Cryptographic Engineering

Efficient and secure cryptographic implementations

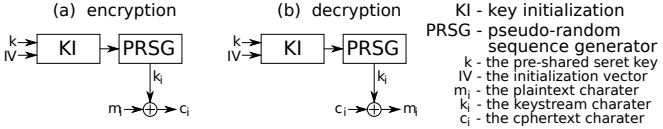
March 4th, 2024

Nuša Zidarič, LIACS, Leiden University
Nele Mentens, LIACS, Leiden University

Symmetric Key Cryptography
&
Aeuthenticated Encryption with Associated Data

Stream Ciphers and PRSGs

- Stream ciphers enc/decrypt a messagestream - one character at a time (bit, word,...)
- One-time pad: `XOR` (must change IV to prevent two-time pad attack!)
- Reduction: security of the cipher completely depends on the keystream $\Rightarrow$ Crytpographic PRSG[1]



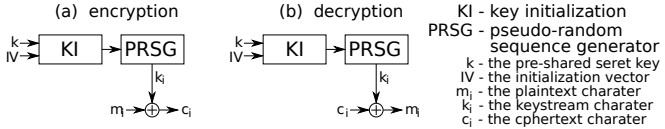
Behavioral model of a stream cipher: (a) encryption and (b) decryption

[1] L.Chen and G. Gong, Communication System Security, Boca Raton FL: CRC Press, 2012

[1] PRSG - Pseudo-Random Sequence Generator

Stream Ciphers and PRSGs

- Stream ciphers enc/decrypt a messagestream - one character at a time (bit, word,...)
- One-time pad: `XOR` (must change IV to prevent two-time pad attack!)
- Reduction: security of the cipher completely depends on the keystream ⇒ Crytpographic PRSG[1]



Behavioral model of a stream cipher: (a) encryption and (b) decryption

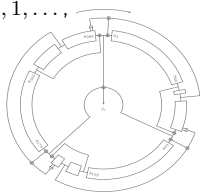
[1] L.Chen and G. Gong, Communication System Security, Boca Raton FL: CRC Press, 2012

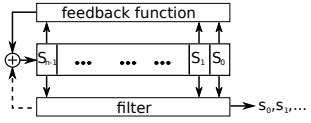
- Design principles:
  - large period
  - good randomness properties (Golomb's randomness postulates, NIST test suite, ...)
  - good cryptographic properties (algebraic degree, nonlinearity, ...)
  - ability to resist known attacks
  - high throughput, efficient in SW and in HW

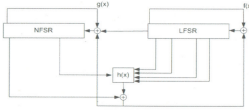
[1] PRSG - Pseudo-Random Sequence Generator

Stream Ciphers and PRSGs

- Feedback Shift Register - FSR
- generates a sequence $\underline{s} = \{s_k\} = s_0, s_1, s_2, \ldots$, where $s_{k+n} = f(s_k, s_{k+1}, \ldots, s_{k+n-1})$ for $k = 0, 1, \ldots,$
- f is a multivariate function in $t = n$ variables $x_0, x_1, \ldots, x_{n-1}$
  - linear case: $deg(f) = 1 \Rightarrow$ LFSR: primitive polynomial over $\mathbb{F}_q$: $h(y) = y^n + \sum_{j=0}^{n-1} c_j y^j$, where y^j corresponds to the stage S_j
    do NOT use stand-alone, alway combine with non-linear filter! exception: round constants
  - non-linear case: $deg(f) > 1$: mostly used as non-linear filters[1]

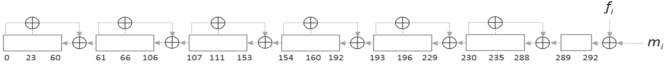






Top level schematic of a n-stage FSR with a filter

- eSTREAM: Trivium, Grain, Mickey, Sosemanuk, CAESAR: ACORN, NIST-LWC: Grain-128AEAD, WAGE
- mobile networks: Snow3G, ZUC,



[1] example of cryptographic properties: N. Zidaric, K. Mandal, G. Gong, M. Aagaard, *The welch-gong stream cipher - evolutionary path*, Cryptogr.Commun., June 2023

Hash Functions

- produces a short (prescribed length) digest (fingerprint) of a message → compression
- Strict Avalanche Condition (SAC): if 1 input bit changes → 50% of the digest changes
- properties: preimage resistance, 2nd preimage resistance, collision resistance
- unkeyed hash: data integrity
  - anyone can generate and verify (anyone can forge!)
  - examples: MDx, SHAx
- keyed hash: data integrity + source integrity (authentication) = non-repudiation
  - must prevent existential and selective forgery, use seq. # to avoid replay attacks
  - examples: HMAC, GHASH
  - usually not a cryptographic primitive, but constructed using primitives (e.g., AES)

Using a Block Cipher for Encryption

- desired security properties for block ciphers
  - ideal block cipher = random permutation
  - key size → to prevent exhaustive search attacks
  - block size → to prevent statistical attacks
  - confusion → to make statistical relationship between input and output as complex as possible
  - diffusion → to spread the effects of each input bit throughout the entire output (strict avalanche condition)

Using a Block Cipher for Encryption

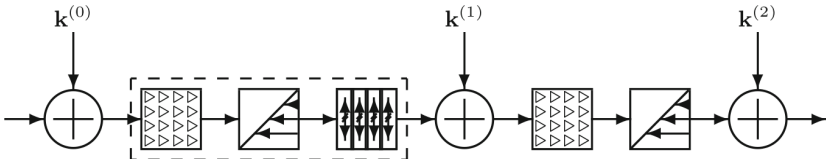
- desired security properties for block ciphers
  - ideal block cipher = random permutation
  - key size → to prevent exhaustive search attacks
  - block size → to prevent statistical attacks
  - confusion → to make statistical relationship between input and output as complex as possible
  - diffusion → to spread the effects of each input bit throughout the entire output (strict avalanche condition)
- AES has an SPN structure (Substitution-Permutation Network)
  - principle: take a simple round function with sufficient number of rounds
  - round function: SubBytes→ ShiftRows→ MixColumns→ AddRoundKey
  - substitution: SubBytes - apply an Sbox to each byte of the current state
  - permutation: ShiftRows and MixColumns - to spread the output of individual Sbox in current round to as many Sboxes in next round as possible

Using a Block Cipher for Encryption

- desired security properties for block ciphers
  - ideal block cipher = random permutation
  - key size → to prevent exhaustive search attacks
  - block size → to prevent statistical attacks
  - confusion → to make statistical relationship between input and output as complex as possible
  - diffusion → to spread the effects of each input bit throughout the entire output (strict avalanche condition)
- AES has an SPN structure (Substitution-Permutation Network)
  - principle: take a simple round function with sufficient number of rounds
  - round function: SubBytes→ ShiftRows→ MixColumns→ AddRoundKey
  - substitution: SubBytes - apply an Sbox to each byte of the current state
  - permutation: ShiftRows and MixColumns - to spread the output of individual Sbox in current round to as many Sboxes in next round as possible
- (a few) Sbox design principles:
  - nonlinear ($S(a) + S(b) \neq S(a + b)$)
  - output ≠ linear combination of input bits ($S(a) \neq L(a)$)
  - if two inputs differ in exactly one bit → their outputs must differ in at least 2 bits ($HD(a, b) = 1 \rightarrow HD(S(a), S(b)) \geq 2$)

Introduction to AES

- AES = Advanced Encryption Standard, standardized in 2001
  `https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197-upd1.pdf`
- SPN with key whitening (K_0 added before the first round)
- block length: 128bits, state S $= 4 \times 4$ byte array
- key length: 128bits (10 rounds), also possible 192, 256 versions
- generic round structure:
  - SubBytes (nonlinear, confusion)
  - ShiftRows (linear, diffusion)
  - MixColumns (linear, diffusion)[1]



[1] J. Daemen, V. Rijmen, *The Design of Rijndael*, 2.ed, Springer Berlin Heidelberg, 2020

[1] no MixColumns in last round

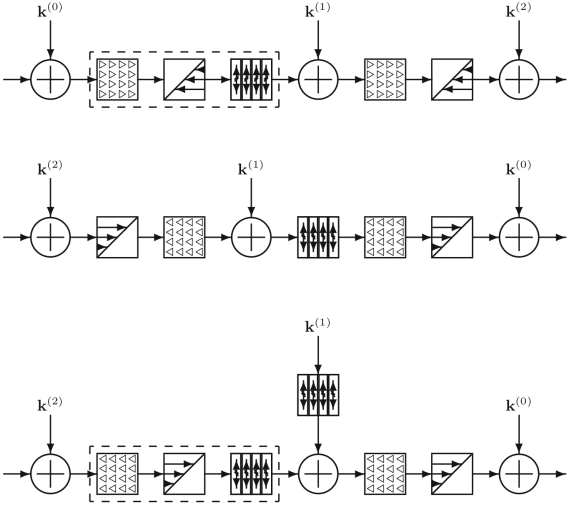


Fig. 3.12. Graphical representation of the algorithm for a two-round Rijndael variant: encryption *(top)*, decryption in the straightforward way *(middle)* and decryption in the equivalent way *(bottom)*. *Dashed boxes* enclose operations that can be implemented together efficiently

[1] J. Daemen, V. Rijmen, *The Design of Rijndael*, 2.ed, Springer Berlin Heidelberg, 2020

There are 10 kinds of people on this world ...
... those who read binary, and those who dont!

```
static const uint8_t sbox[256] = {
  //0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
  0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
  0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,
  0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
  0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,
  0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,
  0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
  0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,
  0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,
  0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
  0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,
  0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
  0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
  0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
  0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,
  0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
  0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16 };
```

to represent elements of the vector space $\mathbb{F}_2^8$, we need a basis

$$B = \{x^7, x^6, x^5, x^4, x^3, x^2, x, 1\}$$

$\mathbb{F}_2^8$ means coordinates from $\mathbb{F}_2 = \{0, 1\}$, dimension $m = 8$
$\mathbb{F}_2$ is closed for addition and multiplication modulo 2

AES Sbox

8-bit HEX notation
sbox(0x00)=0x63

2^3 2^2 2^1 2^0 2^3 2^2 2^1 2^0
0 1 1 0 0 0 1 1

binary vectors
of length $m = 8$

vector space over $\mathbb{F}_2$
of dimension $m = 8$

x^7 x^6 x^5 x^4 x^3 x^2 x^1 x^0
0 1 1 0 0 0 1 1

vector·basis:

$x^6 + x^5 + x + 1$
polynomial in $\mathbb{F}_{2^8}$

Vector Space $\mathbb{F}_2^m \Leftrightarrow \mathbb{F}_{2^m}$ Finite Field

Math behind AES

- defined over $\mathbb{F}_2[x]/(f)$, where $f(x) = x^8 + x^4 + x^3 + x + 1$ (root α, not primitive)
- polynomials basis $B = \{\alpha^7, \ldots, \alpha, 1\}$ ("downto" direction)
- SubBytes - Sbox design: works on one element of the state $A = s_{i,j} \in \mathbb{F}_{2^8}$
  - multiplicative inverse in $\mathbb{F}_{2^8}$, with $0 \mapsto 0$ (good cryptographic properties!)
  - affine transformation: linearized polynomial $L(x) + c$, where c is a constant
    $L(x) = \sum_{i=0}^{7} \gamma_i x^{2^i}$, where $\gamma_i \in \mathbb{F}_{2^8}$, exact coefficients can be found in [1]
  - for implementation we use the matrix form of $L(x)$

$$SubBytes(A) = Z = \begin{bmatrix} z_7 \\ z_6 \\ z_5 \\ z_4 \\ z_3 \\ z_2 \\ z_1 \\ z_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} y_7 \\ y_6 \\ y_5 \\ y_4 \\ y_3 \\ y_2 \\ y_1 \\ y_0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}, \text{ where } Y = A^{-1}$$

$$InvSubBytes(A) = Z = Y^{-1}, \text{ where } \begin{bmatrix} y_7 \\ y_6 \\ y_5 \\ y_4 \\ y_3 \\ y_2 \\ y_1 \\ y_0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

AES Sbox

There are 10 kinds of people on this world ...
... those who read binary, and those who dont!

```
static const uint8_t sbox[256] = {
  //0     1     2     3     4     5     6     7     8     9     A     B     C     D     E     F
  0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
  0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,
  0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
  0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,
  0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,
  0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
  0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,
  0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,
  0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
  0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,
  0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
  0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
  0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
  0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,
  0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
  0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16 };
```

8-bit HEX notation
sbox(0x00)=0x63

$2^3\ 2^2\ 2^1\ 2^0\ 2^3\ 2^2\ 2^1\ 2^0$
0 1 1 0 0 0 1 1

binary vectors
of length $m = 8$

vector space over $\mathbb{F}_2$
of dimension $m = 8$

$x^7\ x^6\ x^5\ x^4\ x^3\ x^2\ x^1\ x^0$
0 1 1 0 0 0 1 1
the constant from
$L(Y) + c$

0x01→ $A = 1 \to Y = 1 \to L = x^6 + x^5 + x^4 + x^3 + x^2 \to 01111100 \to$0x7c

0x02→ $A = x \to Y = x^7 + x^3 + x^2 + 1 \to L = x^6 + x^5 + x^4 + x^2 + x + 1 \to 01110111 \to$0x77

0xf0→ $A = x^7 + x^6 + x^5 + x^4 \to Y = x^6 + x^4 + x^3 + x + 1 \to L = x^7 + x^3 + x^2 \to 10001100 \to$0x78c

test: $(x^7 + x^6 + x^5 + x^4)(x^6 + x^4 + x^3 + x + 1) = x^{13} + x^{12} + x^{10} + x^8 + x^7 + x^4 =$

$= \underline{x^6 + x^3 + x^2 + 1} + \underline{x^7 + x^5 + x^3 + x + 1} + \underline{x^6 + x^5 + x^3 + x^2} + \underline{x^4 + x^3 + x + 1} + x^7 + x^4 = 1$

Math behind AES- continued

- ShiftRows: rotate each row of state S by different amount

row i	ShiftRows	InvShiftRows
0	–	$\ggg 3$ ⟶
1	$\lll 1$ ⟵	$\ggg 2$ ⟶
2	$\lll 2$ ⟵	$\ggg 1$ ⟶
3	$\lll 3$ ⟵	–

- MixColumns: treat each column of the state S as a polynomial over $\mathbb{F}_{2^8}$
  - column $j \rightarrow$ polynomial $s_j(x) = s_{0,j}x^3 + s_{1,j}x^2 + s_{2,j}x + s_{3,j}$
  - new column j is computed as $s_j(x) \cdot c(x) \mod (x^4 + 1) \rightarrow$ NOT irreducible $\Rightarrow$ ring (not a field!)
  - constant polynomial $c(x) = c_0x^3 + c_1x^2 + c_2x + c_3$, where $c_0 = 0x03$, $c_1 = 0x01$, $c_2 = 0x01$, $c_3 = 0x02$

$$\begin{aligned} c(x) &= c_0x^3 + c_1x^2 + c_2x + c_3 \\ xc(x) &= c_0x^4 + c_1x^3 + c_2x^2 + c_3x \\ x^2c(x) &= c_2x^3 + c_3x^2 + c_0x + c_1 \\ x^3c(x) &= c_3x^3 + c_0x^2 + c_1x + c_2 \end{aligned}$$

$$MixColumn(S_j) = \begin{bmatrix} z_{0,j} \\ z_{1,j} \\ z_{2,j} \\ z_{3,j} \end{bmatrix} = \begin{bmatrix} 0x02 & 0x03 & 0x01 & 0x01 \\ 0x01 & 0x02 & 0x03 & 0x01 \\ 0x01 & 0x01 & 0x02 & 0x03 \\ 0x03 & 0x01 & 0x01 & 0x02 \end{bmatrix} \cdot \begin{bmatrix} s_{0,j} \\ s_{1,j} \\ s_{2,j} \\ s_{3,j} \end{bmatrix}$$

- recall: coefficients $c_j \in \mathbb{F}_{2^8}$ with basis $B = \{\alpha^7, \ldots, \alpha, 1\}$, where α root of $f(x) = x^8 + x^4 + x^3 + x + 1$

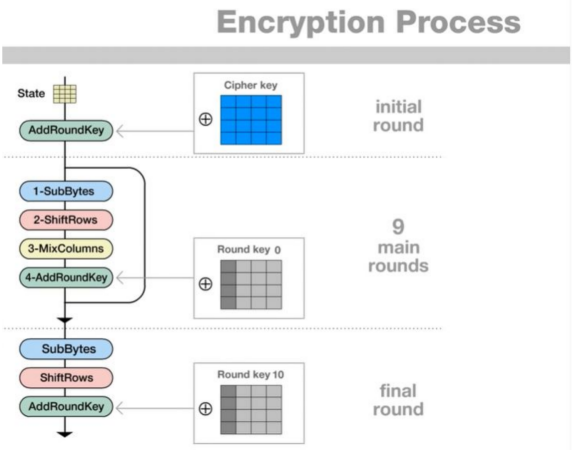
  0x02 $\rightarrow \alpha$, 0x01 $\rightarrow 1$, 0x03 $\rightarrow \alpha + 1$

$c(x)$ is chosen to be coprime with $x^4 + 1$ to make it invertible $\rightarrow$

$\rightarrow$ InvMixColumns: $c^{-1}(x) = c'_0x^3 + c'_1x^2 + c'_2x + c'_3$, where $c'_0 = 0x0B$, $c'_1 = 0x0D$, $c'_2 = 0x09$, $c'_3 = 0x0E$

$$\begin{bmatrix} \alpha & 1+\alpha & 1 & 1 \\ 1 & \alpha & 1+\alpha & 1 \\ 1 & 1 & \alpha & 1+\alpha \\ 1+\alpha & 1 & 1 & \alpha \end{bmatrix}$$

AES Overview



AddRoundKey: bitwise XOR

ShiftRows: shift

MixColumns: AND and XOR

SubBytes:

- Software: in chunks
- Hardware: in parallel (LUTs can be configured as dedicated shift registers)

- State is chopped into chunks of 8 bits
- Each 8-bit value is substituted by another 8-bit value by using an S-box
- The S-box is the most critical part in the efficient implementation of AES

AES animation: `https://www.youtube.com/watch?v=gP4PqVGudtg`

AES T-Table Implementation

T-table implementation approach:

- Combine the round functions into 4 larger look-up tables with 8 input bits and 32 output bits

$$T_0[x] = \begin{bmatrix} S[x] \times 02 \\ S[x] \\ S[x] \\ S[x] \times 03 \end{bmatrix} \quad T_1[x] = \begin{bmatrix} S[x] \times 03 \\ S[x] \times 02 \\ S[x] \\ S[x] \end{bmatrix}$$

$$T_2[x] = \begin{bmatrix} S[x] \\ S[x] \times 03 \\ S[x] \times 02 \\ S[x] \end{bmatrix} \quad T_3[x] = \begin{bmatrix} S[x] \\ S[x] \\ S[x] \times 03 \\ S[x] \times 02 \end{bmatrix}$$

$$E_j = K_{r[j]} \oplus T_0[a_{0,j}] \oplus T_1[a_{1,(j+1 \bmod 4)}] \oplus T_2[a_{2,(j+2 \bmod 4)}] \oplus T_3[a_{3,(j+3 \bmod 4)}]$$

J. Daemen and V. Rijmen, The Design of Rijndael – The Advanced Encryption Standard (AES), 2002

AES software implementations

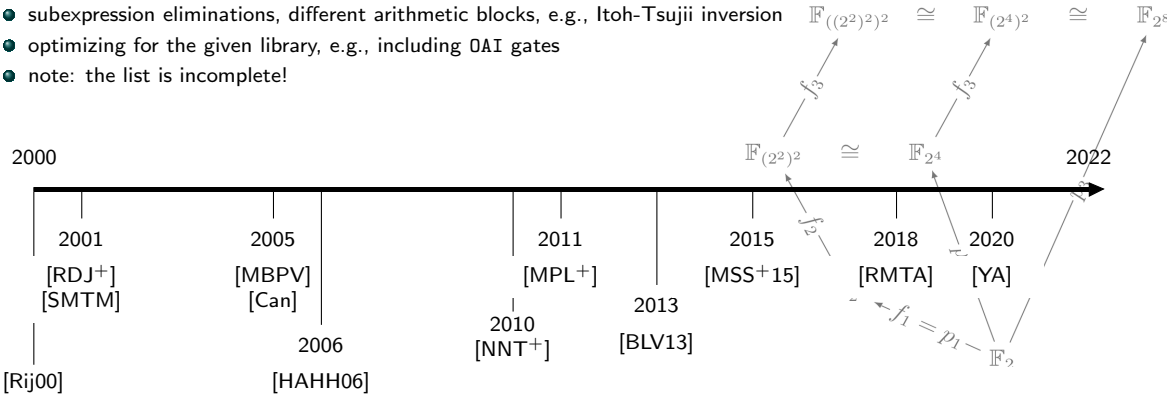
- A short (and incomplete) list of AES software implementations:

  NOTE: we have not checked the functional correctness! please check before using

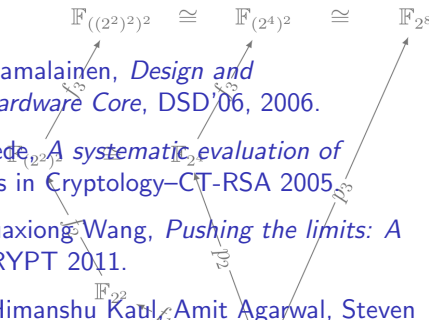
  - LUT-based Sbox: `https://github.com/kokke/tiny-AES-c`
  - 4 implementation approaches: `https://github.com/Ko-/aes-armcortexm`
  - Bitslicing implementations: `https://github.com/PQClean/PQClean/tree/master/common/aes.c`
    `https://github.com/google/adiantum/blob/master/benchmark/src/aes.c`
  - T-table implementation: `https://github.com/pjok1122/AES-Optimization/blob/master/aes.c`
  - masked AES implementation: `https://github.com/CENSUS/masked-aes-c/blob/main/aes.c`
- Some hardware implementations:
  - `https://github.com/lowRISC/opentitan/tree/master/hw/ip/aes`
  - `https://github.com/ProjectVault/orp/tree/master/hardware/mselSoC/src/systems/geophyte/rtl/verilog/crypto_aes/rtl/verilog`
  - `https://github.com/hgrosz/aes-dom`

∼20 years of optimized AES implementations

- exploiting different tower field constructions of $\mathbb{F}_{((2^2)^2)^2}$: $\mathbb{F}_{(2^4)^2}$, $\mathbb{F}_{2^8}$
- exhaustive search on defining polynomials, choice of bases, mixed bases
- subexpression eliminations, different arithmetic blocks, e.g., Itoh-Tsujii inversion
- optimizing for the given library, e.g., including `OAI` gates
- note: the list is incomplete!



- Alexis Bonnecaze, Pierre Liardet, and Alexandre Venelli, *AES side-channel countermeasure using random tower field constructions*, Designs, Codes and Cryptography, 2013.
- David Canright, *A Very Compact S-Box for AES*, CHES 2005.
- Panu Hamalainen, Timo Alho, Marko Hannikainen, and Timo Hamalainen, *Design and Implementation of Low-Area and Low-Power AES Encryption Hardware Core*, DSD'06, 2006.
- Nele Mentens, Lejla Batina, Bart Preneel, and Ingrid Verbauwhede, *A systematic evaluation of compact hardware implementations for the rijndael s-box*, Topics in Cryptology–CT-RSA 2005.
- Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang, *Pushing the limits: A very compact and a threshold implementation of AES*, EUROCRYPT 2011.
- Sanu Mathew, Sudhir Satpathy, Vikram Suresh, Mark Anders, Himanshu Kaul, Amit Agarwal, Steven Hsu, Gregory Chen, and Ram Krishnamurthy, *340 mV–1.1 V, 289 Gbps/W, 2090–Gate NanoAES Hardware Accelerator With Area-Optimized Encrypt/Decrypt* $\mathbb{F}(2^4)^2$ *Polynomials in 22 nm Tri-Gate CMOS*, IEEE Journal of Solid-State Circuits (2015).



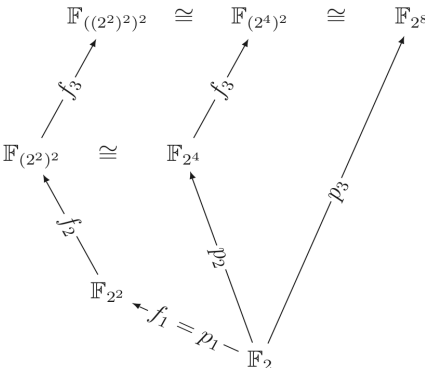
- Yasuyuki Nogami, Kenta Nekado, Tetsumi Toyota, Naoto Hongo, and Yoshitaka Morikawa, *Mixed Bases for Efficient Inversion in $\mathbb{F}((2^2)^2)^2$ and Conversion Matrices of SubBytes of AES*, CHES 2010.
- Atri Rudra, Pradeep K Dubey, Charanjit S Jutla, Vijay Kumar, Josyula R Rao, and Pankaj Rohatgi, *Efficient rijndael encryption implementation with composite field arithmetic*, CHES 2001.
- Vincent Rijmen, *Efficient implementation of the rijndael s-box*, Katholieke Universiteit Leuven, Dept. ESAT. Belgium (2000).
- Arash Reyhani-Masoleh, Mostafa Taha, and Doaa Ashmawy, *New Area Record for the AES Combined S-Box/Inverse S-Box*, ARITH 2018.
- Akashi Satoh, Sumio Morioka, Kohji Takano, and Seiji Munetoh, *A compact rijndael hardware architecture with s-box optimization*, ASIACRYPT 2001.
- Jenny Yu and Mark Aagaard, *Benchmarking and Optimizing AES for Lightweight Cryptography on ASICs*.

Tower Fields

- if m is composite, we can write $m = \prod_{i=1}^{k} m_i$, where $m_i \in \mathbb{Z}^+$, not necessary prime

$$\mathbb{F}_p \xrightarrow[f_1]{m_1} \mathbb{F}_{p^{m_1}} \xrightarrow[f_2]{m_2} \mathbb{F}_{(p^{m_1})^{m_2}} \xrightarrow[f_3]{m_3} \cdots \xrightarrow[f_k]{m_k} \mathbb{F}_{(\ldots((p^{m_1})^{m_2})\ldots)^{m_k}}$$

- for each extension $\mathbb{F}_{q^{m_i}}/\mathbb{F}_q$:
  - we need a defining polynomial f_i such that $m_i = \deg(f_i)$ degree of extension: $m_i = [\mathbb{F}_{q^{m_i}} : \mathbb{F}_q] = \deg(f_i)$
  - we need a basis for the representation of elements: let λ_i be root of f_i, then $\mathrm{PB} = \{1, \lambda_i, \ldots, \lambda_i^{m_i-1}\}$
- note that mixed bases are possible, e.g., polynomial basis on one level, normal basis on next
- note: diagram on the right is not showing all options for $m = 8$



Tower Fields

- Example: $\mathbb{F}_2 \xrightarrow[2]{f_1} F_{2^2} \xrightarrow[2]{f_2} \mathbb{F}_{(2^2)^2}$ with normal bases
- first extension: only one candidate $f_1(x) = x^2 + x + 1$, root λ

  $\lambda^2 = \lambda + 1$ NB$_1$=$\{\lambda, \lambda^2\}$

  $\lambda^3 = \lambda^2 + \lambda = 1$ $\mathbb{F}_2$ =$\{0, 1, \lambda, \lambda^2\}$
- second extension:

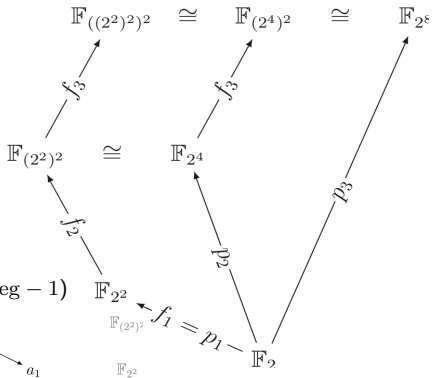
  we need a polynomial of degree 2, that is irreducible over $\mathbb{F}_{2^2}$

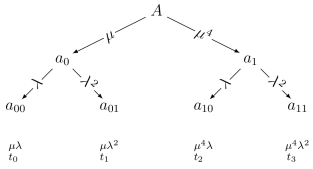
  6 candidates: $x^2 + \lambda x + 1$, $x^2 + \lambda x + \lambda$, $x^2 + x + \lambda$, $x^2 + x + \lambda^2$, $x^2 + \lambda^2 x + 1$, $x^2 + \lambda^2 x + \lambda^2$
- our choice: $f_2(x) = x^2 + x + \lambda$, root μ, NB$_2$=$\{\mu, \mu^4\}$

  (recall: NB elements are conjugates of the normal element β^{q^i} for $0, \ldots, \deg - 1$)
- how do we represent an element $A \in \mathbb{F}_{(2^2)^2}$

  $A = a_0\mu + a_1\mu^4$ where $a_i \in \mathbb{F}_{2^2}$

  $a_i = a_{i0}\lambda + a_{i1}\lambda^2$ where $a_i j \in \mathbb{F}_2$



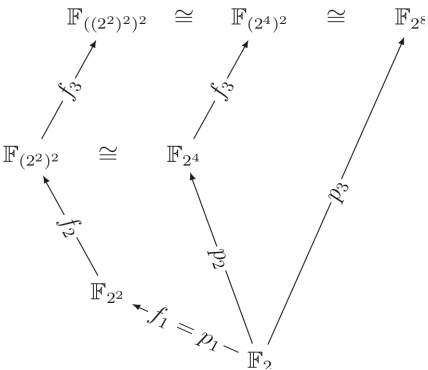


Tower Fields

- Example: $\mathbb{F}_2 \xrightarrow[2]{f_1} F_{2^2} \xrightarrow[2]{f_2} \mathbb{F}_{(2^2)^2}$ with normal bases
- three equations to work with:$\lambda^2 = \lambda + 1$, $\mu^2 = \mu + \lambda$ and $\mu + \mu^4 = 1$
- how to embed the subfields:

  $[0,0]_{\mathrm{NB}_1} = 0 = [0,0]_{\mathrm{NB}_2}$

  $[1,1]_{\mathrm{NB}_1} = 1 =$



Tower Fields

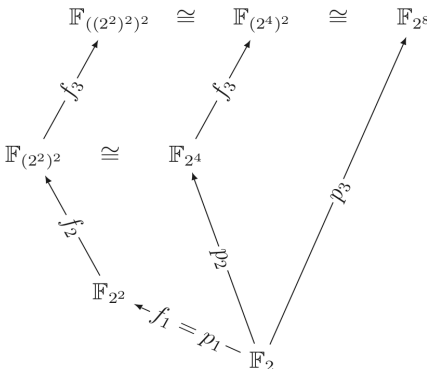
- Example: $\mathbb{F}_2 \xrightarrow[2]{f_1} F_{2^2} \xrightarrow[2]{f_2} \mathbb{F}_{(2^2)^2}$ with normal bases
- three equations to work with:$\lambda^2 = \lambda + 1$, $\mu^2 = \mu + \lambda$ and $\mu + \mu^4 = 1$
- how to embed the subfield:

  $[0,0]_{\mathrm{NB}_1} = 0 = [0,0]_{\mathrm{NB}_2}$

  $[1,1]_{\mathrm{NB}_1} = 1 = \mu + \mu^4 = [1,1]_{\mathrm{NB}_2}$

  $[1,0]_{\mathrm{NB}_1} = \lambda =$

  $[0,1]_{\mathrm{NB}_1} = \lambda^2 =$



Tower Fields

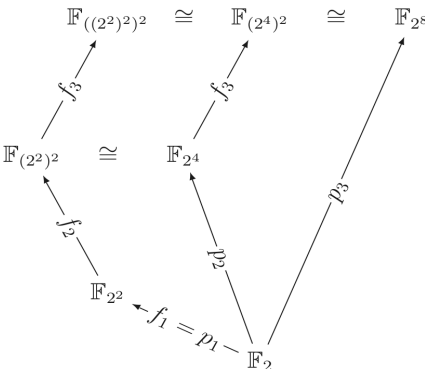
- Example: $\mathbb{F}_2 \xrightarrow[2]{f_1} F_{2^2} \xrightarrow[2]{f_2} \mathbb{F}_{(2^2)^2}$ with normal bases
- three equations to work with:$\lambda^2 = \lambda + 1$, $\mu^2 = \mu + \lambda$ and $\mu + \mu^4 = 1$

- how to embed the subfield:

$$[0,0]_{\mathrm{NB}_1} = 0 = [0,0]_{\mathrm{NB}_2}$$
$$[1,1]_{\mathrm{NB}_1} = 1 = \mu + \mu^4 = [1,1]_{\mathrm{NB}_2}$$
$$[1,0]_{\mathrm{NB}_1} = \lambda = \lambda \cdot 1 = \lambda(\mu + \mu^4) = \lambda\mu + \lambda\mu^4 = [\lambda, \lambda]_{\mathrm{NB}_2}$$
$$[0,1]_{\mathrm{NB}_1} = \lambda^2 = \lambda^2 \cdot 1 = \lambda^2(\mu + \mu^4) = \lambda^2\mu + \lambda^2\mu^4 = \left[\lambda^2, \lambda^2\right]_{\mathrm{NB}_2}$$

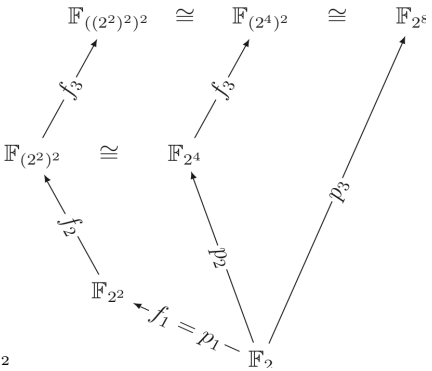
- how about other elements of $\mathbb{F}_{(2^2)^2}$?

$\mu^2 = \mu + \lambda =$



Tower Fields

- Example: $\mathbb{F}_2 \xrightarrow[2]{f_1} F_{2^2} \xrightarrow[2]{f_2} \mathbb{F}_{(2^2)^2}$ with normal bases
- three equations to work with:$\lambda^2 = \lambda + 1$, $\mu^2 = \mu + \lambda$ and $\mu + \mu^4 = 1$



- how to embed the subfield:

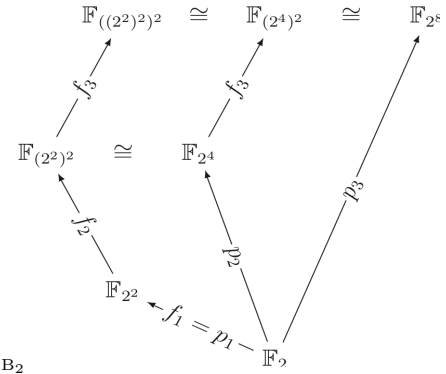
$$[0,0]_{\mathrm{NB}_1} = 0 = [0,0]_{\mathrm{NB}_2}$$
$$[1,1]_{\mathrm{NB}_1} = 1 = \mu + \mu^4 = [1,1]_{\mathrm{NB}_2}$$
$$[1,0]_{\mathrm{NB}_1} = \lambda = \lambda \cdot 1 = \lambda(\mu + \mu^4) = \lambda\mu + \lambda\mu^4 = [\lambda, \lambda]_{\mathrm{NB}_2}$$
$$[0,1]_{\mathrm{NB}_1} = \lambda^2 = \lambda^2 \cdot 1 = \lambda^2(\mu + \mu^4) = \lambda^2\mu + \lambda^2\mu^4 = \left[\lambda^2, \lambda^2\right]_{\mathrm{NB}_2}$$

- how about other elements of $\mathbb{F}_{(2^2)^2}$?

$$\mu^2 = \mu + \lambda = \mu + \lambda(\mu + \mu^4) = (1 + \lambda)\mu + \lambda\mu^4 = \lambda^2\mu + \lambda\mu^4 = \left[\lambda^2, \lambda\right]_{\mathrm{NB}_2}$$
$$\mu^3 = \mu^2 + \lambda\mu =$$

Tower Fields

- Example: $\mathbb{F}_2 \xrightarrow[2]{f_1} F_{2^2} \xrightarrow[2]{f_2} \mathbb{F}_{(2^2)^2}$ with normal bases
- three equations to work with:$\lambda^2 = \lambda + 1$, $\mu^2 = \mu + \lambda$ and $\mu + \mu^4 = 1$



- how to embed the subfield:

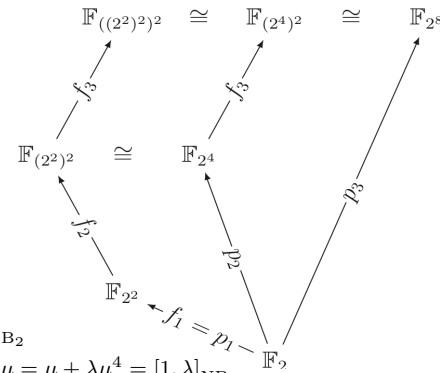
$$[0,0]_{\mathrm{NB}_1} = 0 = [0,0]_{\mathrm{NB}_2}$$
$$[1,1]_{\mathrm{NB}_1} = 1 = \mu + \mu^4 = [1,1]_{\mathrm{NB}_2}$$
$$[1,0]_{\mathrm{NB}_1} = \lambda = \lambda \cdot 1 = \lambda(\mu + \mu^4) = \lambda\mu + \lambda\mu^4 = [\lambda, \lambda]_{\mathrm{NB}_2}$$
$$[0,1]_{\mathrm{NB}_1} = \lambda^2 = \lambda^2 \cdot 1 = \lambda^2(\mu + \mu^4) = \lambda^2\mu + \lambda^2\mu^4 = \left[\lambda^2, \lambda^2\right]_{\mathrm{NB}_2}$$

- how about other elements of $\mathbb{F}_{(2^2)^2}$?

$$\mu^2 = \mu + \lambda = \mu + \lambda(\mu + \mu^4) = (1+\lambda)\mu + \lambda\mu^4 = \lambda^2\mu + \lambda\mu^4 = \left[\lambda^2, \lambda\right]_{\mathrm{NB}_2}$$
$$\mu^3 = \underline{\underline{\mu^2}} + \lambda\mu = \underline{\underline{\mu + \lambda}} + \lambda\mu = \mu + \lambda(\mu + \mu^4) + \lambda\mu = \mu + \underline{\lambda\mu} + \lambda\mu^4 + \underline{\lambda\mu} = \mu + \lambda\mu^4 = [1, \lambda]_{\mathrm{NB}_2}$$
$$\mu^4 = \mu^3 + \lambda\mu^2 =$$

Tower Fields

- Example: $\mathbb{F}_2 \xrightarrow[2]{f_1} F_{2^2} \xrightarrow[2]{f_2} \mathbb{F}_{(2^2)^2}$ with normal bases
- three equations to work with: $\lambda^2 = \lambda + 1$, $\mu^2 = \mu + \lambda$ and $\mu + \mu^4 = 1$



- how to embed the subfield:

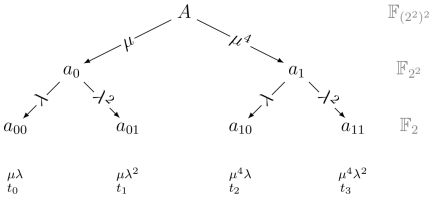
$$[0,0]_{\mathrm{NB}_1} = 0 = [0,0]_{\mathrm{NB}_2}$$
$$[1,1]_{\mathrm{NB}_1} = 1 = \mu + \mu^4 = [1,1]_{\mathrm{NB}_2}$$
$$[1,0]_{\mathrm{NB}_1} = \lambda = \lambda \cdot 1 = \lambda(\mu + \mu^4) = \lambda\mu + \lambda\mu^4 = [\lambda, \lambda]_{\mathrm{NB}_2}$$
$$[0,1]_{\mathrm{NB}_1} = \lambda^2 = \lambda^2 \cdot 1 = \lambda^2(\mu + \mu^4) = \lambda^2\mu + \lambda^2\mu^4 = \left[\lambda^2, \lambda^2\right]_{\mathrm{NB}_2}$$

- how about other elements of $\mathbb{F}_{(2^2)^2}$?

$$\mu^2 = \mu + \lambda = \mu + \lambda(\mu + \mu^4) = (1+\lambda)\mu + \lambda\mu^4 = \lambda^2\mu + \lambda\mu^4 = \left[\lambda^2, \lambda\right]_{\mathrm{NB}_2}$$
$$\mu^3 = \underline{\underline{\mu^2}} + \lambda\mu = \underline{\underline{\mu + \lambda}} + \lambda\mu = \mu + \lambda(\mu + \mu^4) + \lambda\mu = \mu + \underline{\lambda\mu} + \lambda\mu^4 + \underline{\lambda\mu} = \mu + \lambda\mu^4 = [1,\lambda]_{\mathrm{NB}_2}$$
$$\mu^4 = \underline{\underline{\mu^3}} + \lambda\underline{\mu^2} = \underline{\underline{\mu + \lambda\mu^4}} + \lambda(\underline{\lambda + \mu}) = \mu + \lambda\mu^4 + \lambda^2 + \lambda\mu = \mu + \lambda\mu^4 + \lambda^2(\mu + \mu^4) + \lambda\mu =$$
$$= \mu + \lambda\mu + \lambda^2\mu + \lambda^2\mu^4 + \lambda\mu^4 = (1 + \lambda + \lambda^2)\mu + (\lambda^2 + \lambda)\mu^4 = 0 \cdot \mu + 1 \cdot \mu^4 = [0,1]_{\mathrm{NB}_2}$$

Tower Fields

- Example: $\mathbb{F}_2 \xrightarrow[2]{f_1} F_{2^2} \xrightarrow[2]{f_2} \mathbb{F}_{(2^2)^2}$ with normal bases
- first extension: $f_1(x) = x^2 + x + 1$, root λ, $\mathrm{NB}_1 = \{\lambda, \lambda^2\}$
- second extension: $f_2(x) = x^2 + x + \lambda$, root μ, $\mathrm{NB}_2 = \{\mu, \mu^4\}$



- how do we represent an element $A \in \mathbb{F}_{(2^2)^2}$

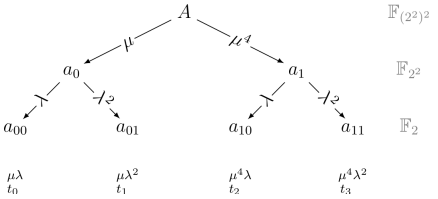
  $A = a_0\mu + a_1\mu^4$ where $a_i \in \mathbb{F}_{2^2}$

  $a_i = a_{i0}\lambda + a_{i1}\lambda^2$ where $a_i j \in \mathbb{F}_2$

- how about elements $t_0 = \mu\lambda$, $t_1 = \mu\lambda^2$, $t_2 = \mu^4\lambda$, $t_3 = \mu^4\lambda^2$?

Tower Fields

- Example: $\mathbb{F}_2 \xrightarrow[2]{f_1} F_{2^2} \xrightarrow[2]{f_2} \mathbb{F}_{(2^2)^2}$ with normal bases
- first extension: $f_1(x) = x^2 + x + 1$, root λ, $\mathrm{NB}_1 = \{\lambda, \lambda^2\}$
- second extension: $f_2(x) = x^2 + x + \lambda$, root μ, $\mathrm{NB}_2 = \{\mu, \mu^4\}$



- how do we represent an element $A \in \mathbb{F}_{(2^2)^2}$

  $A = a_0\mu + a_1\mu^4$ where $a_i \in \mathbb{F}_{2^2}$

  $a_i = a_{i0}\lambda + a_{i1}\lambda^2$ where $a_i j \in \mathbb{F}_2$

- how about elements $t_0 = \mu\lambda$, $t_1 = \mu\lambda^2$,

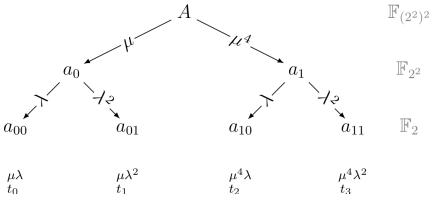
  $t_2 = \mu^4\lambda$, $t_3 = \mu^4\lambda^2$?

  are they linearly independent ? find A, B, C, D such that $At_0 + Bt_1 + Ct_2 + Dt_3 = 0$

  $A(\mu\lambda) + B(\mu\lambda^2) + C(\mu^4\lambda) + D(\mu^4\lambda^2) = 0$

Tower Fields

- Example: $\mathbb{F}_2 \xrightarrow[2]{f_1} F_{2^2} \xrightarrow[2]{f_2} \mathbb{F}_{(2^2)^2}$ with normal bases
- first extension: $f_1(x) = x^2 + x + 1$, root λ, $\mathrm{NB}_1 = \{\lambda, \lambda^2\}$
- second extension: $f_2(x) = x^2 + x + \lambda$, root μ, $\mathrm{NB}_2 = \{\mu, \mu^4\}$



- how do we represent an element $A \in \mathbb{F}_{(2^2)^2}$

  $A = a_0\mu + a_1\mu^4$ where $a_i \in \mathbb{F}_{2^2}$

  $a_i = a_{i0}\lambda + a_{i1}\lambda^2$ where $a_i j \in \mathbb{F}_2$

- how about elements $t_0 = \mu\lambda$, $t_1 = \mu\lambda^2$,

  $t_2 = \mu^4\lambda$, $t_3 = \mu^4\lambda^2$?

  are they linearly independent ? find A, B, C, D such that $At_0 + Bt_1 + Ct_2 + Dt_3 = 0$

  $A(\mu\lambda) + B(\mu\lambda^2) + C(\mu^4\lambda) + D(\mu^4\lambda^2) = 0$

  $\mu(A\lambda + B\lambda^2) + \mu^4(C\lambda + D\lambda^2) = 0$

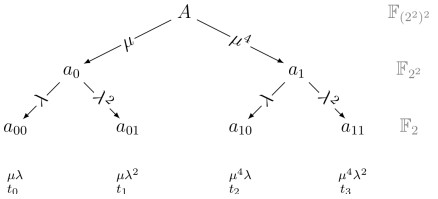
Tower Fields

- Example: $\mathbb{F}_2 \xrightarrow[2]{f_1} F_{2^2} \xrightarrow[2]{f_2} \mathbb{F}_{(2^2)^2}$ with normal bases
- first extension: $f_1(x) = x^2 + x + 1$, root λ, NB$_1$=$\{\lambda, \lambda^2\}$
- second extension: $f_2(x) = x^2 + x + \lambda$, root μ, NB$_2$=$\{\mu, \mu^4\}$

- how do we represent an element $A \in \mathbb{F}_{(2^2)^2}$

  $A = a_0\mu + a_1\mu^4$ where $a_i \in \mathbb{F}_{2^2}$

  $a_i = a_{i0}\lambda + a_{i1}\lambda^2$ where $a_i j \in \mathbb{F}_2$



- how about elements $t_0 = \mu\lambda$, $t_1 = \mu\lambda^2$,

  $t_2 = \mu^4\lambda$, $t_3 = \mu^4\lambda^2$?

  are they linearly independent ? find A, B, C, D such that $At_0 + Bt_1 + Ct_2 + Dt_3 = 0$

  $A(\mu\lambda) + B(\mu\lambda^2) + C(\mu^4\lambda) + D(\mu^4\lambda^2) = 0$

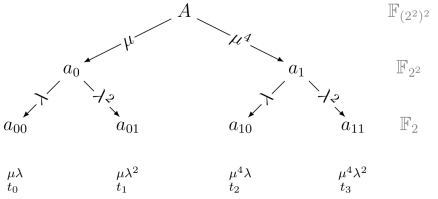
  $\mu(A\lambda + B\lambda^2) + \mu^4(C\lambda + D\lambda^2) = 0 \rightarrow$ NB$_2$ $\rightarrow$ μ and μ^4 are lin.indep. $\rightarrow$ brackets=0 $\Rightarrow$

  $A\lambda + B\lambda^2 = 0$

  $C\lambda + D\lambda^2 = 0$

Tower Fields

- Example: $\mathbb{F}_2 \xrightarrow[2]{f_1} F_{2^2} \xrightarrow[2]{f_2} \mathbb{F}_{(2^2)^2}$ with normal bases
- first extension: $f_1(x) = x^2 + x + 1$, root λ, NB$_1$=$\{\lambda, \lambda^2\}$
- second extension: $f_2(x) = x^2 + x + \lambda$, root μ, NB$_2$=$\{\mu, \mu^4\}$



- how do we represent an element $A \in \mathbb{F}_{(2^2)^2}$

  $A = a_0\mu + a_1\mu^4$ where $a_i \in \mathbb{F}_{2^2}$

  $a_i = a_{i0}\lambda + a_{i1}\lambda^2$ where $a_i j \in \mathbb{F}_2$

- how about elements $t_0 = \mu\lambda$, $t_1 = \mu\lambda^2$,

  $t_2 = \mu^4\lambda$, $t_3 = \mu^4\lambda^2$?

  are they linearly independent ? find A, B, C, D such that $At_0 + Bt_1 + Ct_2 + Dt_3 = 0$

  $A(\mu\lambda) + B(\mu\lambda^2) + C(\mu^4\lambda) + D(\mu^4\lambda^2) = 0$

  $\mu(A\lambda + B\lambda^2) + \mu^4(C\lambda + D\lambda^2) = 0 \rightarrow$ NB$_2$ $\rightarrow \mu$ and μ^4 are lin.indep. $\rightarrow$ brackets=0 $\Rightarrow$

  $A\lambda + B\lambda^2 = 0 \rightarrow$ NB$_1$ $\rightarrow \lambda$ and λ^2 are lin.indep. $\rightarrow A = B = 0$

  $C\lambda + D\lambda^2 = 0 \rightarrow$ NB$_1$ $\rightarrow \lambda$ and λ^2 are lin.indep. $\rightarrow C = D = 0$

- TFB=$\{t_0, t_1, t_2, t_3\}$ also form a basis! we need it for TESTVECTORS

Tower Fields

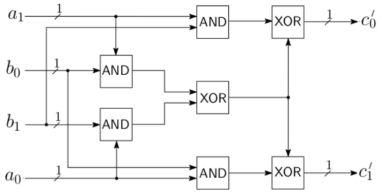
- Putting it all together: knowing that μ is a generator of $\mathbb{F}_{(2^2)^2}$, complete the elements table

TFB				NB_2			
t_0	t_1	t_2	t_3	μ	μ^4	polynomial	μ^i
0	0	0	0	0	0	0	–
0	0	0	1				
0	0	1	0				
0	0	1	1	0	1	μ^4	μ^4
0	1	0	0				
0	1	0	1	λ^2	λ^2	$\lambda^2\mu + \lambda^2\mu^4$	
0	1	1	0	λ^2	λ	$\lambda^2\mu + \lambda\mu^4$	μ^2
0	1	1	1				
1	0	0	0				
1	0	0	1				
1	0	1	0	λ	λ	$\lambda\mu + \lambda\mu^4$	
1	0	1	1				
1	1	0	0	1	0	μ	μ
1	1	0	1				
1	1	1	0	1	λ	$\mu + \lambda\mu^4$	μ^3
1	1	1	1	1	1	$\mu + \mu^4$	1

Tower Field Multiplication

- Example: $\mathbb{F}_2 \xrightarrow[2]{f_1} F_{2^2} \xrightarrow[2]{f_2} \mathbb{F}_{(2^2)^2}$ with normal bases
- first extension: $f_1(x) = x^2 + x + 1$, root λ, $\mathsf{NB}_1 = \{\lambda, \lambda^2\}$
- second extension: $f_2(x) = x^2 + x + \lambda$, root μ, $\mathsf{NB}_2 = \{\mu, \mu^4\}$
- Multiplication in $\mathbb{F}_{2^2}$:

$$\begin{aligned} A \cdot B &= (a_0\lambda + a_1\lambda^2)(b_0\lambda + b_1\lambda^2) \\ &= a_0b_0\lambda^2 + a_1b_1\lambda^3 + a_1b_0\lambda^3 + a_1b_1\lambda^4 \\ &= \dots \\ &= (a_1b_1 + \underline{a_0b_1 + a_1b_0})\lambda + (\underline{a_0b_1 + a_1b_0} + a_0b_0)\lambda^2 \end{aligned}$$



- subexpression!
- multiplier with 4 `AND` gates and 3 `XOR` gates

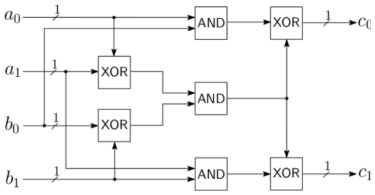
Tower Field Multiplication

- Example: $\mathbb{F}_2 \xrightarrow[2]{f_1} F_{2^2} \xrightarrow[2]{f_2} \mathbb{F}_{(2^2)^2}$ with normal bases
- first extension: $f_1(x) = x^2 + x + 1$, root λ, $\mathsf{NB}_1 = \{\lambda, \lambda^2\}$
- second extension: $f_2(x) = x^2 + x + \lambda$, root μ, $\mathsf{NB}_2 = \{\mu, \mu^4\}$
- Multiplication in $\mathbb{F}_{2^2}$:

$$
\begin{aligned}
A \cdot B &= (a_0\lambda + a_1\lambda^2)(b_0\lambda + b_1\lambda^2) \\
&= \dots \\
&= (a_1b_1 + a_0b_1 + a_1b_0)\lambda + (a_0b_1 + a_1b_0 + a_0b_0)\lambda^2
\end{aligned}
$$

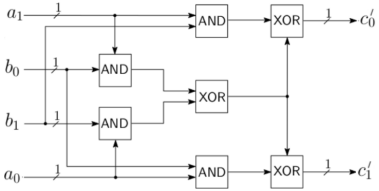
TRICK : convolution + remove missing term

$$
= (\underline{(a_0 + a_1)(b_0 + b_1)} + a_0b_0)\lambda + (\underline{(a_0 + a_1)(b_0 + b_1)} + a_1b_1)\lambda^2
$$

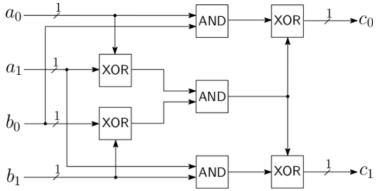


- subexpression!
- multiplier with 3 `AND` gates and 4 `XOR` gates

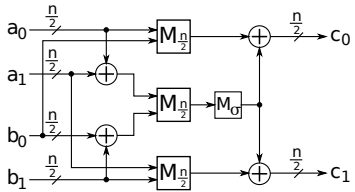
Tower Field Multiplication



4 AND gates, 3 XOR gates



3 AND gates, 4 XOR gates



- why would that be better ?
- when we move up the tower, AND gates become multipliers for the lower level!

- Exercise: continue the example and derive expression for the second level, that is for $A, B \in \mathbb{F}_{(2^2)^2}$

  $\mathbb{F}_2 \xrightarrow[2]{f_1} F_{2^2} \xrightarrow[2]{f_2} \mathbb{F}_{(2^2)^2}$ with normal bases

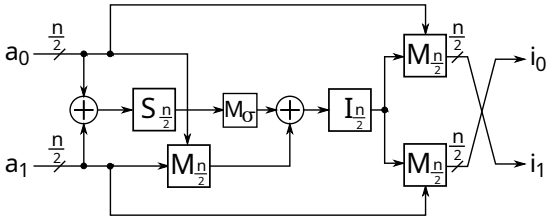
  second extension: $f_2(x) = x^2 + x + \lambda$, root μ, $\mathsf{NB}_2 = \{\mu, \mu^4\}$

Tower Field Inversion: Itoh-Tsujii Algorithm

- Example: $\mathbb{F}_2 \xrightarrow[2]{f_1} F_{2^2} \xrightarrow[2]{f_2} \mathbb{F}_{(2^2)^2}$ with normal bases NB$_1$=$\{\lambda, \lambda^2\}$, NB$_2$=$\{\mu, \mu^4\}$, $A = a_0\mu + a_1\mu^4$, $a_i = a_{i0}\lambda + a_{i1}\lambda^2$
- $q = 2^2 = 4 \rightarrow r = 5 \Rightarrow$ we need A^4 and A^5
- using normal bases: $A^4 = A^q \rightarrow$ rotate $\rightarrow A^4 = a_1\mu + a_0\mu^4$
- $A^5 = A^4 \cdot A = \cdots = \left[a_0a_1 + (a_0 + a_1)^2\lambda\right]\mu + \left[a_0a_1 + (a_0 + a_1)^2\lambda\right]\mu^4 = \left[a_0a_1 + (a_0 + a_1)^2\lambda\right]\cancel{(\mu + \mu^4)}$

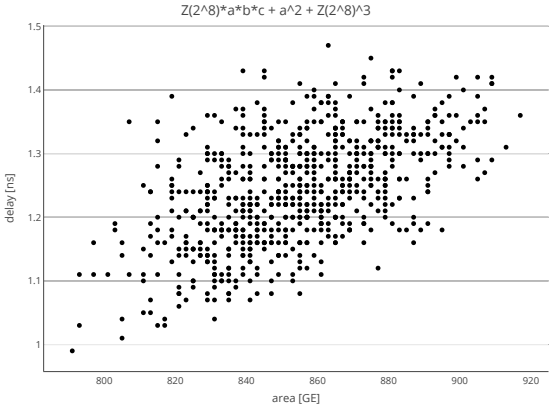
  $a_i \in \mathbb{F}_{2^2} \Rightarrow$ only subfield operations: M$_2$, SQ$_2$, M$_\lambda$
- put it back together: $A^{-1} = (A^5)^{-1} \cdot A^4 = (A^5)^{-1}(a_1\mu + a_0\mu^4) = a_1(A^5)^{-1}\mu + a_0(A^5)^{-1}\mu^4$

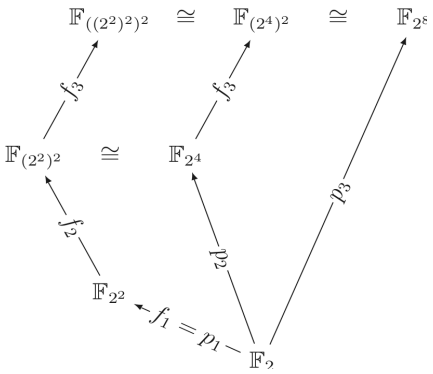
  $a_i, A^5 \in \mathbb{F}_{2^2} \Rightarrow$ only subfield operations: M$_2$, I$_2$, M$_\lambda$



Design Space Exploration

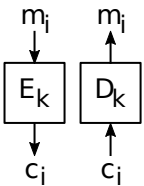
- # poly of degree 8 , irred over $\mathbb{F}_2$: 30
- # combinations for $\mathbb{F}_2 \xrightarrow[2]{f_1} F_{2^2} \xrightarrow[2]{f_2} \mathbb{F}_{(2^2)^2} \xrightarrow[2]{f_3} \mathbb{F}_{((2^2)^2)^2}$: 720
- use mathematical properties to reduce design space!





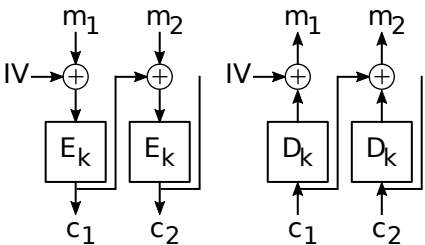
Block Ciphers - Modes of operation

ECB
Electronic
Code Book



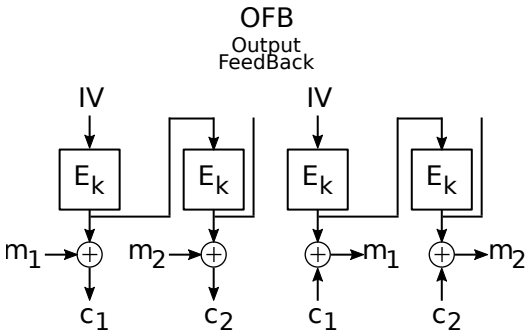
NO MAC !

CBC
Cipher Block
Chaining

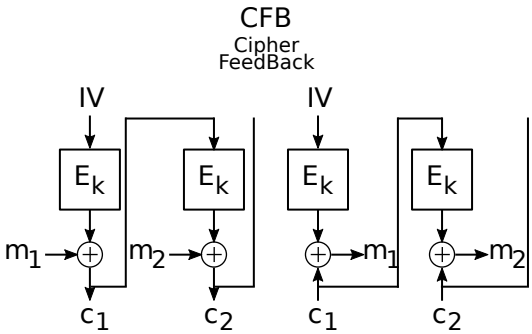


MAC: last block

Block Ciphers - Modes of operation

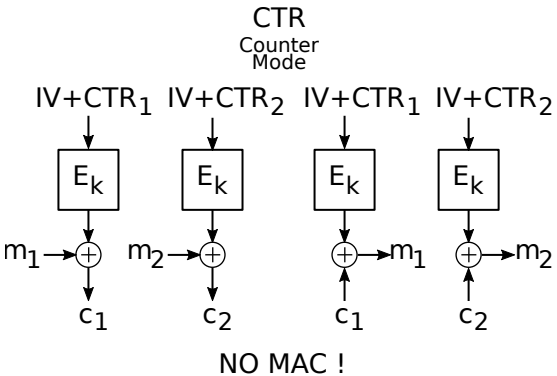


NO MAC !



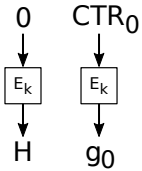
MAC: last block

Block Ciphers - Modes of operation



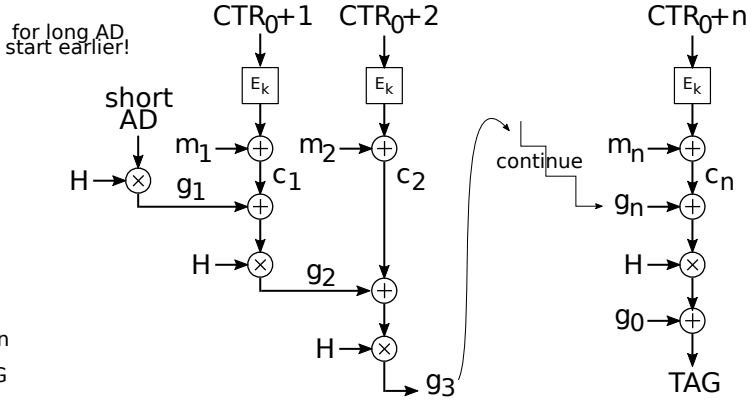
Block Ciphers - Modes of operation

AES-GCM
CTR for enc.
GHASH for MAC



⊗ is GF(2^128) multiplication

output: $c_1 \| c_2 \| \ldots \| c_n \| \text{TAG}$



AES-GCM is used to provide AEAD functionality!

Towards AEAD

- Recall: need keyed hash to provide data integrity and source integrity (authentication)
- ENC and MAC: c = Enc_{k1}(m), TAG = MAC_{k2}(m) — example: SSH
- ENC then MAC: c = Enc_{k1}(m), TAG = MAC_{k2}(c) — example: IPsec
- MAC then ENC: TAG = MAC_{k2}(c), c = Enc_{k1}(m $\|$ TAG), — example: TLS

- Modes of operation examples: CBC, CFB, AES-GCM

`https://www.nist.gov/news-events/news/2023/02/nist-selects-lightweight-cryptography-algorithms-protect-small-devices`

Towards AEAD

- Recall: need keyed hash to provide data integrity and source integrity (authentication)
- ENC and MAC: $c = Enc_{k1}(m)$, $TAG = MAC_{k2}(m)$ — example: SSH
- ENC then MAC: $c = Enc_{k1}(m)$, $TAG = MAC_{k2}(c)$ — example: IPsec
- MAC then ENC: $TAG = MAC_{k2}(c)$, $c = Enc_{k1}(m \parallel TAG)$, — example: TLS

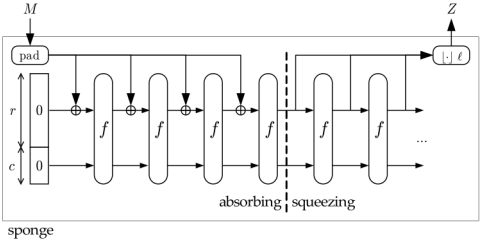
- Modes of operation examples: CBC, CFB, AES-GCM

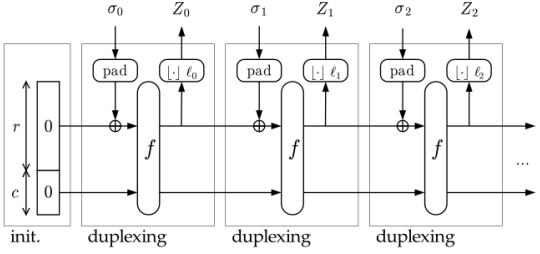
- Dedicated AEAD schemes:
  - CAESAR competition (2013-2019) **Ascon**
    Competition for Authenticated Encryption: Security, Applicability, and Robustness
    `https://competitions.cr.yp.to/caesar.html`
  - NIST LWC (Lightweight Cryptography) standardization process (2018-2023) **Ascon**
    authenticated encryption with associated data (AEAD) functionality, and optionally hashing functionality
    `https://csrc.nist.gov/projects/lightweight-cryptography`
    `https://www.nist.gov/news-events/news/2023/02/nist-selects-lightweight-cryptography-algorithms-protect-small-devices`

Sponge Constructions

- 2007 Keccak team introduced Sponge Functions, more details:
  `https://keccak.team/files/KeccakDIAC2012.pdf` or `https://keccak.team/files/CSF-0.1.pdf`

- NIST SHA-3 Competition (2007-2012) **Keccak**
  - Keccak[r,c] with Keccak-f permutation
    `https://csrc.nist.rip/external/nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf`

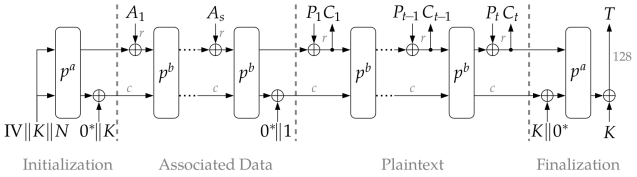




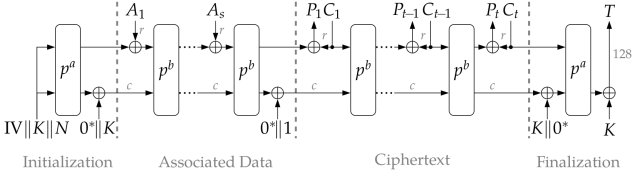
- many NIST LWC submissions used Sponge+permutation design strategy

Ascon

- recommended parameters: Ascon-128 a=12, b=6; Ascon-128 a=12, b=8
- details: `https://ascon.iaik.tugraz.at/index.html` and `https://ascon.iaik.tugraz.at/implementations.html`



(a) Encryption $\mathcal{E}_{k,r,a,b}$



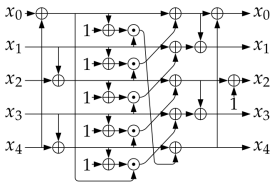
(b) Decryption $\mathcal{D}_{k,r,a,b}$

Ascon

- simple permutation with 3 steps:
  add round constants
  substitution layer
  linear diffusion layer
- some S-box design criteria (SPN):
  each output bit depends on at least 4 input bits
  algebraic degree 2, no fix-points
  efficient bit-sliced implementation

Table 5: Ascon's 5-bit S-box $\mathcal{S}$ as a lookup table.

x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	10	11	12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	1f
$\mathcal{S}(x)$	4	b	1f	14	1a	15	9	2	1b	5	8	12	1d	3	6	1c	1e	13	7	e	0	d	11	18	10	c	1	19	16	a	f	17



(a) Ascon's 5-bit S-box $\mathcal{S}(x)$

$$x_0 \leftarrow \Sigma_0(x_0) = x_0 \oplus (x_0 \ggg 19) \oplus (x_0 \ggg 28)$$
$$x_1 \leftarrow \Sigma_1(x_1) = x_1 \oplus (x_1 \ggg 61) \oplus (x_1 \ggg 39)$$
$$x_2 \leftarrow \Sigma_2(x_2) = x_2 \oplus (x_2 \ggg 1) \oplus (x_2 \ggg 6)$$
$$x_3 \leftarrow \Sigma_3(x_3) = x_3 \oplus (x_3 \ggg 10) \oplus (x_3 \ggg 17)$$
$$x_4 \leftarrow \Sigma_4(x_4) = x_4 \oplus (x_4 \ggg 7) \oplus (x_4 \ggg 41)$$

(b) Ascon's linear layer with 64-bit functions $\Sigma_i(x_i)$

Figure 4: Ascon's substitution layer and linear diffusion layer.

Public Key Cryptography

Reminder

Key generation:

- Choose two different large primes (p and q) and compute n = p*q
- Compute λ(n)
  = lcm(p-1,q-1)
  = (p-1)*(q-1)/gcd(p-1,q-1)
  (lcm = least common multiple)
  (gcd = greatest common divider)
- Choose e < λ(n) and relatively prime to λ(n)
- Calculate d = e^{-1} mod λ(n)

Public key = (e,n)
Private key = (p,q) or d

Public-key encryption/decryption:
Encryption: c = m^e mod n
Decryption: m = c^d mod n

Digital signature generation/verification:
Signature generation: s = $[H(m)]^d$ mod n, then send m || s
Signature verification: H(m) = s^e mod n?

Efficient implementation of RSA

Key generation:

- Choose two different large primes (p and q) and compute n = p*q
- Compute λ(n)
  = lcm(p-1,q-1)
  = (p-1)*(q-1)/gcd(p-1,q-1)
  (lcm = least common multiple)
  (gcd = greatest common divider)
- Choose e < λ(n) and relatively prime to λ(n)
- Calculate d = e^{-1} mod λ(n)

Public key = (e,n)
Private key = (p,q) or d

Key generation only needs to be done once, and can be done offline

Modular exponentiation is needed in each online step, and is a time- and energy-consuming operation

Public-key encryption/decryption:	*Digital signature generation/verification:*
Encryption: c = m^e mod n	Signature generation: s = $[H(m)]^d$ mod n, then send m || s
Decryption: m = c^d mod n	Signature verification: H(m) = s^e mod n?

Efficient modular exponentiation

Naive algorithm:

G = finite group → all operations are modular (mod n)

Algorithm Right-to-left binary exponentiation

INPUT: an element $g \in G$ and integer $e \geq 1$.
OUTPUT: g^e.

1. $A \leftarrow 1$, $S \leftarrow g$.
2. While $e \neq 0$ do the following:
   - 2.1 If e is odd then $A \leftarrow A \cdot S$. → modular multiplication
   - 2.2 $e \leftarrow \lfloor e/2 \rfloor$.
   - 2.3 If $e \neq 0$ then $S \leftarrow S \cdot S$. → modular squaring: more efficient than modular multiplication
3. Return(A).

Menezes et al., *Handbook of Applied Cryptography*, https://cacr.uwaterloo.ca/hac/

Efficient modular exponentiation

First optimization: trading off memory for computational speed

Algorithm Left-to-right k-ary exponentiation

INPUT: g and $e = (e_t e_{t-1} \cdots e_1 e_0)_b$, where $b = 2^k$ for some $k \geq 1$.
OUTPUT: g^e.

1. *Precomputation.*
   1.1 $g_0 \leftarrow 1$.
   1.2 For i from 1 to $(2^k - 1)$ do: $g_i \leftarrow g_{i-1} \cdot g$. (Thus, $g_i = g^i$.)
2. $A \leftarrow 1$.
3. For i from t down to 0 do the following:
   3.1 $A \leftarrow A^{2^k}$.
   3.2 $A \leftarrow A \cdot g_{e_i}$.
4. Return(A).

Multiple consecutive modular squarings in software or dedicated circuit to perform a modular exponentiation with fixed exponent 2^k in hardware

Exercises modular exponentiation

1) Assume: g = 2, e = 303, n = 5

   Compute g^e mod n with the right-to-left binary method, the left-to-right binary method, and the left-to-right 2-ary method

2) Assume: $e = 2^{2048} - 3$, $n = 2^{2203} - 1$

   How many modular multiplications and modular squarings are needed for each of the three methods?

Solution exercise 1: right-to-left with g = 2, e = 303, n = 5

A ← 1, S ← 2
e = 303:
    e is odd: A ← A.S = 1.2 = 2
    e ← 151
    e ≠ 0: S ← S.S = 2.2 = 4
e = 151:
    e is odd: A ← A.S = 2.4 = 3
    e ← 75
    e ≠ 0: S ← S.S = 4.4 = 1
e = 75:
    e is odd: A ← A.S = 3.1 = 3
    e ← 37
    e ≠ 0: S ← S.S = 1.1 = 1
e = 37:
    e is odd: A ← A.S = 3.1 = 3
    e ← 18
    e ≠ 0: S ← S.S = 1.1 = 1
e = 18:
    e ← 9
    e ≠ 0: S ← S.S = 1.1 = 1
e = 9:
    e is odd: A ← A.S = 3.1 = 3
    e ← 4
    e ≠ 0: S ← S.S = 1.1 = 1
e = 4:
    e ← 2
    e ≠ 0: S ← S.S = 1.1 = 1
e = 2:
    e ← 1
    e ≠ 0: S ← S.S = 1.1 = 1
e = 1:
    e is odd: A ← A.S = 3.1 = 3
    e ← 0
Return (3)

Algorithm Right-to-left binary exponentiation

INPUT: an element $g \in G$ and integer $e \geq 1$.
OUTPUT: g^e.

1. $A \leftarrow 1$, $S \leftarrow g$.
2. While $e \neq 0$ do the following:
   2.1 If e is odd then $A \leftarrow A \cdot S$.
   2.2 $e \leftarrow \lfloor e/2 \rfloor$.
   2.3 If $e \neq 0$ then $S \leftarrow S \cdot S$.
3. Return(A).

Solution exercise 1: left-to-right with g = 2, e = 303, n = 5

$e = (303)_{10} = (100101111)_2 = (e_8e_7e_6e_5e_4e_3e_2e_1e_0)_2$

A ← 1

i = 8:

A ← A.A = 1.1 = 1

e_8 = 1: A ← A.g = 1.2 = 2

i = 7:

A ← A.A = 2.2 = 4

i = 6:

A ← A.A = 4.4 = 1

i = 5:

A ← A.A = 1.1 = 1

e_5 = 1: A ← A.g = 1.2 = 2

i = 4:

A ← A.A = 2.2 = 4

i = 3:

A ← A.A = 4.4 = 1

e_3 = 1: A ← A.g = 1.2 = 2

i = 2:

A ← A.A = 2.2 = 4

e_2 = 1: A ← A.g = 4.2 = 3

i = 1:

A ← A.A = 3.3 = 4

e_1 = 1: A ← A.g = 4.2 = 3

i = 0:

A ← A.A = 3.3 = 4

e_0 = 1: A ← A.g = 4.2 = 3

Return (3)

Algorithm Left-to-right binary exponentiation

INPUT: $g \in G$ and a positive integer $e = (e_t e_{t-1} \cdots e_1 e_0)_2$.
OUTPUT: g^e.

1. $A \leftarrow 1$.
2. For i from t down to 0 do the following:
   2.1 $A \leftarrow A \cdot A$.
   2.2 If $e_i = 1$, then $A \leftarrow A \cdot g$.
3. Return(A).

Solution exercise 1: k-ary with k = 2, g = 2, e = 303, n = 5

$e = (303)_{10} = (100101111)_2 = (10233)_4 = (e_4e_3e_2e_1e_0)_4$

Precomputation

$g_0 \leftarrow 1$

i = 1: $g_1 \leftarrow g_0.g = 1.2 = 2$

i = 2: $g_2 \leftarrow g_1.g = 2.2 = 4$

i = 3: $g_3 \leftarrow g_2.g = 4.2 = 3$

$A \leftarrow 1$

i = 4:

$A \leftarrow A^4 = 1^4 = 1$

$A \leftarrow A.g_1 = 1.2 = 2$

i = 3:

$A \leftarrow A^4 = 2^4 = 1$

$A \leftarrow A.g_0 = 1.1 = 1$

i = 2:

$A \leftarrow A^4 = 1^4 = 1$

$A \leftarrow A.g_2 = 1.4 = 4$

i = 1:

$A \leftarrow A^4 = 4^4 = 1$

$A \leftarrow A.g_3 = 1.3 = 3$

i = 0:

$A \leftarrow A^4 = 3^4 = 1$

$A \leftarrow A.g_3 = 1.3 = 3$

Return (3)

Algorithm Left-to-right k-ary exponentiation

INPUT: g and $e = (e_t e_{t-1} \cdots e_1 e_0)_b$, where $b = 2^k$ for some $k \geq 1$.
OUTPUT: g^e.

1. *Precomputation.*
   1.1 $g_0 \leftarrow 1$.
   1.2 For i from 1 to $(2^k - 1)$ do: $g_i \leftarrow g_{i-1} \cdot g$. (Thus, $g_i = g^i$.)
2. $A \leftarrow 1$.
3. For i from t down to 0 do the following:
   3.1 $A \leftarrow A^{2^k}$.
   3.2 $A \leftarrow A \cdot g_{e_i}$.
4. Return(A).

Solution exercise 2: $e = 2^{2048} - 3$, $n = 2^{2203} - 1$

Right-to-left binary exponentiation and right-to-left binary exponentiation

$e = 2^{2048} - 3 = (e_{2047}e_{2046} \ldots e_1e_0)_2$

2048 modular squarings

$2^{2048} - 1 = (1111 \ldots 1111)_2$ (all ones)
$2^{2048} - 3 = (1111 \ldots 1101)_2$ (all ones but e1) → 2047 ones → **2047 modular multiplications**

2-ary modular exponentiation

$e = 2^{2048} - 3 = (e_{1023}e_{1022} \ldots e_1e_0)_4$

Precomputation: 3 modular multiplications

Computation: 1024 times $A \leftarrow A^{2^k}$. = 1024 * 2 modular squarings

$A \leftarrow A \cdot g_{e_i}$. = 1024 modular multiplications

Total: **2048 modular squarings** and **1027 modular multiplications**

Modular exponentiation: further optimization

The Chinese remainder theorem (CRT) can be used for the modular exponentiation in RSA decryption and signature generation. It reduces the computation to modular exponentiations with exponents of half the size, taking into account that n = p*q.

Precomputation:

1. $d_p = d \mod (p-1)$;
2. $d_q = d \mod (q-1)$;
3. $q_{inv} = (1/q) \mod p$, with $p > q$.

Algorithm RSA decryption using the Chinese Remainder Theorem

Require: p, q, d_p, d_q, q_{inv}, c,
Ensure: c^d

1: $m1 \leftarrow c^{d_p} \mod p$
2: $m2 \leftarrow c^{d_q} \mod q$
3: $h \leftarrow (q_{inv} \cdot (m1 - m2)) \mod p$
4: $m \leftarrow m2 + h \cdot q$
5: Return m

Question: Why can't we use CRT for RSA encryption and signature verification?

54

More on Exponentiation

- Exponentiation Algorithms: B^E, where $B \in G$, B=base, E=exponent

- General case: both unknown (we have already seen examples in RSA slides)
- Fixed exponent: optimizations with addition chains (example: RSA until re-keyed)
- Fixed base: optimizations with NAF and τ-NAF representation of exponent (example: Diffie-Hellman)

- recall: Square and Multiply **Always** Algorithm (lecture 2)
  always be aware of what the base and the exponent represent
  if sensitive information (e.g., key), must modify the algorithm to prevent leakage

Modular Multiplication

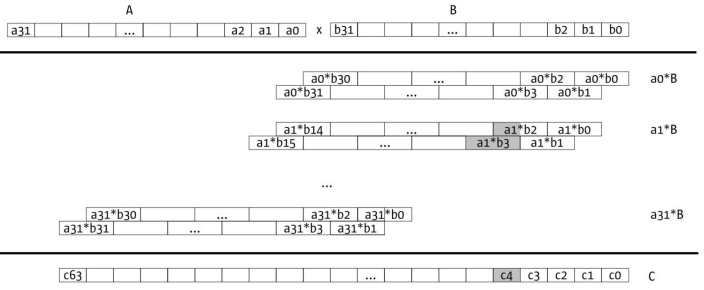
Naive algorithm:

Algorithm Classical modular multiplication

INPUT: two positive integers x, y and a modulus m, all in radix b representation.
OUTPUT: $x \cdot y \bmod m$.

1. Compute $x \cdot y$ → integer multiplication
2. Compute the remainder r when $x \cdot y$ is divided by m → integer division (very compute-intensive)
3. Return(r).

Multi-precision integer multiplication:



Modular Multiplication

Montgomery multiplication:
(no integer division needed, but x*y/R is calculated instead of x*y, where R is a power of 2):

Algorithm Montgomery multiplication

INPUT: integers $m = (m_{n-1} \cdots m_1 m_0)_b$, $x = (x_{n-1} \cdots x_1 x_0)_b$, $y = (y_{n-1} \cdots y_1 y_0)_b$ with $0 \leq x, y < m$, $R = b^n$ with $\gcd(m, b) = 1$, and $m' = -m^{-1} \bmod b$.
OUTPUT: $xyR^{-1} \bmod m$.

1. $A \leftarrow 0$. (Notation: $A = (a_n a_{n-1} \cdots a_1 a_0)_b$.)
2. For i from 0 to $(n-1)$ do the following:
   2.1 $u_i \leftarrow (a_0 + x_i y_0) m' \bmod b$.
   2.2 $A \leftarrow (A + x_i y + u_i m)/b$.
3. If $A \geq m$ then $A \leftarrow A - m$.
4. Return(A).

When b is a power of 2, mod b simply means cutting off the most-significant bits

When b is a power of 2, /b simply means shifting the bits to the right

Modular Multiplication

Montgomery multiplication:

- If we use Montgomery multiplication directly on the input values x and y, we calculate x*y/R mod n instead of x*y mod n
- Therefore, all inputs need to be transformed to “Montgomery representation” first: $x_M = x*R \bmod n$, $y_M = x*R \bmod n$
- Transforming the inputs to Montgomery representation can be done by multiplying each input with R^2 using Montgomery multiplication: $x_M = \text{MontMul}(x, R^2) = x*R^2/R \bmod n = x*R \bmod n$, $y_M = \text{MontMul}(y, R^2) = y*R^2/R \bmod n = y*R \bmod n$
- The product will also be in Montgomery representation: $a_M = \text{MontMul}(x_M, y_M) = x_M*y_M/R = x*R*y*R/R = x*y*R$
- We stick to Montgomery representation until the entire modular exponentiation is completed. Only at the very end, a transformation is done from Montgomery representation back to normal representation. This can be done by multiplying the result of the modular exponentiation with 1 using Montgomery multiplication: $\text{MontMul}(c_M, 1) = c_M*1/R = c*R*1/R = c$

Modular Multiplication

- Montgomery multiplication in software:
  - Several methods to process the data in smaller chunks
  - Follow the original algorithm (2 slides back) or optimizations first introduced in
    Koc et al., Analyzing and comparing Montgomery multiplication algorithms, in IEEE Micro 16(3), pp. 26-33, 1996.
- Montgomery multiplication in hardware:
  - It is possible to process the data in parallel
  - FPGA: use DSP slices (and dedicated shift registers)
  - ASIC: use multipliers in the standard cell library

Modular Multiplication

- Karatsuba - Ofman Multiplication
- main idea: recursively split into "half-size" multiplications
- until some threshold is reached (when size$<$threshold continue naive method)
- set $R = b^n$, $d = 2n \Rightarrow u = (u_{d-1} \dots u_0)_b$ and $v = (v_{d-1} \dots v_0)_b \Rightarrow U = U_1 R + U_0$ and $V = V_1 R + V_0$

$$uv = U_1 V_1 R^2 + \big((U_0 + U_1)(V_0 + V_1) - U_1 V_1 - U_0 V_0\big) R + U_0 V_0.$$

Algorithm Karatsuba multiplication of positive multiprecision integers

INPUT: An n-word integer $u = (u_{n-1} \dots u_0)_b$, an m-word integer $v = (v_{m-1} \dots v_0)_b$, the size $d = \max\{m, n\}$, and a threshold d_0.
OUTPUT: The $(m+n)$-word integer $w = (w_{m+n-1} \dots w_0)_b$ such that $w = uv$.

1. **if** $d \leqslant d_0$ **then return** uv
2. $p \leftarrow \lfloor d/2 \rfloor$ and $q \leftarrow \lceil d/2 \rceil$
3. $U_0 \leftarrow (u_{q-1} \dots u_0)_b$ and $V_0 \leftarrow (v_{q-1} \dots v_0)_b$
4. $U_1 \leftarrow (u_{p+q-1} \dots u_q)_b$ and $V_1 \leftarrow (v_{p+q-1} \dots v_q)_b$ [pad with 0's if necessary]
5. $U_s \leftarrow U_0 + U_1$ and $V_s \leftarrow V_0 + V_1$
6. compute recursively $U_0 V_0$, $U_1 V_1$ and $U_s V_s$ [corresponding sizes being q, p and q]
7. **return** $U_1 V_1 b^{2q} + \big((U_s V_s - U_1 V_1 - U_0 V_0)\big) b^q + U_0 V_0$