(c7c6c5c4c3c2c1c0) ← (01100011)

copyright: In the following loop, all subscripts are to be reduced modulo 8

for i ← 0 to 7
    do bi ← (ai + ai+4 + ai+5 + ai+6 + ai+7 + ci) mod 2

return (b7b6b5b4b3b2b1b0)
In contrast to the S-boxes in DES, which are apparently “random” substitutions, the AES S-box can be defined algebraically. The algebraic formulation of the AES S-box involves operations in a finite field (finite fields are discussed in detail in Section 7.4). We include the following description for the benefit of readers who are already familiar with finite fields (other readers may want to skip this description, or read Section 7.4 first): The permutation $\pi_S$ incorporates operations in the finite field

$$F_{2^8} = \mathbb{Z}_2[x]/(x^8 + x^4 + x^3 + x + 1).$$

Let $\text{FIELDINV}$ denote the multiplicative inverse of a field element; let $\text{BINARYTEOFIELD}$ convert a byte to a field element; and let $\text{FIELDTOBINARY}$ perform the inverse conversion. This conversion is done in the obvious way: the field element

$$\sum_{i=0}^{7} a_i x^i$$

corresponds to the byte

$$a_7a_6a_5a_4a_3a_2a_1a_0,$$

where $a_i \in \mathbb{Z}_2$ for $0 \leq i \leq 7$. Then the permutation $\pi_S$ is defined according to Algorithm 4.4. In this algorithm, the eight input bits $a_7a_6a_5a_4a_3a_2a_1a_0$ are replaced by the eight output bits $b_7b_6b_5b_4b_3b_2b_1b_0$. 
<table>
<thead>
<tr>
<th>x</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>63</td>
<td>7c</td>
<td>77</td>
<td>7b</td>
<td>f2</td>
<td>6b</td>
<td>6f</td>
<td>c5</td>
<td>30</td>
<td>01</td>
<td>67</td>
<td>2b</td>
<td>fe</td>
<td>d7</td>
<td>ab</td>
<td>76</td>
</tr>
<tr>
<td>1</td>
<td>ca</td>
<td>82</td>
<td>c9</td>
<td>7d</td>
<td>fa</td>
<td>59</td>
<td>47</td>
<td>f0</td>
<td>ad</td>
<td>d4</td>
<td>a2</td>
<td>af</td>
<td>9c</td>
<td>a4</td>
<td>72</td>
<td>c0</td>
</tr>
<tr>
<td>2</td>
<td>7b</td>
<td>fd</td>
<td>93</td>
<td>26</td>
<td>36</td>
<td>3f</td>
<td>f7</td>
<td>cc</td>
<td>34</td>
<td>a5</td>
<td>e5</td>
<td>f1</td>
<td>71</td>
<td>d8</td>
<td>31</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>04</td>
<td>c7</td>
<td>23</td>
<td>c3</td>
<td>18</td>
<td>96</td>
<td>05</td>
<td>9a</td>
<td>07</td>
<td>12</td>
<td>80</td>
<td>e2</td>
<td>eb</td>
<td>27</td>
<td>b2</td>
<td>75</td>
</tr>
<tr>
<td>4</td>
<td>09</td>
<td>83</td>
<td>2c</td>
<td>1a</td>
<td>1b</td>
<td>6e</td>
<td>5a</td>
<td>a0</td>
<td>52</td>
<td>3b</td>
<td>d6</td>
<td>b3</td>
<td>29</td>
<td>e3</td>
<td>2f</td>
<td>84</td>
</tr>
<tr>
<td>5</td>
<td>53</td>
<td>d1</td>
<td>00</td>
<td>ed</td>
<td>20</td>
<td>fc</td>
<td>b1</td>
<td>5b</td>
<td>6a</td>
<td>cb</td>
<td>be</td>
<td>39</td>
<td>4a</td>
<td>4c</td>
<td>58</td>
<td>cf</td>
</tr>
<tr>
<td>6</td>
<td>00</td>
<td>ef</td>
<td>aa</td>
<td>fb</td>
<td>43</td>
<td>4d</td>
<td>33</td>
<td>85</td>
<td>45</td>
<td>f9</td>
<td>02</td>
<td>7f</td>
<td>50</td>
<td>3c</td>
<td>9f</td>
<td>a8</td>
</tr>
<tr>
<td>7</td>
<td>51</td>
<td>a3</td>
<td>40</td>
<td>8f</td>
<td>92</td>
<td>9d</td>
<td>38</td>
<td>f5</td>
<td>bc</td>
<td>b6</td>
<td>da</td>
<td>21</td>
<td>10</td>
<td>ff</td>
<td>f3</td>
<td>d2</td>
</tr>
<tr>
<td>8</td>
<td>cd</td>
<td>0c</td>
<td>13</td>
<td>ec</td>
<td>5f</td>
<td>97</td>
<td>44</td>
<td>17</td>
<td>c4</td>
<td>a7</td>
<td>7e</td>
<td>3d</td>
<td>64</td>
<td>5d</td>
<td>19</td>
<td>73</td>
</tr>
<tr>
<td>9</td>
<td>60</td>
<td>81</td>
<td>4f</td>
<td>dc</td>
<td>22</td>
<td>2a</td>
<td>90</td>
<td>88</td>
<td>46</td>
<td>ee</td>
<td>b8</td>
<td>14</td>
<td>de</td>
<td>5e</td>
<td>0b</td>
<td>db</td>
</tr>
<tr>
<td>a</td>
<td>0e</td>
<td>32</td>
<td>3a</td>
<td>0a</td>
<td>49</td>
<td>06</td>
<td>24</td>
<td>5c</td>
<td>c2</td>
<td>d3</td>
<td>ac</td>
<td>62</td>
<td>91</td>
<td>95</td>
<td>e4</td>
<td>79</td>
</tr>
<tr>
<td>b</td>
<td>78</td>
<td>25</td>
<td>1e</td>
<td>a6</td>
<td>b4</td>
<td>c6</td>
<td>e8</td>
<td>dd</td>
<td>74</td>
<td>1f</td>
<td>4b</td>
<td>bd</td>
<td>8b</td>
<td>8a</td>
<td>e2</td>
<td>09</td>
</tr>
<tr>
<td>c</td>
<td>ba</td>
<td>70</td>
<td>3e</td>
<td>b5</td>
<td>66</td>
<td>48</td>
<td>03</td>
<td>f6</td>
<td>0e</td>
<td>61</td>
<td>35</td>
<td>57</td>
<td>b9</td>
<td>86</td>
<td>c1</td>
<td>1d</td>
</tr>
<tr>
<td>d</td>
<td>e1</td>
<td>f8</td>
<td>98</td>
<td>11</td>
<td>69</td>
<td>d9</td>
<td>8e</td>
<td>94</td>
<td>9b</td>
<td>1e</td>
<td>87</td>
<td>e9</td>
<td>ce</td>
<td>55</td>
<td>28</td>
<td>df</td>
</tr>
<tr>
<td>e</td>
<td>8c</td>
<td>a1</td>
<td>89</td>
<td>0d</td>
<td>bf</td>
<td>e6</td>
<td>42</td>
<td>68</td>
<td>41</td>
<td>99</td>
<td>2d</td>
<td>0f</td>
<td>b0</td>
<td>54</td>
<td>bb</td>
<td>16</td>
</tr>
</tbody>
</table>

Figure 7. S-box: substitution values for the byte \( xy \) (in hexadecimal format).
Example: We do a small example to illustrate Algorithm 4.4, where we also include the conversions to hexadecimal. Suppose we begin with (hexadecimal) 53. In binary, this is 01010011,

- which represents the field element \( x^6 + x^4 + x + 1 \)
- The multiplicative inverse (in the field \( \mathbb{F}_{2^8} \)) can be shown to be \( x^7 + x^6 + x^3 + x \).
- Therefore, in binary notation, we have \( (a_7a_6a_5a_4a_3a_2a_1a_0) = (11001010) \).

Next, we compute \( b_0 = a_0 + a_4 + a_5 + a_6 + a_7 + c_0 \mod 2 \)
\[ = 0 + 0 + 0 + 1 + 1 + 1 \mod 2 = 1, \]
followed by \( b_1 = a_1 + a_5 + a_6 + a_7 + a_0 + c_1 \mod 2 \)
\[ = 1 + 0 + 1 + 1 + 0 + 1 \mod 2 = 0 \]

etc.

The result is that \( (b_7b_6b_5b_4b_3b_2b_1b_0) = (11101101) \). In hexadecimal notation, \( 11101101 \) is ED. This computation can be checked by verifying that the entry in row 5 and column 3 is ED.
Figure 8. ShiftRows() cyclically shifts the last three rows in the State.
MIXCOLUMN(c)

external FIELDMULT, BINARYTOFIELD, FIELDTOBINARY

for i ← 0 to 3
    ti ← BINARYTOFIELD(si,c)

u0 ← FIELDMULT(x, t0) ⊕ FIELDMULT(x + 1, t1) ⊕ t2 ⊕ t3
u1 ← FIELDMULT(x, t1) ⊕ FIELDMULT(x + 1, t2) ⊕ t3 ⊕ t0
u2 ← FIELDMULT(x, t2) ⊕ FIELDMULT(x + 1, t3) ⊕ t0 ⊕ t1
u3 ← FIELDMULT(x, t3) ⊕ FIELDMULT(x + 1, t0) ⊕ t1 ⊕ t2

for i ← 0 to 3
    si,c ← FIELDTOBINARY(ui)
\[ s'_{0,c} = (\{02\} \cdot s_{0,c}) \oplus (\{03\} \cdot s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \]
\[ s'_{1,c} = s_{0,c} \oplus (\{02\} \cdot s_{1,c}) \oplus (\{03\} \cdot s_{2,c}) \oplus s_{3,c} \]
\[ s'_{2,c} = s_{0,c} \oplus s_{1,c} \oplus (\{02\} \cdot s_{2,c}) \oplus (\{03\} \cdot s_{3,c}) \]
\[ s'_{3,c} = (\{03\} \cdot s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \cdot s_{3,c}) \]

Figure 9 illustrates the `MixColumns()` transformation.

Figure 9. `MixColumns()` operates on the State column-by-column.
More compactly, the column operations can be shown as

\[
\begin{bmatrix}
02 & 03 & 01 & 01 \\
01 & 02 & 03 & 01 \\
01 & 01 & 02 & 03 \\
03 & 01 & 01 & 02 \\
\end{bmatrix}
\times
\begin{bmatrix}
s_{0.0} & s_{0.1} & s_{0.2} & s_{0.3} \\
s_{1.0} & s_{1.1} & s_{1.2} & s_{1.3} \\
s_{2.0} & s_{2.1} & s_{2.2} & s_{2.3} \\
s_{3.0} & s_{3.1} & s_{3.2} & s_{3.3} \\
\end{bmatrix}
= 
\begin{bmatrix}
s'_{0.0} & s'_{0.1} & s'_{0.2} & s'_{0.3} \\
s'_{1.0} & s'_{1.1} & s'_{1.2} & s'_{1.3} \\
s'_{2.0} & s'_{2.1} & s'_{2.2} & s'_{2.3} \\
s'_{3.0} & s'_{3.1} & s'_{3.2} & s'_{3.3} \\
\end{bmatrix}
\]

where, on the left hand side, when a row of the leftmost matrix multiples a column of the state array matrix, additions involved are meant to be XOR operations.
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk )
{
    word temp;
    i = 0;
    while (i < Nk)
    {
        w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
        i = i+1;
    }
    i = Nk;
    while (i < Nb * (Nr+1) )
    {
        temp = w[i-1];
        if (i mod Nk = 0)
            temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
        else if (Nk > 6 and i mod Nk = 4)
            temp = SubWord(temp);

        w[i] =  w[i-Nk] xor  temp;
        i = i + 1
    }//end while
}//end Key expansion

Note that Nk=4, 6, and 8 do not all have to be implemented; they are all included in the conditional statement above for conciseness. Specific implementation requirements for the Cipher Key are presented in Sec. 6.1.
Figure 1: This figure shows the four words of the original 128-bit key being expanded into a key schedule consisting of 44 words. Section 8.8 explains the procedure used for this key expansion. (This figure is from Lecture 8 of “Computer and Network Security” by Avi Kak)
Figure 4: The key expansion takes place on a four-word to four-word basis as shown here. (This figure is from Lecture 8 of "Computer and Network Security" by Avi Kak)
• Let’s say that we have the four words of the round key for the \( i \)th round:

\[
\begin{align*}
    w_i & \quad w_{i+1} & \quad w_{i+2} & \quad w_{i+3}
\end{align*}
\]

• Now we need to determine the words

\[
\begin{align*}
    w_{i+4} & \quad w_{i+5} & \quad w_{i+6} & \quad w_{i+7}
\end{align*}
\]

from the words

\[
\begin{align*}
    w_i & \quad w_{i+1} & \quad w_{i+2} & \quad w_{i+3}
\end{align*}
\]
\[ w_{i+5} = w_{i+4} \otimes w_{i+1} \]
\[ w_{i+6} = w_{i+5} \otimes w_{i+2} \]
\[ w_{i+7} = w_{i+6} \otimes w_{i+3} \]

\[ w_{i+4} = w_i \otimes g(w_{i+3}) \]

- The function \( g() \) consists of the following three steps:
  - Perform a one-byte left circular rotation on the argument 4-byte word.
  - Perform a byte substitution for each byte of the word returned by the previous step by using the same \( 16 \times 16 \) lookup table as used in the SubBytes step of the encryption rounds.
  - XOR the bytes obtained from the previous step with what is known as a round constant. The round constant is a word whose three rightmost bytes are always zero. Therefore, XOR’ing with the round constant amounts to XOR’ing with just its leftmost byte.
InvCipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
  byte state[4,Nb]
  state = in
  AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1]) // See Sec. 5.1.4
for round = Nr-1 step -1 downto 1
  InvShiftRows(state) // See Sec. 5.3.1
  InvSubBytes(state) // See Sec. 5.3.2
  AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
  InvMixColumns(state) // See Sec. 5.3.3
end for
InvShiftRows(state)
InvSubBytes(state)
AddRoundKey(state, w[0, Nb-1])
out = state
end
AES MODES OF OPERATION

Electronic codebook mode (ECB mode):

Given a sequence $x_1 x_2 \ldots$ of plaintext blocks, each $x_i$ is encrypted with the same key $K$, ciphertext blocks, $y_1 y_2 \ldots$.

(BAD!)
Cipher block chaining mode (CBC mode):

$y_i$ is $x$-ored with the next plaintext block, $x_{i+1}$, before being encrypted with the key $K$.

we start with an initialization vector, denoted by IV, and define $y_0 = IV$. (Note that IV has the same length as a plaintext block.)

$$y_i = e_K(y_{i-1} \oplus x_i), \quad i \geq 1$$
Output feedback mode (OFB mode):

keystream is produced by repeatedly encrypting an initialization vector, IV. We define \( z_0 = IV \), and then compute the keystream \( z_1, z_2, \ldots \) using the rule
\[
    z_i = e_K(z_{i-1}) \quad \text{for all } i \geq 1
\]

The plaintext sequence \( x_1, x_2, \ldots \) is then encrypted by computing
\[
    y_i = x_i \oplus z_i, \text{ for all } i \geq 1.
\]
Cipher feedback mode (CFB mode):

We start with $y_0 = IV$ (an initialization vector) and we produce the keystream element $z_i$ by encrypting the previous ciphertext block. That is, $z_i = e_K(y_{i-1})$ for all $i \geq 1$

$y_i = x_i \oplus z_i$, for all $i \geq 1$.

Again, the encryption function $e_K$ is used for both encryption and decryption in CFB mode.
Counter Mode:

Counter mode is similar to OFB mode; the only difference is in how the keystream is constructed. Suppose that the length of a plaintext block is denoted by $m$. In counter mode, we choose a **counter**, denoted $\text{ctr}$, which is a bitstring of length $m$. Then we construct a sequence of bitstrings of length $m$, denoted $T_1, T_2, \ldots$, defined as follows:

$$T_i = \text{ctr} + i - 1 \mod 2^m$$

for all $i \geq 1$. Then we encrypt the plaintext blocks $x_1, x_2, \ldots$ by computing

$$y_i = x_i \oplus e_K(T_i),$$
Frame Format – CTR mode

Original frame

Counter #1 → Cipher → + → Encrypted Packet [15:0]

Counter #2 → Cipher → + → Encrypted Packet [31:16]

... → Cipher → + → Encrypted Packet [Length: (N-1)*16]

Encrypted frame

Counter #1 → Cipher → + → Packet [15:0]

Counter #2 → Cipher → + → Packet [31:16]

... → Cipher → + → Packet [Length: (N-1)*16]

Decoded frame

Counter #1

Counter #2

... → Cipher

Packet [15:0]

Packet [31:16]

Packet [Length: (N-1)*16]
Counter Initialization Vector (IV)

- 32-bit cycle_counter is incremented whenever PON clock wraps around
  - LSB of cycle_counter is transmitted in preamble
  - optical network unit (ONU) resets cycle_counter when changing keys
    - optical line terminal (OLT) monitors changes in key number. When detected:
      - Counter is reset if LSB is 0
      - Counter is set to 32’h0000_0001 if LSB is 1
Counter Initialization Vector (IV)

- Counter Initialization Vector must be used only once (nonce) per a specific key
  (Nonce: A randomly chosen value, different from previous choices, inserted in a message to protect against replays. ...)

- The counter is based on PON clock (PON: Passive Optical Network)

- The counter IV will be a concatenation of:
  - cycle_counter [32 bits]
  - (PON clock+ 16)[32:5] – PON clock rounded to the closest 512nSec boundaries
  - N [7 bits] – Serial number of cipher block inside frame
counter with cipher-block chaining MAC (CCM mode)

Basically, CCM mode combines the use of counter mode (for encryption) with CBC-mode (for authentication). This mode, which is discussed further later; This is used for authenticated encryption.
CCM Mode

This combines Counter mode for encryption and CBC mode for authentication.

Details Later!
Galois/counter mode (GCM):

GCM is another mode used for authenticated encryption. Later
Why Block Cipher Mode of Operation is Needed?

- Advantages of adding Block Cipher Mode of Operation are:
  - The same plaintext will be encrypted to a different ciphertext, at any invocation
    - Important when analyzing repeating patterns (DA, SA, Ethertype, IP headers, TCP headers, known packets like ICMP, ARP, etc…)
  - Helps in handling last block problem
    - Last block can be smaller than code block size
  - Impossible to perform packet replay
  - Increases encryption level
Block Cipher Mode of Operation Recommendation

- National Institute of Standards and Technology (NIST) recommends 5 block cipher modes of operation (publication 800-38A)

- Two features are required:
  - Maintaining packet length – Length of last block should be arbitrary
  - Enabling parallel encryption - Encryption of a plaintext block shouldn’t depend on result from previous block

- From these modes only Counter (CTR) mode supports both Features

- Additional mode Offset Codebook (OCB) was proposed for 802.11, but it is protected by patents
Other Block Ciphers

Blowfish, Twofish – Bruce Schneier et al
CAST – Entrust – S-boxes not fixed
FEAL – more complex per round than DES so fewer rounds - FEAL-4 broken with 5 known plaintexts
GOST – Soviet “DES” std with 256-bit keys, 32 rounds
IDEA – 128-bit keys, PGP used in early versions
RC2 – “Ron’s code” (Ron Rivest), variable size key
RC5 – variable size key
RC6 – candidate for AES
Skipjack – 80-bit key, 32 rounds, NSA initially classified