CPSC 418/MATH 318 Introduction to Cryptography

Advanced Encryption Standard

Renate Scheidler

Department of Mathematics & Statistics
Department of Computer Science
University of Calgary

Week 4

Outline

AES Competition

A lesson on how to **definitely** introduce standardized crypto!

In 1997, NIST initiated a world-wide process of candidate submission and evaluation for the *Advanced Encryption Standard* to replace DES.

The process was completely transparent and public!

Requirements:

- possible key sizes of 128, 192, and 256 bits
- plaintexts and ciphertexts of 128 bits
- should work on a wide variety of hardware (from chip cards to supercomputers)
- fast
- secure
- world-wide royalty-free availability (!)

Selection Criteria

Candidates were selected according to:

- security – resistance against all known attacks
- cost — speed and code compactness on a wide variety of platforms
- simplicity of design

Most important: *public* evaluation process

- series of three conferences: algorithms, attacks, evaluations presented and discussed
- 21 submissions from all over the world evaluated during 1998-1999
- final selection done by NIST

The Winner: Rijndael

Rijndael (pronounced "Reign Dahl" or "Rhine Dahl", but NOT "Region Deal" was chosen by NIST.

- Inventors: Vincent Rijmen and Joan Daemen.
- Standardized as AES in 2001 (FIPS 197)
- See also docs on "handouts" page.

The Rijndael algorithm uses two different types of arithmetic:

- Arithmetic on bytes (8 bit vectors — actually, elements of the finite field $GF(2^8)$ of 256 elements)
- 4-byte vectors (actually polynomial operations over $GF(2^8)$).

Rijndael's structure is a *substitution-permutation network*, not a Feistel network. In addition to permutations, it uses one big S-box rather than many small ones acting on substrings.

Rijndael - Round Overview

The algorithm uses addition, multiplication, and inversion on bytes as well as addition and multiplication of 4 byte vectors.

Rijndael is a product cipher, but NOT a Feistel cipher like DES. Instead, it has three *layers* per round:

- a linear mixing layer (ShiftRows, transposition, and MixColumns, a linear transformation; for diffusion over multiple rounds)
- a non-linear layer (SubBytes, substitution, done with an S-box)
- a key addition layer (AddRoundKey, X-OR with key)

Check out the story of AES in the form of a four-act play involvin stick figures at `http://www.moserware.com/2009/09/stick-figure-guide-to-advanced.html`

Arithmetic on Bytes

Consider a byte $b = (b_7, b_6, \ldots, b_1, b_0)$ (an 8-bit vector) as a polynomial with coefficients in $\{0, 1\}$:

$$b \mapsto b(x) = b_7x^7 + b_6x^6 + \cdots + b_1x + b_0 \ .$$

Rijndael makes use of the following operations on bytes, interpreting them as polynomials:

1. Addition
2. Modular multiplication
3. Inversion

Under these operations, polynomials of degree ≤ 7 with coefficients in $\{0, 1\}$ form the *field* $GF(2^8)$.

By associating bytes with these polynomials, we obtain these operations on bytes.

Addition of Bytes in Rijndael

Polynomial addition takes X-OR (addition mod 2) of coefficients:

$$\begin{array}{cccccccccc} & b_7x^7 & + & b_6x^6 & +\cdots+ & b_1x & + & b_0 \\ + & c_7x^7 & + & c_6x^6 & +\cdots+ & c_1x & + & c_0 \\ \hline & (b_7 \oplus c_7)x^7 & + & (b_6 \oplus c_6)x^6 & +\cdots+ & (b_1 \oplus c_1)x & + & (b_0 \oplus c_0) \end{array}$$

The sum of two polynomials taken in this manner yields another polynomial of degree ≤ 7.

In other words, component-wise X-OR of bytes is identified with this addition operation on polynomials.

Modular Multiplication in Rijndael

Polynomial multiplication (coefficients are in $\{0,1\}$) is taken modulo

$$m(x) = x^8 + x^4 + x^3 + x + 1$$

(remainder when dividing by $m(x)$, analogous to modulo arithmetic with integers).

The remainder when dividing by a degree 8 polynomial will have degree ≤ 7. Thus, the "product" of two bytes is associated with the product of their polynomial equivalents modulo $m(x)$.

Note 1

$m(x)$ is the lexicographically first polynomial that is *irreducible* over $GF(2)$, *i.e.* does not split into two polynomials of smaller positive degree with coefficients in $\{0,1\}$.

Inversion of Bytes in Rijndael

$b(x)^{-1}$, the inverse of $b(x) = b_7x^7 + b_6x^6 + \cdots + b_1x + b_0$, is the polynomial of degree ≤ 7 with coefficients in $\{0,1\}$ such that

$$b(x)b(x)^{-1} \equiv 1 \pmod{m(x)} .$$

Note that this is completely analogous to the case of integer arithmetic modulo n.

The "inverse" of the byte $b = (b_7, b_6, \ldots, b_1, b_0)$ is the byte associated with the inverse of $b(x) = b_7x^7 + b_6x^6 + \cdots + b_1x + b_0$.

Rijndael uses inverse as above in its SubByte operation.

Arithmetic on 4-byte Vectors

In Rijndael's MixColumn operation, 4-byte vectors are considered as degree 3 polynomials with coefficients in $GF(2^8)$. That is, the 4-byte vector (a_3, a_2, a_1, a_0) is associated with the polynomial

$$a(y) = a_3y^3 + a_2y^2 + a_1y + a_0,$$

where each coefficient is a byte viewed as an element of $GF(2^8)$ (addition, multiplication, and inversion of the coefficients is performed as described above).

Operations on 4-byte Vectors

We have the following operations on these polynomials:

1. addition: component-wise "addition" of coefficients (addition as described above)
2. multiplication: polynomial multiplication (addition and multiplication of coefficients as described above) modulo $M(y) = y^4 + 1$. Result is a degree 3 polynomial with coefficients in $GF(2^8)$.

Note 2

Using $M(y) = y^4 + 1$ makes for very efficient arithmetic (simple circular shifts)

Examples for Rijndael Arithmetic

Example 1

Let $b_1 = (10001110)$ and $b_2 = (00001101)$ be bytes. Compute $b_3 = b_1 + b_2$ and $b_4 = b_1 b_2$ in the Rijndael field $GF(2^8)$.

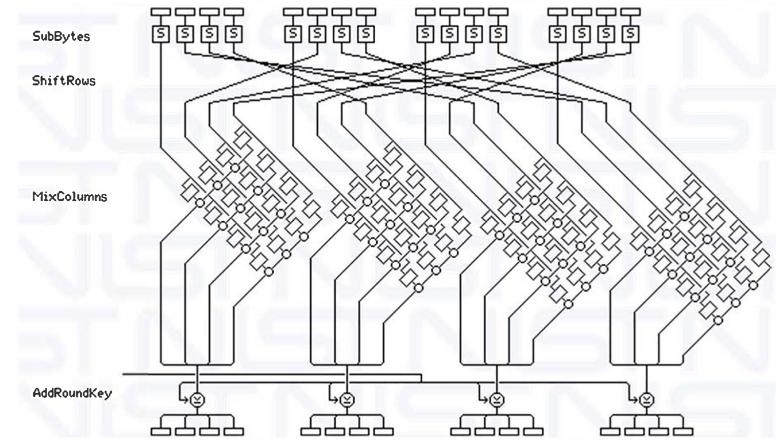
Example 2

$$\text{Let}\quad a_1 = (00000001,\ 00000000,\ 10001110,\ 00000010) \quad \text{and}$$
$$a_2 = (00000000,\ 00000001,\ 00001101,\ 00000000)$$

be vectors whose entries are bytes in the Rijndael field $GF(2^8)$. Compute $a_3 = a_1 + a_2$ and $a_4 = a_1 a_2$ using Rijndael's arithmetic on 4-byte vectors.

See the Rijndael arithmetic examples handout on the "handouts" page.

Rijndahl Overview



Graphic taken with modifications from the cover of NIST GCR 18-017 "The Economic Impacts of the Advanced Encryption Standard, 1996-2017" (NIST, September 2018)

Rijndael Properties

Designed for block sizes and key lengths to be any multiple of 32, including those specified in the AES.

Iterated cipher: number of rounds N_r depends on the key length. 10 rounds for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys.

Algorithm operates on a 4×4 array of bytes (8 bit vectors) called the *state*:

$s_{0,0}$	$s_{0,1}$	$s_{0,2}$	$s_{0,3}$
$s_{1,0}$	$s_{1,1}$	$s_{1,2}$	$s_{1,3}$
$s_{2,0}$	$s_{2,1}$	$s_{2,2}$	$s_{2,3}$
$s_{3,0}$	$s_{3,1}$	$s_{3,2}$	$s_{3,3}$

AES Initialization

The Rijndael algorithm (given plaintext M) proceeds as follows:

1. Initialize `State` with M :

$$\begin{array}{|c|c|c|c|} \hline s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ \hline s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ \hline s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ \hline s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \\ \hline \end{array} \leftarrow \begin{array}{|c|c|c|c|} \hline m_0 & m_4 & m_8 & m_{12} \\ \hline m_1 & m_5 & m_9 & m_{13} \\ \hline m_2 & m_6 & m_{10} & m_{14} \\ \hline m_3 & m_7 & m_{11} & m_{15} \\ \hline \end{array}$$

where M consists of the 16 bytes $m_0, m_1, \ldots, m_{15}$.

AES Algorithm

On input the `State` whose columns are the 16 message bytes:

2. Perform AddRoundKey, which X-OR's the first `RoundKey` with `State`.
3. For each of the first $N_r - 1$ rounds:
   - Perform SubBytes on `State` (using an S-box on each byte of `State`),
   - Perform ShiftRows (a permutation) on `State`,
   - Perform MixColumns (a linear transformation) on `State`,
   - Perform AddRoundKey.
4. For the last round:
   - Perform SubBytes,
   - Perform ShiftRows,
   - Perform AddRoundKey.
5. Define the ciphertext C to be `State` (using the same byte ordering).

The SubBytes Operation

Each byte of `State` is substituted independently, using an invertible S-box (see p. 16 of FIPS 197 for the exact S-Box).

Algebraically, SubBytes performs on each byte:

- an inversion as described above (the inverse of the zero byte is defined to be zero here), followed by
- an affine transformation, *i.e.* a linear transformation (multiplication by a matrix) followed by the addition of a fixed vector. More exactly, the i-th bit of the output byte is

  $$b_i' = b_i \oplus b_{i+4 \bmod 8} \oplus b_{i+5 \bmod 8} \oplus b_{i+6 \bmod 8} \oplus b_{i+7 \bmod 8} \oplus c_i$$

  where b_i is the i-th input bit and c_i is the i^{th}-th bit of $c = (11000110)$.

The SubBytes Affine Transformation

An *affine* transformation first multiplies a vector by a matrix (i.e. a linear transformation) and then adds a vector to the result (which makes it non-linear)

The affine transformation in SubBytes is given as follows:

$$\begin{bmatrix} b_0' \\ b_1' \\ b_2' \\ b_3' \\ b_4' \\ b_5' \\ b_6' \\ b_7' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Inverse of SubBytes

The inverse of SubBytes (called InvSubBytes) applies the inverse S-box to each byte in the `State` (see p. 22 of FIPS 197 for the inverse of the S-Box).

Algebraically, you first apply the inverse affine transformation (add the vector, then multiply by the inverse of the matrix) to each byte and then perform byte inversion.

The ShiftRows Operation

Shifts the first, second, third, and last rows of `State` by 0, 1, 2, or 3 cells to the left, respectively:

$$\begin{array}{|c|c|c|c|}\hline s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ \hline s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ \hline s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ \hline s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \\ \hline \end{array} \leftarrow \begin{array}{|c|c|c|c|}\hline s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ \hline s_{1,1} & s_{1,2} & s_{1,3} & s_{1,0} \\ \hline s_{2,2} & s_{2,3} & s_{2,0} & s_{2,1} \\ \hline s_{3,3} & s_{3,0} & s_{3,1} & s_{3,2} \\ \hline \end{array}$$

The inverse operation InvShiftRows applies right shifts instead of left shifts.

The MixColumns Operation

Each column of `State` is a 4-byte vector which can be interpreted as a four-term polynomial with coefficients in $GF(2^8)$ as described above. For example:

$$(s_{0,0}, s_{1,0}, s_{2,0}, s_{3,0}) \mapsto s_{3,0}y^3 + s_{2,0}y^2 + s_{1,0}y + s_{0,0} = col_0(x) \ .$$

MixColumns multiplies $col_i(y)$ by a polynomial $c(y)$ using the 4-byte vector multiplication modulo $y^4 + 1$ described earlier, resulting in a new 4-byte column. Here

$$\begin{aligned} c(y) &= 3y^3 + y^2 + y + 2 \ \text{ in hexadecimal} \\ &= 00000011\, y^3 + 00000001\, y^2 + 00000001\, y + 00000010 \ \text{ in binary} \end{aligned}$$

MixColumns: Algebraic Description

MixColumns can also be described as a linear transformation applied to each column of `State`, *i.e.* multiplying each 4-element column vector by the 4×4 matrix

$$\begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix}$$

Note that rows 0, 1, 2, 3 of this matrix are circular shifts of row 0 by 0, 1, 2, 3 cells to the right.

InvMixColumns: Algebraic Description

The inverse (called InvMixColumns) multiplies each column of `State` by the inverse of $c(y)$ (mod $y^4 + 1$) which is

$$c^{-1}(y) = By^3 + Dy^2 + 9y + E$$

in hex notation.

It can also be described as multiplication by the following matrix (in hex):

$$\begin{pmatrix} E & B & D & 9 \\ 9 & E & B & D \\ D & 9 & E & B \\ B & D & 9 & E \end{pmatrix}$$

The ADDROUNDKEY Operation

In ADDROUNDKEY, each column of State is X-ORed with one word of the round key:

$$col_j \leftarrow col_j \oplus w_{4i+j} \qquad (0 \le i \le N_r - 1,\ 0 \le j \le 3)$$

$$\begin{array}{|c|c|c|c|}\hline s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ \hline s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ \hline s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ \hline s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \\ \hline \end{array} \leftarrow \begin{array}{|c|c|c|c|}\hline s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ \hline s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ \hline s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ \hline s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \\ \hline \end{array} \oplus \begin{array}{|c|c|c|c|}\hline w_{0,4i} & w_{0,4i+1} & w_{0,4i+2} & w_{0,4i+3} \\ w_{1,4i} & w_{1,4i+1} & w_{1,4i+2} & w_{1,4i+3} \\ w_{2,4i} & w_{2,4i+1} & w_{2,4i+2} & w_{2,4i+3} \\ w_{3,4i} & w_{3,4i+1} & w_{3,4i+2} & w_{3,4i+3} \\ \hline \end{array}$$

where $w_{4i+j} = (w_{0,4i+j}, w_{1,4i+j}, w_{2,4i+j}, w_{3,4i+j})$, $0 \le j \le 3$ are the four 4-byte key words for round i, $0 \le i \le N_r - 1$.

ADDROUNDKEY is clearly its own inverse.

Key Schedule

The key schedule uses:

- the S-box from SubBytes
- cyclic left shifts by one byte on 4-byte vectors
- multiplication by powers of x (each such power is interpreted as first byte of a 4-byte vector whose other bytes are 0)

Consider 128-bit Rijndael. There are 10 rounds plus one preliminary application of ADDROUNDKEY, so the key schedule must produce 11 round keys, each consisting of four 4-byte words, from the 128-bit key (16 bytes).

KEYEXPANSION

Produces an expanded key consisting of the required 44 words (assuming 128-bit key).

In the following, the key $K = (k_0, k_1, k_2, k_3)$, where the k_i are 4-byte words, and the expanded key is denoted by the word-vector $(w_0, w_1, w_2, \ldots, w_{43})$.

1. for $i \in \{0, 1, 2, 3\}$, $w_i = k_i$
2. for $i \in \{4, \ldots, 43\}$:

$$w_i = w_{i-4} \oplus \begin{cases} \text{SUBWORD}(\text{ROTWORD}(w_{i-1})) \oplus \text{RCON}_{i/4} & \text{if } 4 \mid i \\ w_{i-1} & \text{otherwise} \end{cases}$$

KEYEXPANSION (cont'd)

The components of KEYEXPANSION are:

- ROTWORD is a one-byte circular left shift on a word.
- SUBWORD performs a byte substitution (using the S-box SUBBYTES on each byte of its input word).
- RCON is a table of round constants (RCON_j is used in round j). Each is a word with the three right-most bytes equal to 0 and the left-most byte a power of x
  - Corresponding to polynomials $R(y) = by^3$ where $b(x)$ = some power of x

KEYEXPANSION is similar for 192 and 256-bit keys.

Decryption

To decrypt, perform cipher in reverse order, using inverses of components and the reverse of the key schedule:

1. AddRoundKey with round key N_r
2. For rounds $N_r - 1$ to 1 :
   - InvShiftRows
   - InvSubBytes
   - AddRoundKey
   - InvMixColumns
3. For round 1 :
   - InvShiftRows
   - InvSubBytes
   - AddRoundKey using round key 1

Note 3

Straightforward inverse cipher has a different sequence of transformations in the rounds. It is possible to reorganize this so that the sequence is the same as that of encryption (see A2 of FIPS-197).

Strengths of Rijndael

Secure against all known attacks at the time; some newer attacks seem to pose no real threat

Non-linearity resides in S-boxes (SubBytes):

- Linear approximation and difference tables are close to uniform (thwarting linear and differential cryptanalysis — more later)
- No fixed points ($S(a) = a$) or opposite fixed points ($S(a) = \bar{a}$)
- Not an involution ($S(S(a)) \neq a$, or equivalently, $S(a) = S^{-1}(a)$)

ShiftRows and MixColumns ensure that after a few rounds, all output bits depend on all input bits (great diffusion).

Strengths (cont'd)

Secure key schedule (great confusion):

- Knowledge of part of the cipher key or round key does not enable calculation of many other round key bits
- Each key bit affects many round key bits

Very low memory requirements

Very fast (hardware and software)

Weaknesses of Rijndael

Decryption is slower than encryption.

Decryption algorithm is different from encryption (requires separate circuits and/or tables).

- Depending on the mode of operation, however, this may not be an issue (*i.e.* OFB, CTR, CFB) since only encryption is used in these modes.

Security of AES

There is no mathematical proof that AES is secure

All we know is that in practice, it withstands all modern attacks.

Next: an overview of modern attacks on block ciphers

Attacking Block Ciphers – Exhaustive Search

Brute-force search for the key is the simplest attack on a block cipher.

Set $N = |\mathcal{K}|$ (number of keys).

Simple exhaustive search (COA) — requires up to N encryptions

- feasible for DES: $N = 2^{56} \approx 10^{17}$ possible keys.
- infeasible for 3DES: $N = 2^{112} \approx 10^{34}$ possible key combinations.
- infeasible for AES: $N = 2^{128} \approx 10^{38}$ possible keys

Parallelism can speed up exhaustive search.

Perspective on 10^{38}: number of molecules in 3 trillion liters of water (almost 2 Lake Ontarios)
number of stars in a quadrillion universes

Hellman's Time-Memory Tradeoff (1980)

KPA that shortens search time by using a lot of memory (details omitted here).

- The attacker knows a plaintext/ciphertext pair (M_0, C_0).
- The goal is to find the (or a) key K such that $C_0 = E_K(M_0)$.

Expected approximate cost (# of test encryptions) is

Precomputation time:	N
Expected time:	$N^{2/3}$
Expected memory:	$N^{2/3}$

Large precomputation time, but improvement for individual keys

- For DES, $N^{2/3} \approx 10^{12}$ — can be done in hours or even minutes on a modern computer.

Meet-in-the-Middle Attack on Double Encryption

Naïve exhaustive search for double encryption requires up to N^2 encryptions (N^2 key pairs).

The meet-in-the-middle attack is a much faster KPA, but more memory-intense.

Setup:

- Adversary has two known plaintext/ciphertexts pairs (m_1, c_1), (m_2, c_2) (one for key search, the other for checking correct guess)
- Assume double-encryption: $c_i = E_{k_1}(E_{k_2}(m_i))$ for $i = 1, 2$, where k_1, k_2 are two unknown keys.

Important observation: $D_{k_1}(c_i) = E_{k_2}(m_i)$ for $i = 1, 2$.

The Attack

The adversary proceeds as follows:

1. Single-encrypt m_1 under every key K_i to compute $C_i = E_{K_i}(m_1)$ for $1 \le i \le N$.
2. Sort the table (or create a hash table or look-up table) of all the C_i, $1 \le i \le N$.
3. For $j = 1$ to N do
   a. Single-decrypt c_1 under key K_j to compute $M_j = D_{K_j}(c_1)$.
   b. Search for M_j in the table of C_i. If $M_j = C_i$ for some i, i.e. $D_{K_j}(c_1) = E_{K_i}(m_1)$, then check if $D_{K_j}(c_2) = E_{K_i}(m_2)$. If yes, then guess $k_2 = K_i$ and $k_1 = K_j$ and quit.

Analysis

There are at most N values $E_{K_i}(m_1)$ and at most N values $D_{K_j}(c_1)$ for $1 \le i, j \le N$.

- Assuming random distribution, the chances of a match are $1/N$.
- Thus, $(N \cdot N)/N = N$ key pairs (K_i, K_j) satisfy $E_{K_i}(m_1) = D_{K_j}(c_1)$.

The chances that such a key pair also satisfies $E_{K_i}(m_2) = D_{K_j}(c_2)$ are very small (paranoid users could try a third message/ciphertext pair (m_3, c_3)).

Thus, the probability of guessing correctly is very high.

Analysis (cont'd)

Time required:

- Step 1: N encryptions
- Step 2: of order N (hash table) or $N \log(N)$ (sorting)
- Step 3 a: at most N decryptions
- Step 3 b: negligible in light of Step 2

Total: $2N$ encryptions/decryptions plus table creation

Memory: N keys and corresponding ciphertexts (the table of (C_i, K_i) pairs)

Conclusion: double encryption offers little extra protection over single encryption (hence 3DES instead of 2DES).