AES: The Advanced Encryption Standard

- The block cipher AES is a product cipher, it transforms plaintext into ciphertext by iterated use of some round functions ...
- ... the AES round functions form a substitution-permutation network.
- A (very) brief history:
  - Rijndael, by Joan Daemen and Vincent Rijmen, submitted to NIST "DES replacement" contest in 1997-ish; based (at least partly) on Square.
  - The five year contest selected Rijndael as the winner in 2001 and standardised concrete parametrisation of it (e.g., AES-128) as the Advanced Encryption Standard (AES).
- We'll consider:
  1. AES-128 only; keep in mind AES means AES-128 from here on, and that we set:
     - ... the plaintext (and ciphertext) size to 128 bits (i.e., $N_b = 4$).
     - ... the key size to 128 bits (i.e., $N_k = 4$).
     - ... the number of rounds to 11 (i.e., $N_r = 10$ with the first and last rounds differing from the rest).
  2. AES encryption only; with CTR mode this is all we need, but with other use-cases decryption is still important.

AES: The Advanced Encryption Standard

- Keep in mind the following:
  - AES uses the finite field $\mathbb{F}_{2^8}[X]/X^8 + X^4 + X^3 + X + 1$; you can think of elements in this field as bytes with non-standard operations on them ...
  - ... i.e., addition is not integer addition any more; $\oplus_{\mathbb{F}_{2^8}}$, $\odot_{\mathbb{F}_{2^8}}$, $\oslash_{\mathbb{F}_{2^8}}$ are field addition, multiplication and division.
  - It is convenient to use a short-hand for constant field elements:

$$03_{(16)} = \langle 1, 1, 0, 0, 0, 0, 0, 0 \rangle = X + 1$$

  - AES operates on matrices of field elements; note that we can directly read matrix entries column-wise from byte arrays.

AES Overview (1)

- Given the 128-bit secret key, the key schedule computes a sequence of eleven 128-bit round keys ...
- ... within the k-th round, elements of the state are combined via key addition with round key elements:

$$\textsc{Key-Addition}\left(\begin{bmatrix} S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} \\ S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} \\ S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} \\ S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} \end{bmatrix}, \begin{bmatrix} K_{0,4k} & K_{0,4k+1} & K_{0,4k+2} & K_{0,4k+3} \\ K_{1,4k} & K_{1,4k+1} & K_{1,4k+2} & K_{1,4k+3} \\ K_{2,4k} & K_{2,4k+1} & K_{2,4k+2} & K_{2,4k+3} \\ K_{3,4k} & K_{3,4k+1} & K_{3,4k+2} & K_{3,4k+3} \end{bmatrix} \right)$$

$$\downarrow$$

$$\begin{bmatrix} (S_{0,0} \oplus_{\mathbb{F}_{2^8}} K_{0,4k}) & (S_{0,1} \oplus_{\mathbb{F}_{2^8}} K_{0,4k+1}) & (S_{0,2} \oplus_{\mathbb{F}_{2^8}} K_{0,4k+2}) & (S_{0,3} \oplus_{\mathbb{F}_{2^8}} K_{0,4k+3}) \\ (S_{1,0} \oplus_{\mathbb{F}_{2^8}} K_{1,4k}) & (S_{1,1} \oplus_{\mathbb{F}_{2^8}} K_{1,4k+1}) & (S_{1,2} \oplus_{\mathbb{F}_{2^8}} K_{1,4k+2}) & (S_{1,3} \oplus_{\mathbb{F}_{2^8}} K_{1,4k+3}) \\ (S_{2,0} \oplus_{\mathbb{F}_{2^8}} K_{2,4k}) & (S_{2,1} \oplus_{\mathbb{F}_{2^8}} K_{2,4k+1}) & (S_{2,2} \oplus_{\mathbb{F}_{2^8}} K_{2,4k+2}) & (S_{2,3} \oplus_{\mathbb{F}_{2^8}} K_{2,4k+3}) \\ (S_{3,0} \oplus_{\mathbb{F}_{2^8}} K_{3,4k}) & (S_{3,1} \oplus_{\mathbb{F}_{2^8}} K_{3,4k+1}) & (S_{3,2} \oplus_{\mathbb{F}_{2^8}} K_{3,4k+2}) & (S_{3,3} \oplus_{\mathbb{F}_{2^8}} K_{3,4k+3}) \end{bmatrix}$$

AES Overview (2)

- Within the k-th round (apart from $k = 0$) elements in the state are substituted (i.e., replaced) using a non-linear S-box:

$$\textsc{Sub-Bytes}\left(\begin{bmatrix} S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} \\ S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} \\ S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} \\ S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} \end{bmatrix}\right)$$

$$\downarrow$$

$$\begin{bmatrix} \textsc{S-Box}(S_{0,0}) & \textsc{S-Box}(S_{0,1}) & \textsc{S-Box}(S_{0,2}) & \textsc{S-Box}(S_{0,3}) \\ \textsc{S-Box}(S_{1,0}) & \textsc{S-Box}(S_{1,1}) & \textsc{S-Box}(S_{1,2}) & \textsc{S-Box}(S_{1,3}) \\ \textsc{S-Box}(S_{2,0}) & \textsc{S-Box}(S_{2,1}) & \textsc{S-Box}(S_{2,2}) & \textsc{S-Box}(S_{2,3}) \\ \textsc{S-Box}(S_{3,0}) & \textsc{S-Box}(S_{3,1}) & \textsc{S-Box}(S_{3,2}) & \textsc{S-Box}(S_{3,3}) \end{bmatrix}$$

- Unlike DES which has eight S-boxes, AES uses just one ...
- ... the entries are carefully selected to defeat certain attacks.

AES Overview (3)

- Within the k-th round (apart from $k = 0$) elements in the state undergo a row-wise linear transformation (a permutation) to facilitate diffusion:

$$\textsc{Shift-Rows}\left(\begin{bmatrix} S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} \\ S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} \\ S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} \\ S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} \end{bmatrix}\right)$$

$$\downarrow$$

$$\begin{bmatrix} S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} \\ S_{1,1} & S_{1,2} & S_{1,3} & S_{1,0} \\ S_{2,2} & S_{2,3} & S_{2,0} & S_{2,1} \\ S_{3,3} & S_{3,0} & S_{3,1} & S_{3,2} \end{bmatrix}$$

AES Overview (4)

- Within the k-th round (apart from $k = 0$ and $k = 10$) elements in the state undergo a column-wise linear transformation to facilitate diffusion:

$$\textsc{Mix-Columns}\left(\begin{bmatrix} S_{0,j} \\ S_{1,j} \\ S_{2,j} \\ S_{3,j} \end{bmatrix}\right)$$

$$\downarrow$$

$$\begin{bmatrix} 02_{(16)} & 03_{(16)} & 01_{(16)} & 01_{(16)} \\ 01_{(16)} & 02_{(16)} & 03_{(16)} & 01_{(16)} \\ 01_{(16)} & 01_{(16)} & 02_{(16)} & 03_{(16)} \\ 03_{(16)} & 01_{(16)} & 01_{(16)} & 02_{(16)} \end{bmatrix} \otimes_{\mathbb{F}_{2^8}} \begin{bmatrix} S_{0,j} \\ S_{1,j} \\ S_{2,j} \\ S_{3,j} \end{bmatrix}$$

- AES uses a Maximum Distance Separable (MDS) constant matrix ...
- ... the constants are all small, so we never need to perform a general purpose field multiplication within Mix-Columns.

AES Overview (5)

Algorithm (AES-KeyGen)

1. The round keys are formed from a sequence of column-vectors; for $0 \leq j \leq 3$, elements are taken directly from the secret key:

$$
\overbrace{\begin{bmatrix} K_{0,0} \\ K_{1,0} \\ K_{2,0} \\ K_{3,0} \end{bmatrix} \begin{bmatrix} K_{0,1} \\ K_{1,1} \\ K_{2,1} \\ K_{3,1} \end{bmatrix} \begin{bmatrix} K_{0,2} \\ K_{1,2} \\ K_{2,2} \\ K_{3,2} \end{bmatrix} \begin{bmatrix} K_{0,3} \\ K_{1,3} \\ K_{2,3} \\ K_{3,3} \end{bmatrix}}^{\text{round key 0}} \overbrace{\begin{bmatrix} K_{0,4} \\ K_{1,4} \\ K_{2,4} \\ K_{3,4} \end{bmatrix} \begin{bmatrix} K_{0,5} \\ K_{1,5} \\ K_{2,5} \\ K_{3,5} \end{bmatrix} \begin{bmatrix} K_{0,6} \\ K_{1,6} \\ K_{2,6} \\ K_{3,6} \end{bmatrix} \begin{bmatrix} K_{0,7} \\ K_{1,7} \\ K_{2,7} \\ K_{3,7} \end{bmatrix}}^{\text{round key 1}} \cdots \begin{bmatrix} K_{0,43} \\ K_{1,43} \\ K_{2,43} \\ K_{3,43} \end{bmatrix}
$$

2. For $4 \leq j \leq 43$, we repeatedly apply

$$
\begin{bmatrix} K_{0,j} \\ K_{1,j} \\ K_{2,j} \\ K_{3,j} \end{bmatrix} = \begin{cases} \begin{bmatrix} K_{0,j-1} \oplus_{\mathbb{F}_{2^8}} K_{0,j-4} \\ K_{1,j-1} \oplus_{\mathbb{F}_{2^8}} K_{1,j-4} \\ K_{2,j-1} \oplus_{\mathbb{F}_{2^8}} K_{2,j-4} \\ K_{3,j-1} \oplus_{\mathbb{F}_{2^8}} K_{3,j-4} \end{bmatrix} & \text{if } j \neq 0 \pmod 4 \\ \begin{bmatrix} RC[j/4] \oplus_{\mathbb{F}_{2^8}} \text{S-Box}(K_{1,j-1}) \oplus_{\mathbb{F}_{2^8}} K_{0,j-4} \\ \text{S-Box}(K_{2,j-1}) \oplus_{\mathbb{F}_{2^8}} K_{1,j-4} \\ \text{S-Box}(K_{3,j-1}) \oplus_{\mathbb{F}_{2^8}} K_{2,j-4} \\ \text{S-Box}(K_{0,j-1}) \oplus_{\mathbb{F}_{2^8}} K_{3,j-4} \end{bmatrix} & \text{if } j = 0 \pmod 4 \end{cases}
$$

AES Overview (6)

- To encrypt a 128-bit plaintext message M under a 128-bit secret key K, i.e., compute $C = \text{AES-Enc}(K, M)$ ...

Algorithm (AES-Enc)

Input: An 11-element sequence K of 128-bit round keys, the 128-bit plaintext M.
Output: The 128-bit ciphertext C.

$S \leftarrow M$
$S \leftarrow \text{Key-Addition}(S, K_0)$

for $i = 1$ **upto** 9 **step** 1 **do**
    $S \leftarrow \text{Sub-Bytes}(S)$
    $S \leftarrow \text{Shift-Rows}(S)$
    $S \leftarrow \text{Mix-Columns}(S)$
    $S \leftarrow \text{Key-Addition}(S, K_i)$
end

$S \leftarrow \text{Sub-Bytes}(S)$
$S \leftarrow \text{Shift-Rows}(S)$
$S \leftarrow \text{Key-Addition}(S, K_{10})$

return S

- ... where K_i is the i-th round key derived from K.

Implementation #1 (1)

Strategy

- AES was specifically designed to be efficient on constrained platforms (e.g., 8-bit smart-cards).
- An implementation strategy in this case could be:
  1. Represent S as a 16-element array of 8-bit bytes.
  2. Generate round keys during encryption, i.e., online, and potentially recompute them even if secret key is static.
  3. Implement the round functions directly, using various features to make a trade-off in favour of space (i.e., memory footprint) over time (i.e., performance).

Implementation #1 (2)

```
void aes_key( U8* S, U8* K )
{
  S[  0 ] = S[  0 ] ^ K[  0 ];
  S[  1 ] = S[  1 ] ^ K[  1 ];
  S[  2 ] = S[  2 ] ^ K[  2 ];
  S[  3 ] = S[  3 ] ^ K[  3 ];

  S[  4 ] = S[  4 ] ^ K[  4 ];
  S[  5 ] = S[  5 ] ^ K[  5 ];
  S[  6 ] = S[  6 ] ^ K[  6 ];
  S[  7 ] = S[  7 ] ^ K[  7 ];

  S[  8 ] = S[  8 ] ^ K[  8 ];
  S[  9 ] = S[  9 ] ^ K[  9 ];
  S[ 10 ] = S[ 10 ] ^ K[ 10 ];
  S[ 11 ] = S[ 11 ] ^ K[ 11 ];

  S[ 12 ] = S[ 12 ] ^ K[ 12 ];
  S[ 13 ] = S[ 13 ] ^ K[ 13 ];
  S[ 14 ] = S[ 14 ] ^ K[ 14 ];
  S[ 15 ] = S[ 15 ] ^ K[ 15 ];
}
```

```
void aes_sub( U8* S )
{
  S[  0 ] = sbox( S[  0 ] );
  S[  1 ] = sbox( S[  1 ] );
  S[  2 ] = sbox( S[  2 ] );
  S[  3 ] = sbox( S[  3 ] );

  S[  4 ] = sbox( S[  4 ] );
  S[  5 ] = sbox( S[  5 ] );
  S[  6 ] = sbox( S[  6 ] );
  S[  7 ] = sbox( S[  7 ] );

  S[  8 ] = sbox( S[  8 ] );
  S[  9 ] = sbox( S[  9 ] );
  S[ 10 ] = sbox( S[ 10 ] );
  S[ 11 ] = sbox( S[ 11 ] );

  S[ 12 ] = sbox( S[ 12 ] );
  S[ 13 ] = sbox( S[ 13 ] );
  S[ 14 ] = sbox( S[ 14 ] );
  S[ 15 ] = sbox( S[ 15 ] );
}
```

Implementation #1 (3)

```
void aes_row( U8* S )
{
  U8 t0, t1, t2;

  t0      = S[  1 ];
  S[  1 ] = S[  5 ];
  S[  5 ] = S[  9 ];
  S[  9 ] = S[ 13 ];
  S[ 13 ] =      t0;

  t0      = S[  2 ];
  t1      = S[  6 ];
  S[  2 ] = S[ 10 ];
  S[  6 ] = S[ 14 ];
  S[ 10 ] =      t0;
  S[ 14 ] =      t1;

  t0      = S[  3 ];
  t1      = S[  7 ];
  t2      = S[ 11 ];
  S[  3 ] = S[ 15 ];
  S[  7 ] =      t0;
  S[ 11 ] =      t1;
  S[ 15 ] =      t2;
}
```

```
#define MIX_STEP(a,b,c,d)                                  \
{                                                          \
  U8 a1 = S[ a ];        U8 b1 = S[ b ];                   \
  U8 a2 = fmulx( a1 ); U8 b2 = fmulx( b1 );                \
  U8 a3 = a1 ^ a2;       U8 b3 = b1 ^ b2;                  \
                                                           \
  U8 c1 = S[ c ];        U8 d1 = S[ d ];                   \
  U8 c2 = fmulx( c1 ); U8 d2 = fmulx( d1 );                \
  U8 c3 = c1 ^ c2;       U8 d3 = d1 ^ d2;                  \
                                                           \
  S[ a ] = a2 ^ b3 ^ c1 ^ d1;                              \
  S[ b ] = a1 ^ b2 ^ c3 ^ d1;                              \
  S[ c ] = a1 ^ b1 ^ c2 ^ d3;                              \
  S[ d ] = a3 ^ b1 ^ c1 ^ d2;                              \
}

void aes_mix( U8* S )
{
  MIX_STEP(  0,  1,  2,  3 )
  MIX_STEP(  4,  5,  6,  7 )
  MIX_STEP(  8,  9, 10, 11 )
  MIX_STEP( 12, 13, 14, 15 )
}
```

Implementation #1 (4)

```
#define ROUND1()                        \
{                                       \
  aes_key( S, RK );                     \
}
```

```
#define ROUND2()                        \
{                                       \
  aes_sub( S      );                    \
  aes_row( S      );                    \
  aes_mix( S      );                    \
                                        \
  RC = aes_schedule( RK, RC );          \
  aes_key( S, RK );                     \
}
```

```
#define ROUND3()                        \
{                                       \
  aes_sub( S      );                    \
  aes_row( S      );                    \
                                        \
  RC = aes_schedule( RK, RC );          \
  aes_key( S, RK );                     \
}
```

Implementation #1 (5)

```
U8 aes_schedule( U8* RK, U8 RC )
{
  RK[  0 ] = RC ^ sbox( RK[ 13 ] ) ^ RK[  0 ];
  RK[  1 ] =      sbox( RK[ 14 ] ) ^ RK[  1 ];
  RK[  2 ] =      sbox( RK[ 15 ] ) ^ RK[  2 ];
  RK[  3 ] =      sbox( RK[ 12 ] ) ^ RK[  3 ];

  RK[  4 ] =            RK[  0 ]   ^ RK[  4 ];
  RK[  5 ] =            RK[  1 ]   ^ RK[  5 ];
  RK[  6 ] =            RK[  2 ]   ^ RK[  6 ];
  RK[  7 ] =            RK[  3 ]   ^ RK[  7 ];

  RK[  8 ] =            RK[  4 ]   ^ RK[  8 ];
  RK[  9 ] =            RK[  5 ]   ^ RK[  9 ];
  RK[ 10 ] =            RK[  6 ]   ^ RK[ 10 ];
  RK[ 11 ] =            RK[  7 ]   ^ RK[ 11 ];

  RK[ 12 ] =            RK[  8 ]   ^ RK[ 12 ];
  RK[ 13 ] =            RK[  9 ]   ^ RK[ 13 ];
  RK[ 14 ] =            RK[ 10 ]   ^ RK[ 14 ];
  RK[ 15 ] =            RK[ 11 ]   ^ RK[ 15 ];

  return fmulx( RC );
}
```

```
void aes_encrypt( U8* C, U8* M, U8* K )
{
  U8 S[ 16 ], RK[ 16 ], RC = 0x01;

  U8_TO_U8_N(  S, M );
  U8_TO_U8_N( RK, K );

  ROUND1();

  for( int i = 1; i < 10; i++ ) {
    ROUND2();
  }

  ROUND3();

  U8_TO_U8_N(  C, S );
}
```

Implementation #1 (6)

- The first component we are missing is FMUL-X (more often called `xtime`) which multiplies a field element by X ...
- ... there are two main options:
  1. Pre-compute the function offline; we end up with a 256-entry look-up table.
  2. Compute the function online; this actually isn't too hard:

```
U8 fmulx( U8 x )
{
  if( x & 0x80 )
    return  0x1B ^ ( x << 1 );
  else
    return         ( x << 1 );
}
```

```
U8 fmulx( U8 x )
{
  U8 t = x << 1;
     x = x >> 7;
     x = x *  0x1B;
     x = x ^  t;

  return x;
}
```

- The left-hand and right-hand implementations compute the same thing:
  - ... the left-shift is performing multiplication by X.
  - ... the XOR with $1B_{(16)}$ performs reduction modulo $X^8 + X^4 + X^3 + X + 1$.
- The right-hand implementation lacks any conditional branches; this can offer resistance against some side-channel attacks.

Implementation #1 (7)

- The second component we are missing is the S-box; this is defined as the composition of two functions

$$\text{S-Box}(a) = f(g(a))$$

where

$$g(a) = 1 \oslash_{\mathbb{F}_{2^8}} a,$$

i.e., g is a field inversion, and

$$f\left(\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{bmatrix}\right) = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \otimes_{\mathbb{F}_2} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{bmatrix} \oplus_{\mathbb{F}_2} \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Implementation #1 (8)

- ... there are a few options:
  1. Pre-compute the whole S-box offline (i.e., the composition of g and f) rather than compute it online; we end up with a 256-entry look-up table that supports only encryption.
  2. Compute part of the S-box (either f or g, or both) online rather than pre-compute it offline:
     - 2.1 Computing f and g is useful; we end up with no look-up table at all.
     - 2.2 Computing f but not g is useful; we end up with a 256-entry look-up table that supports both encryption and decryption.
     - 2.3 Computing g but not f isn't so useful; we'd end up with a 256-entry look-up table that supports only encryption, so may as well pre-compute the whole S-box.
- Clearly this is a classic trade-off: more pre-computation means more space, more computation means more time.

Implementation #1 (9)

- To compute f, notice that we can expand the whole thing to get:

$$
f\left(\left[\begin{array}{c} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{array}\right]\right) = \left[\begin{array}{ccccccccccccccccc}
a_0 & \oplus_{\mathbb{F}_{2^8}} & & & & & & & a_4 & \oplus_{\mathbb{F}_{2^8}} & a_5 & \oplus_{\mathbb{F}_{2^8}} & a_6 & \oplus_{\mathbb{F}_{2^8}} & a_7 & \oplus_{\mathbb{F}_{2^8}} & 1 \\
a_0 & \oplus_{\mathbb{F}_{2^8}} & a_1 & \oplus_{\mathbb{F}_{2^8}} & & & & & & & a_5 & \oplus_{\mathbb{F}_{2^8}} & a_6 & \oplus_{\mathbb{F}_{2^8}} & a_7 & \oplus_{\mathbb{F}_{2^8}} & 1 \\
a_0 & \oplus_{\mathbb{F}_{2^8}} & a_1 & \oplus_{\mathbb{F}_{2^8}} & a_2 & \oplus_{\mathbb{F}_{2^8}} & & & & & & & a_6 & \oplus_{\mathbb{F}_{2^8}} & a_7 & \oplus_{\mathbb{F}_{2^8}} & 0 \\
a_0 & \oplus_{\mathbb{F}_{2^8}} & a_1 & \oplus_{\mathbb{F}_{2^8}} & a_2 & \oplus_{\mathbb{F}_{2^8}} & a_3 & \oplus_{\mathbb{F}_{2^8}} & & & & & & & a_7 & \oplus_{\mathbb{F}_{2^8}} & 0 \\
a_0 & \oplus_{\mathbb{F}_{2^8}} & a_1 & \oplus_{\mathbb{F}_{2^8}} & a_2 & \oplus_{\mathbb{F}_{2^8}} & a_3 & \oplus_{\mathbb{F}_{2^8}} & a_4 & & & & & & & \oplus_{\mathbb{F}_{2^8}} & 0 \\
 & & a_1 & \oplus_{\mathbb{F}_{2^8}} & a_2 & \oplus_{\mathbb{F}_{2^8}} & a_3 & \oplus_{\mathbb{F}_{2^8}} & a_4 & \oplus_{\mathbb{F}_{2^8}} & a_5 & & & & & \oplus_{\mathbb{F}_{2^8}} & 1 \\
 & & & & a_2 & \oplus_{\mathbb{F}_{2^8}} & a_3 & \oplus_{\mathbb{F}_{2^8}} & a_4 & \oplus_{\mathbb{F}_{2^8}} & a_5 & \oplus_{\mathbb{F}_{2^8}} & a_6 & & & \oplus_{\mathbb{F}_{2^8}} & 1 \\
 & & & & & & a_3 & \oplus_{\mathbb{F}_{2^8}} & a_4 & \oplus_{\mathbb{F}_{2^8}} & a_5 & \oplus_{\mathbb{F}_{2^8}} & a_6 & \oplus_{\mathbb{F}_{2^8}} & a_7 & \oplus_{\mathbb{F}_{2^8}} & 0
\end{array}\right]
$$

- This looks somewhat hideous ...

Implementation #1 (10)

- ... but we can implement it with just a few XORs and shifts:

```
U8 sbox( U8 x )
{
  U8 t = finv( x );

  return ( 0x63   ) ^ // <   0,   1,   1,   0,   0,   0,   1,   1 >
         ( t      ) ^ // < t_7, t_6, t_5, t_4, t_3, t_2, t_1, t_0 >
         ( t << 1 ) ^ // < t_6, t_5, t_4, t_3, t_2, t_1, t_0,   0 >
         ( t << 2 ) ^ // < t_5, t_4, t_3, t_2, t_1, t_0,   0,   0 >
         ( t << 3 ) ^ // < t_4, t_3, t_2, t_1, t_0,   0,   0,   0 >
         ( t << 4 ) ^ // < t_3, t_2, t_1, t_0,   0,   0,   0,   0 >
         ( t >> 7 ) ^ // <   0,   0,   0,   0,   0,   0,   0, t_7 >
         ( t >> 6 ) ^ // <   0,   0,   0,   0,   0,   0, t_7, t_6 >
         ( t >> 5 ) ^ // <   0,   0,   0,   0,   0, t_7, t_6, t_5 >
         ( t >> 4 ) ; // <   0,   0,   0,   0, t_7, t_6, t_5, t_4 >
}
```

Implementation #1 (11)

- To compute g, we need to find a b such that $a \odot_{\mathbb{F}_{2^8}} b = 1$ ...
- ... there are a few options:
  1. Use brute force search; since the field is small, we can just try every element until we find a b that suits.
  2. Use a version of the Extended Euclidean Algorithm (EEA), or XGCD, to directly compute such a b.
  3. Use Fermat's little theorem; it tells us in a finite field with q elements

$$\begin{array}{lcl} a^q & = & a \\ a^{q-1} & = & 1 \\ a^{q-2} & = & a^{-1} \end{array}$$

     and hence for $q = 2^8$ we just compute $a^{254} = a^{-1}$.
  4. Decompose $\mathbb{F}_{2^8}$ into smaller fields, and compute the inversion in $\mathbb{F}_{2^8}$ as a combination of operations in said fields.

Implementation #1 (12)

- To adopt the Fermat-style approach, we need two components:
  1. A function to implement a general purpose field multiplication, i.e., $\odot_{\mathbb{F}_{2^8}}$.
  2. A function that uses $\odot_{\mathbb{F}_{2^8}}$ do exponentiation in an efficient way.

```
U8 fmul( U8 x, U8 y )
{
  U8 t = 0;

  for( int i = 7; i >= 0; i-- ) {
    t = fmulx( t );

    if( ( y >> i ) & 1 ) {
      t = t ^ x;
    }
  }

  return t;
}
```

```
U8 finv( U8 x )
{
  U8 t0 = fmul(  x,  x ); // x^2
  U8 t1 = fmul( t0,  x ); // x^3
     t0 = fmul( t0, t0 ); // x^4
     t1 = fmul( t1, t0 ); // x^7
     t0 = fmul( t0, t0 ); // x^8
     t0 = fmul( t1, t0 ); // x^15
     t0 = fmul( t0, t0 ); // x^30
     t0 = fmul( t0, t0 ); // x^60
     t1 = fmul( t1, t0 ); // x^67
     t0 = fmul( t0, t1 ); // x^127
     t0 = fmul( t0, t0 ); // x^254

  return t0;
}
```

Implementation #2 (1)

Strategy

- AES was specifically designed to be efficient on unconstrained platforms (e.g., 32-bit workstations and servers).
- An implementation strategy in this case could be:
  1. Represent S by packing it column-wise into four 32-bit words; the state is held as

     $$\begin{array}{c} \langle S_{0,0}, S_{1,0}, S_{2,0}, S_{3,0} \rangle \\ \langle S_{0,1}, S_{1,1}, S_{2,1}, S_{3,1} \rangle \\ \langle S_{0,2}, S_{1,2}, S_{2,2}, S_{3,2} \rangle \\ \langle S_{0,3}, S_{1,3}, S_{2,3}, S_{3,3} \rangle \end{array}$$

     and we can take advantage of the 32-bit data-path.
  2. Generate round keys before encryption, i.e., offline, and potentially reuse them if secret key is static.
  3. Pre-compute and specialise (e.g., unroll loops) as much as possible to make a trade-off in favour of time (i.e., performance) over space (i.e., memory footprint).

Implementation #2 (2)

- The main bottleneck is MIX-COLUMNS; first, notice that we can expand the matrix multiplication to get:

$$\textsc{Mix-Columns}\left(\left[\begin{array}{c} S_{0,j} \\ S_{1,j} \\ S_{2,j} \\ S_{3,j} \end{array}\right]\right)$$

$$\downarrow$$

$$\left[\begin{array}{c}
(02_{(16)} \odot_{\mathbb{F}_{2^8}} S_{0,j}) \oplus_{\mathbb{F}_{2^8}} (03_{(16)} \odot_{\mathbb{F}_{2^8}} S_{1,j}) \oplus_{\mathbb{F}_{2^8}} (01_{(16)} \odot_{\mathbb{F}_{2^8}} S_{2,j}) \oplus_{\mathbb{F}_{2^8}} (01_{(16)} \odot_{\mathbb{F}_{2^8}} S_{3,j}) \\
(01_{(16)} \odot_{\mathbb{F}_{2^8}} S_{0,j}) \oplus_{\mathbb{F}_{2^8}} (02_{(16)} \odot_{\mathbb{F}_{2^8}} S_{1,j}) \oplus_{\mathbb{F}_{2^8}} (03_{(16)} \odot_{\mathbb{F}_{2^8}} S_{2,j}) \oplus_{\mathbb{F}_{2^8}} (01_{(16)} \odot_{\mathbb{F}_{2^8}} S_{3,j}) \\
(01_{(16)} \odot_{\mathbb{F}_{2^8}} S_{0,j}) \oplus_{\mathbb{F}_{2^8}} (01_{(16)} \odot_{\mathbb{F}_{2^8}} S_{1,j}) \oplus_{\mathbb{F}_{2^8}} (02_{(16)} \odot_{\mathbb{F}_{2^8}} S_{2,j}) \oplus_{\mathbb{F}_{2^8}} (03_{(16)} \odot_{\mathbb{F}_{2^8}} S_{3,j}) \\
(03_{(16)} \odot_{\mathbb{F}_{2^8}} S_{0,j}) \oplus_{\mathbb{F}_{2^8}} (01_{(16)} \odot_{\mathbb{F}_{2^8}} S_{1,j}) \oplus_{\mathbb{F}_{2^8}} (01_{(16)} \odot_{\mathbb{F}_{2^8}} S_{2,j}) \oplus_{\mathbb{F}_{2^8}} (02_{(16)} \odot_{\mathbb{F}_{2^8}} S_{3,j})
\end{array}\right]$$

Implementation #2 (2)

- The main bottleneck is MIX-COLUMNS; first, notice that we can expand the matrix multiplication to get:

$$\textsc{Mix-Columns}\left(\left[\begin{array}{c} s_{0,j} \\ s_{1,j} \\ s_{2,j} \\ s_{3,j} \end{array} \right] \right)$$

$$\downarrow$$

$$\left[\begin{array}{c} 02_{(16)} \odot_{\mathbb{F}_{2^8}} s_{0,j} \\ 01_{(16)} \odot_{\mathbb{F}_{2^8}} s_{0,j} \\ 01_{(16)} \odot_{\mathbb{F}_{2^8}} s_{0,j} \\ 03_{(16)} \odot_{\mathbb{F}_{2^8}} s_{0,j} \end{array} \right] \oplus_{\mathbb{F}_{2^8}} \left[\begin{array}{c} 03_{(16)} \odot_{\mathbb{F}_{2^8}} s_{1,j} \\ 02_{(16)} \odot_{\mathbb{F}_{2^8}} s_{1,j} \\ 01_{(16)} \odot_{\mathbb{F}_{2^8}} s_{1,j} \\ 01_{(16)} \odot_{\mathbb{F}_{2^8}} s_{1,j} \end{array} \right] \oplus_{\mathbb{F}_{2^8}} \left[\begin{array}{c} 01_{(16)} \odot_{\mathbb{F}_{2^8}} s_{2,j} \\ 03_{(16)} \odot_{\mathbb{F}_{2^8}} s_{2,j} \\ 02_{(16)} \odot_{\mathbb{F}_{2^8}} s_{2,j} \\ 01_{(16)} \odot_{\mathbb{F}_{2^8}} s_{2,j} \end{array} \right] \oplus_{\mathbb{F}_{2^8}} \left[\begin{array}{c} 01_{(16)} \odot_{\mathbb{F}_{2^8}} s_{3,j} \\ 01_{(16)} \odot_{\mathbb{F}_{2^8}} s_{3,j} \\ 03_{(16)} \odot_{\mathbb{F}_{2^8}} s_{3,j} \\ 02_{(16)} \odot_{\mathbb{F}_{2^8}} s_{3,j} \end{array} \right]$$

Implementation #2 (3)

- Then, we can re-write MIX-COLUMNS as ...

$$\textsc{Mix-Columns}\left(\begin{bmatrix} S_{0,j} \\ S_{1,j} \\ S_{2,j} \\ S_{3,j} \end{bmatrix}\right)$$

$$\downarrow$$

$$T_0[S_{0,j}] \oplus_{\mathbb{F}_{2^8}} T_1[S_{1,j}] \oplus_{\mathbb{F}_{2^8}} T_2[S_{2,j}] \oplus_{\mathbb{F}_{2^8}} T_3[S_{3,j}]$$

- ... (i.e., four table look-ups and three XORs) if we pre-compute:

$$T_0[a] = \begin{bmatrix} 02_{(16)} \odot_{\mathbb{F}_{2^8}} a \\ 01_{(16)} \odot_{\mathbb{F}_{2^8}} a \\ 01_{(16)} \odot_{\mathbb{F}_{2^8}} a \\ 03_{(16)} \odot_{\mathbb{F}_{2^8}} a \end{bmatrix} \quad T_1[a] = \begin{bmatrix} 03_{(16)} \odot_{\mathbb{F}_{2^8}} a \\ 02_{(16)} \odot_{\mathbb{F}_{2^8}} a \\ 01_{(16)} \odot_{\mathbb{F}_{2^8}} a \\ 01_{(16)} \odot_{\mathbb{F}_{2^8}} a \end{bmatrix}$$

$$T_2[a] = \begin{bmatrix} 01_{(16)} \odot_{\mathbb{F}_{2^8}} a \\ 03_{(16)} \odot_{\mathbb{F}_{2^8}} a \\ 02_{(16)} \odot_{\mathbb{F}_{2^8}} a \\ 01_{(16)} \odot_{\mathbb{F}_{2^8}} a \end{bmatrix} \quad T_3[a] = \begin{bmatrix} 01_{(16)} \odot_{\mathbb{F}_{2^8}} a \\ 01_{(16)} \odot_{\mathbb{F}_{2^8}} a \\ 03_{(16)} \odot_{\mathbb{F}_{2^8}} a \\ 02_{(16)} \odot_{\mathbb{F}_{2^8}} a \end{bmatrix}$$

Implementation #2 (4)

- Even better, in rounds 1 to 9 we always apply SUB-BYTES before MIX-COLUMNS ...
- ... so at no extra cost, we can push S-BOX into the T-tables as well, i.e.,

$$T_0[a] = \begin{bmatrix} 02_{(16)} \odot_{\mathbb{F}_{2^8}} \text{S-Box}(a) \\ 01_{(16)} \odot_{\mathbb{F}_{2^8}} \text{S-Box}(a) \\ 01_{(16)} \odot_{\mathbb{F}_{2^8}} \text{S-Box}(a) \\ 03_{(16)} \odot_{\mathbb{F}_{2^8}} \text{S-Box}(a) \end{bmatrix} \quad T_1[a] = \begin{bmatrix} 03_{(16)} \odot_{\mathbb{F}_{2^8}} \text{S-Box}(a) \\ 02_{(16)} \odot_{\mathbb{F}_{2^8}} \text{S-Box}(a) \\ 01_{(16)} \odot_{\mathbb{F}_{2^8}} \text{S-Box}(a) \\ 01_{(16)} \odot_{\mathbb{F}_{2^8}} \text{S-Box}(a) \end{bmatrix}$$

$$T_2[a] = \begin{bmatrix} 01_{(16)} \odot_{\mathbb{F}_{2^8}} \text{S-Box}(a) \\ 03_{(16)} \odot_{\mathbb{F}_{2^8}} \text{S-Box}(a) \\ 02_{(16)} \odot_{\mathbb{F}_{2^8}} \text{S-Box}(a) \\ 01_{(16)} \odot_{\mathbb{F}_{2^8}} \text{S-Box}(a) \end{bmatrix} \quad T_3[a] = \begin{bmatrix} 01_{(16)} \odot_{\mathbb{F}_{2^8}} \text{S-Box}(a) \\ 01_{(16)} \odot_{\mathbb{F}_{2^8}} \text{S-Box}(a) \\ 03_{(16)} \odot_{\mathbb{F}_{2^8}} \text{S-Box}(a) \\ 02_{(16)} \odot_{\mathbb{F}_{2^8}} \text{S-Box}(a) \end{bmatrix}$$

Implementation #2 (5)

```
#define ROUND2(a,b,c,d)                                          \
{                                                                \
  t4 = ( T0[ ( t0 >>  0 ) & 0xFF ] ) ^                           \
       ( T1[ ( t1 >>  8 ) & 0xFF ] ) ^                           \
       ( T2[ ( t2 >> 16 ) & 0xFF ] ) ^                           \
       ( T3[ ( t3 >> 24 ) & 0xFF ] ) ^ RK[ a ];                  \
  t5 = ( T0[ ( t1 >>  0 ) & 0xFF ] ) ^                           \
       ( T1[ ( t2 >>  8 ) & 0xFF ] ) ^                           \
       ( T2[ ( t3 >> 16 ) & 0xFF ] ) ^                           \
       ( T3[ ( t0 >> 24 ) & 0xFF ] ) ^ RK[ b ];                  \
  t6 = ( T0[ ( t2 >>  0 ) & 0xFF ] ) ^                           \
       ( T1[ ( t3 >>  8 ) & 0xFF ] ) ^                           \
       ( T2[ ( t0 >> 16 ) & 0xFF ] ) ^                           \
       ( T3[ ( t1 >> 24 ) & 0xFF ] ) ^ RK[ c ];                  \
  t7 = ( T0[ ( t3 >>  0 ) & 0xFF ] ) ^                           \
       ( T1[ ( t0 >>  8 ) & 0xFF ] ) ^                           \
       ( T2[ ( t1 >> 16 ) & 0xFF ] ) ^                           \
       ( T3[ ( t2 >> 24 ) & 0xFF ] ) ^ RK[ d ];                  \
                                                                 \
  t0 = t4;                                                       \
  t1 = t5;                                                       \
  t2 = t6;                                                       \
  t3 = t7;                                                       \
}
```

Implementation #2 (6)

```
void aes_schedule( U32* RK, U8* K )
{
  U32 t0, t1, t2, t3;

  U8_TO_U32( t0, K,  0 ); U8_TO_U32( t1, K,  4 );
  U8_TO_U32( t2, K,  8 ); U8_TO_U32( t3, K, 12 );

  RK[ 0 ] = t0;
  RK[ 1 ] = t1;
  RK[ 2 ] = t2;
  RK[ 3 ] = t3;

  for( int i = 1; i < 11; i++ ) {
    t0 = t0 ^ ( T4[ ( t3 >>  8 ) & 0xFF ] & 0x000000FF ) ^
              ( T4[ ( t3 >> 16 ) & 0xFF ] & 0x0000FF00 ) ^
              ( T4[ ( t3 >> 24 ) & 0xFF ] & 0x00FF0000 ) ^
              ( T4[ ( t3 >>  0 ) & 0xFF ] & 0xFF000000 ) ^ RC[ i - 1 ];

    t1 = t0 ^ t1;
    t2 = t1 ^ t2;
    t3 = t2 ^ t3;

    RK[ ( 4 * i) + 0 ] = t0;
    RK[ ( 4 * i) + 1 ] = t1;
    RK[ ( 4 * i) + 2 ] = t2;
    RK[ ( 4 * i) + 3 ] = t3;
  }
}
```

Implementation #2 (7)

```
void aes_encrypt( U8* C, U8* M, U32* RK )
{
  U32 t0, t1, t2, t3, t4, t5, t6, t7;

  U8_TO_U32( t0, M,  0 ); U8_TO_U32( t1, M,  4 );
  U8_TO_U32( t2, M,  8 ); U8_TO_U32( t3, M, 12 );

  ROUND1(  0,  1,  2,  3 );
  ROUND2(  4,  5,  6,  7 ); ROUND2(  8,  9, 10, 11 ); ROUND2( 12, 13, 14, 15 );
  ROUND2( 16, 17, 18, 19 ); ROUND2( 20, 21, 22, 23 ); ROUND2( 24, 25, 26, 27 );
  ROUND2( 28, 29, 30, 31 ); ROUND2( 32, 33, 34, 35 ); ROUND2( 36, 37, 38, 39 );
  ROUND3( 40, 41, 42, 43 );

  U32_TO_U8( C, t4,  0 ); U32_TO_U8( C, t5,  4 );
  U32_TO_U8( C, t6,  8 ); U32_TO_U8( C, t7, 12 );
}
```

Implementation #3 (1)

Strategy

- It can make sense to implement AES using a mix of the high-performance and low-footprint approaches we've looked at ...
- ... for example, if you have an embedded processor without much memory but with a 32-bit data-path (e.g., ARM7).
- An implementation strategy in this case could be:
  1. Represent S by packing it row-wise into four 32-bit words; the state is held as

$$\begin{array}{c} \langle S_{0,0}, S_{0,1}, S_{0,2}, S_{0,3} \rangle \\ \langle S_{1,0}, S_{1,1}, S_{1,2}, S_{1,3} \rangle \\ \langle S_{2,0}, S_{2,1}, S_{2,2}, S_{2,3} \rangle \\ \langle S_{3,0}, S_{3,1}, S_{3,2}, S_{3,3} \rangle \end{array}$$

     and we can take advantage of the 32-bit data-path. Another way to think about this is that we've just transposed the state matrix (and round keys).
  2. Rather than pre-computing the T-tables, use the low-footprint approach of computing round functions directly.

Implementation #3 (2)

- The bottleneck is again MIX-COLUMNS; we can attempt to improve performance using packed operations ...
- ... the idea is to compute applications of FMUL-X to all four elements of the packed vector in parallel; this permits MIX-COLUMNS to be more efficient:

Algorithm (PACKED-FMUL-X)

Input: A packed 32-bit state matrix row x.
Output: A packed 32-bit state matrix row x' with $x'_i = \text{FMUL-X}(x_i)$ for $0 \leq i < 4$.

$t_0 \leftarrow x \wedge 7F7F7F7F_{(16)}$
$t_1 \leftarrow x \wedge 80808080_{(16)}$
$t_2 \leftarrow t_0 \ll 1$
$t_3 \leftarrow t_1 \gg 7$
$t_4 \leftarrow t_3 \cdot 1B_{(16)}$
$t_5 \leftarrow t_2 \oplus t_4$

return t_5

Algorithm (MIX-COLUMNS)

Input: Four packed 32-bit state matrix rows x_i, for $0 \leq i < 4$.
Output: Four packed 32-bit state matrix rows x'_i with $x'_i = \text{MIX-COLUMNS}(x_i)$ for $0 \leq i < 4$.

$y_0 \leftarrow x_1 \oplus x_2 \oplus x_3$
$y_1 \leftarrow x_0 \oplus x_2 \oplus x_3$
$y_2 \leftarrow x_0 \oplus x_1 \oplus x_3$
$y_3 \leftarrow x_0 \oplus x_1 \oplus x_2$

$x_0 \leftarrow \text{PACKED-FMUL-X}(x_0)$
$x_1 \leftarrow \text{PACKED-FMUL-X}(x_1)$
$x_2 \leftarrow \text{PACKED-FMUL-X}(x_2)$
$x_3 \leftarrow \text{PACKED-FMUL-X}(x_3)$

$y_0 \leftarrow y_0 \oplus x_0 \oplus x_1$
$y_1 \leftarrow y_1 \oplus x_1 \oplus x_2$
$y_2 \leftarrow y_2 \oplus x_2 \oplus x_3$
$y_3 \leftarrow y_3 \oplus x_0 \oplus x_3$

return y_0, y_1, y_2, y_3

Implementation $\#3$ (3)

- As a (short) example of why this works consider the computation of y_0, from x_0, x_1, x_2 and x_3, in isolation ...
- ... first, we compute:

$$
\begin{array}{rcl}
x_0 = & \langle & s_{0,0} \;,\; s_{0,1} \;,\; s_{0,2} \;,\; s_{0,3} \;\rangle \\
x_1 = & \langle & s_{1,0} \;,\; s_{1,1} \;,\; s_{1,2} \;,\; s_{1,3} \;\rangle \\
x_2 = & \langle & s_{2,0} \;,\; s_{2,1} \;,\; s_{2,2} \;,\; s_{2,3} \;\rangle \\
x_3 = & \langle & s_{3,0} \;,\; s_{3,1} \;,\; s_{3,2} \;,\; s_{3,3} \;\rangle
\end{array}
$$

$$
y_0 = x_1 \oplus x_2 \oplus x_3 = \left\langle \begin{array}{c} s_{1,0} \\ \oplus \\ s_{2,0} \\ \oplus \\ s_{3,0} \end{array} , \begin{array}{c} s_{1,1} \\ \oplus \\ s_{2,1} \\ \oplus \\ s_{3,1} \end{array} , \begin{array}{c} s_{1,2} \\ \oplus \\ s_{2,2} \\ \oplus \\ s_{3,2} \end{array} , \begin{array}{c} s_{1,3} \\ \oplus \\ s_{2,3} \\ \oplus \\ s_{3,3} \end{array} \right\rangle
$$

$$
\begin{array}{l}
x_0 = \textsc{Packed-FMul-X}(x_0) = \langle 02_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{0,0} , 02_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{0,1} , 02_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{0,2} , 02_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{0,3} \rangle \\
x_1 = \textsc{Packed-FMul-X}(x_1) = \langle 02_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{1,0} , 02_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{1,1} , 02_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{1,2} , 02_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{1,3} \rangle \\
x_2 = \textsc{Packed-FMul-X}(x_2) = \langle 02_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{2,0} , 02_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{2,1} , 02_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{2,2} , 02_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{2,3} \rangle \\
x_3 = \textsc{Packed-FMul-X}(x_3) = \langle 02_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{3,0} , 02_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{3,1} , 02_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{3,2} , 02_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{3,3} \rangle
\end{array}
$$

Implementation #3 (4)

- Then finally we compute y_0, i.e., the first row of the output from MIX-COLUMNS, as follows ...

$$
y_0 = y_0 \oplus x_0 \oplus x_1 = \left\langle
\begin{array}{c}
s_{1,0} \\ \oplus \\ s_{2,0} \\ \oplus \\ s_{3,0} \\ \oplus \\ 02_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{0,0} \\ \oplus \\ 02_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{1,0}
\end{array},
\begin{array}{c}
s_{1,1} \\ \oplus \\ s_{2,1} \\ \oplus \\ s_{3,1} \\ \oplus \\ 02_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{0,1} \\ \oplus \\ 02_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{1,1}
\end{array},
\begin{array}{c}
s_{1,2} \\ \oplus \\ s_{2,2} \\ \oplus \\ s_{3,2} \\ \oplus \\ 02_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{0,2} \\ \oplus \\ 02_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{1,2}
\end{array},
\begin{array}{c}
s_{1,3} \\ \oplus \\ s_{2,3} \\ \oplus \\ s_{3,3} \\ \oplus \\ 02_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{0,3} \\ \oplus \\ 02_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{1,3}
\end{array}
\right\rangle
$$

$$
= \left\langle
\begin{array}{c}
02_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{0,0} \\ \oplus \\ 03_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{1,0} \\ \oplus \\ 01_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{2,0} \\ \oplus \\ 01_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{3,0}
\end{array},
\begin{array}{c}
02_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{0,1} \\ \oplus \\ 03_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{1,1} \\ \oplus \\ 01_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{2,1} \\ \oplus \\ 01_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{3,1}
\end{array},
\begin{array}{c}
02_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{0,2} \\ \oplus \\ 03_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{1,2} \\ \oplus \\ 01_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{2,2} \\ \oplus \\ 01_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{3,2}
\end{array},
\begin{array}{c}
02_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{0,3} \\ \oplus \\ 03_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{1,3} \\ \oplus \\ 01_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{2,3} \\ \oplus \\ 01_{(16)} \otimes_{\mathbb{F}_{2^8}} s_{3,3}
\end{array}
\right\rangle
$$

- ... noting that this matches the expanded matrix multiplication we originally wrote down, i.e., the right elements are multiplied by the right constants.

Implementation #3 (5)

```
void aes_key( U8* S, U8* RK )
{
  U32* Sp = ( U32* )(  S );
  U32* Kp = ( U32* )( RK );

  Sp[ 0 ] = Sp[ 0 ] ^ Kp[ 0 ];
  Sp[ 1 ] = Sp[ 1 ] ^ Kp[ 1 ];
  Sp[ 2 ] = Sp[ 2 ] ^ Kp[ 2 ];
  Sp[ 3 ] = Sp[ 3 ] ^ Kp[ 3 ];
}
```

```
void aes_sub( U8* S )
{
  S[  0 ] = sbox( S[  0 ] );
  S[  1 ] = sbox( S[  1 ] );
  S[  2 ] = sbox( S[  2 ] );
  S[  3 ] = sbox( S[  3 ] );

  S[  4 ] = sbox( S[  4 ] );
  S[  5 ] = sbox( S[  5 ] );
  S[  6 ] = sbox( S[  6 ] );
  S[  7 ] = sbox( S[  7 ] );

  S[  8 ] = sbox( S[  8 ] );
  S[  9 ] = sbox( S[  9 ] );
  S[ 10 ] = sbox( S[ 10 ] );
  S[ 11 ] = sbox( S[ 11 ] );

  S[ 12 ] = sbox( S[ 12 ] );
  S[ 13 ] = sbox( S[ 13 ] );
  S[ 14 ] = sbox( S[ 14 ] );
  S[ 15 ] = sbox( S[ 15 ] );
}
```

Implementation #3 (6)

```
void aes_row( U8* S )
{
  U32* Sp = ( U32* )( S );

  Sp[ 1 ] = rotr_32( Sp[ 1 ],  8 );
  Sp[ 2 ] = rotr_32( Sp[ 2 ], 16 );
  Sp[ 3 ] = rotr_32( Sp[ 3 ], 24 );
}
```

```
void aes_mix( U8* S )
{
  U32* Sp = ( U32* )( S );

  U32  t0 = Sp[ 0 ], t1 = Sp[ 1 ];
  U32  t2 = Sp[ 2 ], t3 = Sp[ 3 ];

  U32  t4 = t1 ^ t2 ^ t3;
  U32  t5 = t0 ^ t2 ^ t3;
  U32  t6 = t0 ^ t1 ^ t3;
  U32  t7 = t0 ^ t1 ^ t2;

       t0 = fmulx( t0 );
       t1 = fmulx( t1 );
       t2 = fmulx( t2 );
       t3 = fmulx( t3 );

       t4 = t4 ^ t0 ^ t1;
       t5 = t5 ^ t1 ^ t2;
       t6 = t6 ^ t2 ^ t3;
       t7 = t7 ^ t0 ^ t3;

       Sp[ 0 ] = t4; Sp[ 1 ] = t5;
       Sp[ 2 ] = t6; Sp[ 3 ] = t7;
}
```

Conclusions

- One of the criteria for selecting the AES was flexibility:
  - ... efficiency on constrained platforms (e.g., 8-bit smart-cards).
  - ... efficiency on unconstrained platforms (e.g., 32-bit workstations and servers)
  - ... efficiency in hardware.
  - ... resistance against physical attack.
- In part, Rijndael was selected as the AES because it allows a broad range of effective implementation approaches; in part, this is because of flexibility in the more direct mathematical underpinnings (compared to DES).
- AES is a fairly young algorithm (compared to DES), and new approaches (e.g., vectorisation, bit-slicing) are still being developed and evaluated.

Further Reading

- National Institute of Standards and Technology (NIST). FIPS-197: Advanced Encryption Standard (AES), 2001.
  `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.
- J. Daemen and V. Rijmen. *The Design of Rijndael*. Springer-Verlag, 2002. ISBN: 3-540-42580-2.
- D.J. Bernstein and P. Schwabe. New AES Software Speed Records. In *Progress in Cryptology (INDOCRYPT)* Springer-Verlag LNCS 5365, 322–336, 2008.
- G. Bertoni, L. Breveglieri, P. Fragneto, M. Macchetti and S. Marchesin. Efficient Software Implementation of AES on 32-Bit Platforms. In *Cryptographic Hardware and Embedded Systems (CHES)*, Springer-Verlag LNCS 2523, 159–171, 2002.