Block ciphers

Dr. Yee Wei Law ⟨yeewei.law@unisa.edu.au⟩

2023-09-12

Contents

Acronyms

AES Advanced Encryption Standard 2
CRYPTREC Cryptography Research and Evaluation Committee 4
DES Data Encryption Standard 2
ECB electronic cookbook 2
NESSIE New European Schemes for Signatures, Integrity and Encryption 4
TDEA Triple Data Encryption Algorithm 2

1 Introduction

Whereas a stream cipher contains a memory, embodied in its current state, a block cipher is *memoryless* outside its current block and therefore has no current state [PBO+03, Sec. 2.1].

Definition 1: Block cipher [Gol04, Definition 5.3.5]

A block cipher is a triple of PPT algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ satisfying two conditions:

1. On input 1^n, algorithm Gen outputs a pair of bit strings (keys).
2. There exists a polynomially bounded function $\ell : \mathbb{N} \to \mathbb{N}$, called the *block length*, such that for every pair (e, d) in the range of $\mathsf{Gen}(1^n)$, and for each $m \in \{0,1\}^{\ell(n)}$, algorithms Enc and Dec satisfy

$$\Pr\{\mathsf{Dec}_d(\mathsf{Enc}_e(m)) = m\} = 1.$$

In Definition 1, $\mathsf{Gen}(1^n)$ outputs a pair of keys that can be

- different, in which case we get a public-key block cipher;
- the same, in which case we get a symmetric-key block cipher.

A block cipher is commonly assumed/understood to be a symmetric-key block cipher.

NIST SP 800-175B [Bar20] defines a block cipher as ❝ a family of functions and their inverse functions that is parameterised by cryptographic keys; the functions map bit strings of a *fixed length* to bit strings of the same length. ❞ 👈 This definition is consistent with Definition 1, although not as precise.

Some authors are more specific in their definition of block ciphers. For example, Katz and Lindell [KL21, Sec. 3.6.3] define a block cipher as a ❝ strong pseudorandom permutation ❞ [Gol01, Definition 3.7.5]. 👈 This definition includes key generation implicitly and is specific about the properties of `Enc` and `Dec`.

- Watch Dan Boneh's ▶ "What are block ciphers?", which introduces block ciphers as pseudorandom permutation functions.
- For the purpose of this course, we shall separate the assumption/construction (e.g., pseudorandom permutation) and the implicit security notion from the definition itself.

Currently, there are two NIST-approved block ciphers, namely

1. Advanced Encryption Standard (AES), see Sec. 2;
2. Triple Data Encryption Algorithm (TDEA), also called Triple DES.

The block ciphers Skipjack [Sch15, Sec. 13.12] and Data Encryption Standard (DES) [Sch15, Ch. 12] were previously approved but their approval has been withdrawn; watch ▶ LinkedIn Learning video.

Table 1 summarises the standards and approval statuses.

Table 1: Standards and approval statuses of block ciphers [BR19, Sec. 2].

Block cipher	Standard	Status
AES	FIPS 197	Acceptable
Two-key TDEA encryption	NIST SP 800-67	Disallowed
Two-key TDEA decryption		Legacy use
Three-key TDEA encryption	NIST SP 800-67	Deprecated through 2023, disallowed after 2023
Three-key TDEA decryption		Legacy use
Skipjack encryption	FIPS 185	Disallowed
Skipjack decryption		Legacy use
DES encryption	FIPS 46-3	Disallowed
DES decryption		Legacy use

Applying a block cipher to the same input block will always produce the same output block when the same key is used [Bar20, Sec. 3.2.1.5]. 👈 This naive mode of operation, as illustrated in Figure 1, is called *electronic cookbook* (ECB) mode.

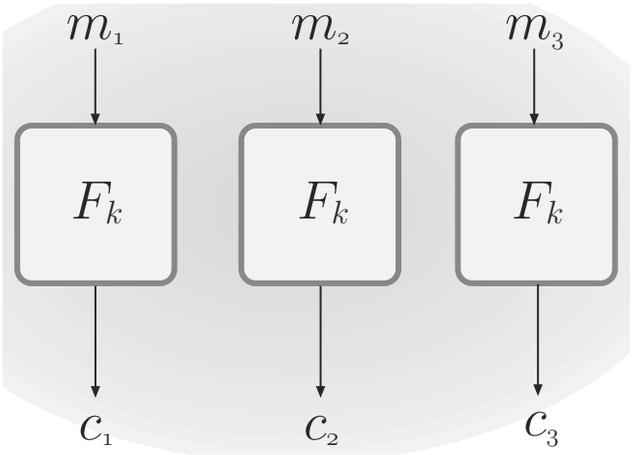


Figure 1: Applying block cipher F_k in ECB mode [KL21, Figure 3.4].

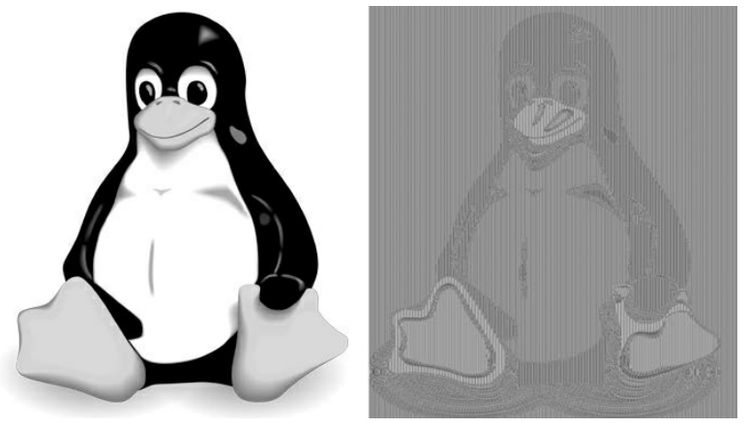


Figure 2: Image of Tux the penguin in plaintext and in ECB-mode ciphertext [Aum18, Figure 4-7].

In the ECB mode, data patterns in the plaintext, such as repeated blocks, remain apparent in the ciphertext, as evident in Figure 2.

If the multiple blocks in a typical message are encrypted separately, an adversary can easily substitute individual blocks, possibly without detection [Bar20, Sec. 3.2.1.5].

⚠ ECB mode is deterministic and *not* semantically secure [KL21, p. 89].

Quiz 1

Can you device an IND experiment that shows violation of IND-PASS?

ℹ Detail: Why the name "electronic codebook"?

If a plaintext block is always encrypted to the same ciphertext block, whenever the same key is used, then an attacker can build a codebook of plaintext-ciphertext pairs that it can look up whenever it needs to encrypt/decrypt some text. 👈 This is called a *codebook attack*.

If the block size is 2^b bits for $b \in \mathbb{N}$, an *electronic cookbook* needs at least $\underbrace{2^{2^b}}_{\text{\#rows}} \times \underbrace{2^b}_{\text{\#bits per row}} = 2^{2^b+b}$ bits.

- When $b = 4$, we need 128 kilobytes.
- When $b = 5$, we need 16 gigabytes.
- When $b = 6$, we need 128 exabytes.

The codebook attack not only rules out usage of the ECB mode but also indicates the need for a sufficiently large block size, i.e., at least $2^6 = 64$ bits in a block. AES uses $2^7 = 128$-bit blocks.

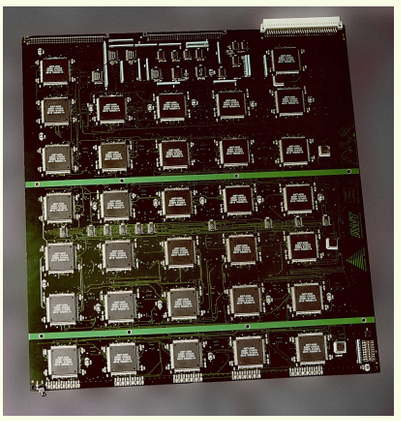
Other modes of operation than ECB for block ciphers are discussed in the next lecture (see knowledge base entry).

2 AES

The significance of the AES warrants a quick review of the historical context:

Detail: Historical context

The DES algorithm was first published in 1975 and then standardised in FIPS PUB 46 in 1977.



DES was theoretically secure but its small key length of 56 bits rendered it vulnerable to exhaustive (key) search attacks.

- In 1997, RSA Security organised the first DES Challenge, and the key was found in 96 days.
- In the 1999 round of DES Challenge, the key was found in 22 hours 15 minutes using a purpose-built machine called "Deep Crack" and 100,000 computers during their idle time [McN99].
- Watch Dan Boneh's "Exhaustive search attacks".

In January 1997, NIST initiated the development of the successor to DES, namely the AES, inviting proposals from the global community.

So that it can be used to protect sensitive government information well into the 21st century, the AES must support a block size of 128 bits, and key sizes of 128, 192 and 256 bits.

In August 1998, NIST shortlisted 15 candidates.

In October 2000, NIST announced the selection of Joan Daemen and Vincent Rijmen's Rijndael as the AES.

In November 2001, the AES was published as FIPS 197 [NIS01].

In 2003, the New European Schemes for Signatures, Integrity and Encryption (NESSIE) project formally recommended Rijndael/AES [Pre03].

At about the same time, the (Japanese) Cryptography Research and Evaluation Committee (CRYPTREC) also formally recommended Rijndael/AES, which remains a recommendation as of 2022 [CRY22].

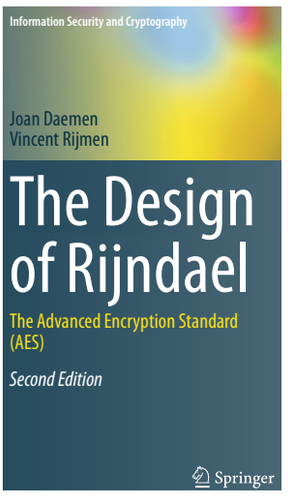
As of 2020, more than 5700 AES algorithm implementations had been validated by Cryptographic Algorithm Validation Program (CAVP) as conforming to FIPS 197 specifications [NIS21].

Nowadays, almost all modern 64-bit processors have native instructions for AES [Mou21], e.g., Intel has AES New Instructions (AES-NI) for not only accelerating AES computation but also mitigating timing and cache-based attacks [Gue09].

Quiz 2

In [CRY22], what is the other recommended 128-bit block cipher than AES?

Our discussion shall be guided by FIPS 197 [NIS01], [DR20] (👉) and the OpenSSL implementation of AES (which is different from BoringSSL implementation of AES).



A rigorous coverage of the AES would cover finite fields, linear codes and boolean functions first [DR20, Ch. 2], but only a user's view of finite fields will be covered in Sec. 2.1.

Rijndael was designed to support a variable block length and a variable key length; both can be independently specified to any multiple of 32 bits, with a minimum of 128 bits and a maximum of 256 bits.

The AES is thus, strictly speaking, a subset of Rijndael with the block length fixed at 128 bits, and key lengths limited to 128, 192 and 256 bits.

For a quick overview of the AES, watch Dan Boneh's ▶ "The AES Block Cipher".

For our discussion here, an unconventional starting point is taken, by inspecting the OpenSSL source code at `https://github.com/openssl/openssl/tree/master/crypto/aes`.

- We shall look for an implementation of the AES in ECB mode (which is basically the AES itself).
- The security of the AES depends on viewing a single byte of data in two different ways: 1️⃣ a byte as an eight-dimensional vector over the Galois field GF(2); 2️⃣ a byte as an element of the Galois field GF(2^8).

Rationale for these two views ☝: operations that are easy to analyse in one representation are difficult to analyse in another [CMR06], thwarting cryptanalysts' effort to formulate straightforward algebraic attacks.

We discuss arithmetic in Galois fields — an important part of the AES — in Sec. 2.1.

- We then discuss the key schedule in Sec. 2.2, before looking into encryption/decryption in Sec. 2.3.

2.1 Arithmetic in GF(2^n)

A *Galois field*, also called a *finite field*, which in turn is a *field* (see Definition 2) with a finite number of elements.

Definition 2: Field [HW03, p. 176]

A field is a triple $(\mathbb{F}, +, \times)$, consisting of set $\mathbb{F}$ and binary operations $+$ and $\times$ on $\mathbb{F}$, that satisfies the following axioms:

Closure $\forall a, b \in \mathbb{F}$,
- $a + b \in \mathbb{F}$,
- $a \times b \in \mathbb{F}$.

Associativity	$\forall a, b, c \in \mathbb{F}$, • $(a + b) + c = a + (b + c)$, • $(a \times b) \times c = a \times (b \times c)$.
Commutativity	$\forall a, b \in \mathbb{F}$, • $a + b = b + a$, • $a \times b = b \times a$.
Identity/neutral element	$\forall a \in \mathbb{F}$, • there exists an additive identity element $0 \in \mathbb{F}$ such that $a + 0 = a$, • there exists a multiplicative identity element $1 \neq 0$ and $1 \in \mathbb{F}$ such that $a \cdot 1 = a$. ⚠ 0 and 1 above ☝ serve as symbols rather than literal zero and one.
Inverse element	$\forall a \in \mathbb{F}$, • there exists an additive inverse element $-a \in \mathbb{F}$ such that $a + (-a) = 0$, • except for element 0, there exists a multiplicative inverse element $a^{-1} \in \mathbb{F}$ such that $a \times a^{-1} = 1$.
Distributivity	$\times$ is distributive over $+$, i.e., $\forall a, b, c \in \mathbb{F}$, $(a + b) \times c = (a \times c) + (b \times c)$.

Quiz 3

Is $a \times (b + c) = (a \times b) + (a \times c)$ true $\forall a, b, c \in \mathbb{F}$?

ℹ Detail: Field ← ring ← group [LN94, p. 12], [Lov22, Definitions 1.2.1, 3.1.1, 3.1.22]

The study of Galois fields belongs to the field of *abstract algebra* (see knowledge base entry).

An introduction to Galois fields typically starts with *groups* followed by *rings*, but let us trace backwards from fields to rings to groups:

- A *field* is a *commutative division ring*.
- A *division ring* is a *ring* whose elements except 0 form a *group* under multiplication. 👈 This means all elements except 0 has a multiplicative inverse.
- A *ring* is a triple $(R, +, \times)$, where 1️⃣ $(R, +)$ forms a *commutative group*, 2️⃣ $\times$ is associative, and 3️⃣ $\times$ is distributive over $+$.
- And finally, a *group* is a pair $(G, \star)$, consisting of set G and binary operation $\star$ on G, that satisfies the three axioms: 1️⃣ $\star$ is associative, 2️⃣ G has an identity element with respect to $\star$, and 3️⃣ every element in G has an inverse with respect to $\star$.

A Galois field is essentially a finite set on which we can perform the arithmetic operations of addition, subtraction, multiplication and division (but not division by zero).

The number of elements in a Galois field is called the *order* or *cardinality* of the field.

- The order of a Galois field is necessarily of the form p^m, where p is a prime number and m is any positive integer [Kib17, Proposition 2.2].
- Furthermore, all Galois fields of the same order are isomorphic to each other [Kib17, Proposition 2.3], thus all Galois fields of the same order share the same notation $\mathrm{GF}(p^m)$.

Example 1

✔️ The triple $(\{0, 1, \ldots, p-1\}, + \bmod p, \times \bmod p)$ is isomorphic to the Galois field $\mathrm{GF}(p)$.

❌ The triple $(\{0, 1, 2, 3, 4, 5\}, + \bmod 6, \times \bmod 6)$ is not a Galois field, since the elements $2, 3, 4$ do not have a multiplicative inverse.

❌ The triple $(\{0, 1, 2, 3\}, + \bmod 4, \times \bmod 4)$ is not a Galois field, since the element 2 does not have a multiplicative inverse, although there are exactly 2^2 elements.

A structure having order of the form p^m is not necessarily a Galois field, but a Galois field has necessarily order of the form p^m.

Consider the set of n-bit strings:

- They can be viewed as elements of the Galois field $\mathrm{GF}(2^n)$.
- They can also be viewed as binary polynomials of degree up to $n-1$, e.g., the 4-bit string 1001 can be viewed as the polynomial $x^3 + 1$.

In other words, we can view the Galois field $\mathrm{GF}(2^n)$ as a set of binary polynomials of degree up to $n-1$:

- Elements of $\mathrm{GF}(2^n)$ can be added modulo 2:

$$\sum_{i=0}^{n-1} c_i x^i + \sum_{i=0}^{n-1} d_i x^i = \sum_{i=0}^{n-1} (c_i \oplus d_i) x^i.$$

  🔎 Note addition modulo 2 is equivalent to XOR ☝. x is a so-called indeterminate, which merely serves as a symbol.
- Elements of $\mathrm{GF}(2^n)$ can be multiplied modulo an irreducible polynomial (i.e., a polynomial whose divisors are one and itself) of degree n:

$$\left(\sum_{i=0}^{n-1} c_i x^i\right)\left(\sum_{i=0}^{n-1} d_i x^i\right) \bmod m(x).$$

  For any Galois field of $n > 2$, multiple irreducible polynomials are available for choosing.

Example 2

For Galois field $\mathrm{GF}(2^n)$, we can determine the *primitive polynomials* (a subset of irreducible polynomials) using the command `primpoly` from the MATLAB Communications Toolbox.

The following shows the MATLAB code and associated outputs for $\mathrm{GF}(2^2)$ and

GF(2^8):

```
>> primpoly(2, 'all')

Primitive polynomial(s) =

D^2+D^1+1

ans =

     7
```

```
>> primpoly(8, 'all')

Primitive polynomial(s) =

D^8+D^4+D^3+D^2+1
D^8+D^5+D^3+D^1+1
D^8+D^5+D^3+D^2+1
D^8+D^6+D^3+D^2+1
D^8+D^6+D^4+D^3+D^2+D^1+1
D^8+D^6+D^5+D^1+1
D^8+D^6+D^5+D^2+1
D^8+D^6+D^5+D^3+1
D^8+D^6+D^5+D^4+1
D^8+D^7+D^2+D^1+1
D^8+D^7+D^3+D^2+1
D^8+D^7+D^5+D^3+1
D^8+D^7+D^6+D^1+1
D^8+D^7+D^6+D^3+D^2+D^1+1
D^8+D^7+D^6+D^5+D^2+D^1+1
D^8+D^7+D^6+D^5+D^4+D^2+1

ans =

    285
    299
    301
    333
    351
    355
    357
    361
    369
    391
    397
    425
    451
    463
    487
    501
```

To interpret the integer results, e.g., 7 ☝, observe 1️⃣ the coefficients of the associated polynomial $D^2 + D^1 + 1$ are 1, 1, 1; and 2️⃣ $111_2 \equiv 7_{10}$. 👈 We shall use this primitive polynomial in the next example.

Alternatively, we can use the function `irreducible_polys` from the Python package `galois` to determine the irreducible polynomials for GF(p^m).

The following shows the Python code and associated output for GF(2^8):

```
import galois
list(galois.irreducible_polys(2, 8))
```

```
[Poly(x^8 + x^4 + x^3 + x + 1, GF(2)),
 Poly(x^8 + x^5 + x^3 + x + 1, GF(2)),
 Poly(x^8 + x^5 + x^4 + x^3 + x^2 + x + 1, GF(2)),
 Poly(x^8 + x^5 + x^3 + x^2 + 1, GF(2)),
 Poly(x^8 + x^5 + x^4 + x^3 + 1, GF(2)),
 Poly(x^8 + x^4 + x^3 + x^2 + 1, GF(2)),
 Poly(x^8 + x^6 + x^3 + x^2 + 1, GF(2)),
 Poly(x^8 + x^6 + x^4 + x^3 + x^2 + x + 1, GF(2)),
```

```
Poly(x^8 + x^6 + x^5 + x + 1, GF(2)),
Poly(x^8 + x^6 + x^5 + x^2 + 1, GF(2)),
Poly(x^8 + x^6 + x^5 + x^3 + 1, GF(2)),
Poly(x^8 + x^6 + x^5 + x^4 + 1, GF(2)),
Poly(x^8 + x^6 + x^5 + x^4 + x^2 + x + 1, GF(2)),
Poly(x^8 + x^6 + x^5 + x^4 + x^3 + x + 1, GF(2)),
Poly(x^8 + x^7 + x^2 + x + 1, GF(2)),
Poly(x^8 + x^7 + x^3 + x + 1, GF(2)),
Poly(x^8 + x^7 + x^3 + x^2 + 1, GF(2)),
Poly(x^8 + x^7 + x^4 + x^3 + x^2 + x + 1, GF(2)),
Poly(x^8 + x^7 + x^5 + x + 1, GF(2)),
Poly(x^8 + x^7 + x^5 + x^3 + 1, GF(2)),
Poly(x^8 + x^7 + x^5 + x^4 + 1, GF(2)),
Poly(x^8 + x^7 + x^5 + x^4 + x^3 + x^2 + 1, GF(2)),
Poly(x^8 + x^7 + x^6 + x + 1, GF(2)),
Poly(x^8 + x^7 + x^6 + x^3 + x^2 + x + 1, GF(2)),
Poly(x^8 + x^7 + x^6 + x^4 + x^2 + x + 1, GF(2)),
Poly(x^8 + x^7 + x^6 + x^4 + x^3 + x^2 + 1, GF(2)),
Poly(x^8 + x^7 + x^6 + x^5 + x^2 + x + 1, GF(2)),
Poly(x^8 + x^7 + x^6 + x^5 + x^4 + x + 1, GF(2)),
Poly(x^8 + x^7 + x^6 + x^5 + x^4 + x^2 + 1, GF(2)),
Poly(x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + 1, GF(2))]
```

Example 3

Consider the Galois field GF(2^2) where addition modulo 2 is equivalent to XOR and multiplication is modulo the irreducible polynomial $m(x) \triangleq x^2+x+1$ obtained in the preceding example.

While addition/XOR is straightforward, multiplication deserves a closer look.

We can generate a multiplication table through the *modular* outer product [HW03, p. 187]:

$$\begin{bmatrix} 0 \\ 1 \\ x \\ x+1 \end{bmatrix} \begin{bmatrix} 0 & 1 & x & x+1 \end{bmatrix} \bmod m(x) = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & x & x+1 \\ 0 & x & x^2 & x(x+1) \\ 0 & x+1 & (x+1)x & (x+1)^2 \end{bmatrix} \bmod m(x)$$
$$= \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & x & x+1 \\ 0 & x & -x-1 & -1 \\ 0 & x+1 & -1 & x \end{bmatrix}. \tag{1}$$

Four of the elements in the preceding matrix have order 2 and have been reduced modulo $m(x)$, e.g.,

$$x^2 \bmod m(x) = (x^2+x+1)(1) + (-x-1) \bmod (x^2+x+1) = -x-1.$$

Furthermore, the negative ones in the final matrix of Eq. (1) are equivalent to (1 mod 2), thus one final reduction modulo 2 gives us the multiplication table:

$\times$	0	1	x	$x+1$
0	0	0	0	0
1	0	1	x	$x+1$
x	0	x	$x+1$	1
$x+1$	0	$x+1$	1	x

The table above is readily verifiable using the following Python code (output omitted):

```
import galois
GF = galois.GF(2**2, repr="poly")
print(GF.arithmetic_table("*"))
```

Applying the binary representation to the polynomials in the preceding table gives us the equivalent multiplication table:

×	00	01	10	11
00	00	00	00	00
01	00	01	10	11
10	00	10	11	01
11	00	11	01	10

We can further convert the binary representation to the integer representation using the Python code:

```
import galois
GF = galois.GF(2**2, repr="int")
print(GF.arithmetic_table("*"))
print(GF.repr_table())
```

```
x * y | 0  1  2  3
------+------------
    0 | 0  0  0  0
    1 | 0  1  2  3
    2 | 0  2  3  1
    3 | 0  3  1  2
 Power    Polynomial    Vector    Integer
-------  ------------  --------  ---------
   0            0        [0, 0]       0
  x^0           1        [0, 1]       1
  x^1           x        [1, 0]       2
  x^2       x + 1        [1, 1]       3
```

The table returned by the function `repr_table` 👉 lets us map the integer representation in the rightmost column to the polynomial and binary/vector representations in the middle columns with ease.

Rijndael and the AES use the Galois field GF(2^8) and the irreducible polynomial

$$m_{\text{AES}}(x) \triangleq x^8 + x^4 + x^3 + x + 1. \tag{2}$$

Details to follow.

2.2 Key schedule

Both the DES and AES leverage the strategies of *diffusion* and *confusion* [CMR06, p. 2]:

Diffusion aims to spread the influence of the key and plaintext to the ciphertext. This is done primarily through *permutations*.

Confusion aims to complicate the relationship among the key, plaintext and ciphertext. This is done primarily through *substitutions*.

The two strategies above ☝ can already be seen in the key schedule.

The AES, like most block ciphers, is an *iterated* cipher where the output is computed by applying in an iterative fashion a fixed key-dependent function — called a *round transformation* — N_r times to the input, where N_r is the number of rounds [vJ11, p. 152].

Figure 3 shows values of N_r for different values of block length and key length.

A *key scheduling* algorithm takes as input a user-selected key and produces a set of *round keys* [vJ11, p. 152], a subset of which is applied in each round transformation.

For the AES, key scheduling comprises 1️⃣ *key expansion* and 2️⃣ *round key selection* [DR20, Sec. 3.6].

Key expansion is implemented in the function `AES_set_encrypt_key` in the file `crypto/aes/aes_core.c`.

	N_b				
N_k	4	5	6	7	8
4	10	11	12	13	14
5	11	11	12	13	14
6	12	12	12	13	14
7	13	13	13	13	14
8	14	14	14	14	14

Figure 3: Number of rounds N_r as a function of block length N_b and key length N_k [DR20, Table 3.2]. Lengths are in multiples of a word (32 bits).

There are three versions of the function `AES_set_encrypt_key`: one textbook version and two optimised versions. The textbook version is:

Listing 1: `AES_set_encrypt_key`.

```
int AES_set_encrypt_key(const unsigned char *userKey, const int bits, AES_KEY *key)
{
  u64 *rk;

  if (!userKey || !key) return -1;
  if (bits ≠ 128 && bits ≠ 192 && bits ≠ 256) return -2;

  rk = (u64*)key→rd_key;

  if (bits == 128)
      key→rounds = 10;
  else if (bits == 192)
      key→rounds = 12;
  else
      key→rounds = 14;

  KeyExpansion(userKey, rk, key→rounds, bits/32);
  return 0;
}

typedef union { unsigned char b[8]; u32 w[2]; u64 d; } uni;

static void KeyExpansion(const unsigned char *key, u64 *w, int nr, int nk)
{
  u32 rcon; uni prev; u32 temp; int i, n;

  memcpy(w, key, nk*4); /* copy key to expanded key w */
  memcpy(&rcon, "\1\0\0\0", 4);
  n = nk/2;             /* n=2 for 128-bit keys */
  prev.d = w[n-1];      /* w[1] for 128-bit keys */
  for (i = n; i < (nr+1)*2; i++) {     /* i=2 to 22 for 128-bit keys */
    temp = prev.w[1];                  /* temp gets w[i-1] from prev */
    if (i % n == 0) {
      RotWord(&temp); SubWord(&temp); temp ^= rcon; XtimeWord(&rcon);
    } else if (nk > 6 && i % n == 2) { /* only for 256-bit keys */
      SubWord(&temp);
    }
    prev.d = w[i-n];
    prev.w[0] ^= temp;
    prev.w[1] ^= prev.w[0];
    w[i] = prev.d;
  }
```

```
}
```

⚠ Even this textbook version ☝ does not follow FIPS 197 [NIS01] precisely, let alone [DR20].

In Listing 1,

- `rk` is an array of $4(N_r + 1)$ words; see `include/openssl/aes.h`.
- `w` is a 64-bit pointer at `rk`. 🥴 This is the dizzying part: `w[i]` here is equivalent to `w[2*i]w[2*i+1]` in [NIS01, Sec. 5.2], but understandably this is done for efficiency.
- `nr` = number of rounds, N_r.
- `nk` = key length, N_k, in number of words; and `n` ≜ `nk/2`. 👉
- `rcon` is the round constant, the purpose of which is to eliminate symmetries [DR20, Sec. 3.6.1].

Key length	`nr`	`nk`	`n`	N_b
128 bits	10	4	2	4
192 bits	12	6	3	4
256 bits	14	8	4	4

For brevity, the following discussion focuses on the 128-bit (`nk` = 4) case.

Eq. (3) captures the essence of the `KeyExpansion` function in Listing 1, while Figure 4 illustrates the workflow.

$$w[i] = \begin{cases} \texttt{key}[i] & \text{if } i < n, \\ w[i-n] \oplus \texttt{SubWord}(\texttt{RotWord}(w[i-1])) \oplus \texttt{rcon}_{i/n} & \text{if } i \geq n \text{ and } i = 0 \bmod n, \\ w[i-n] \oplus w[i-1] & \text{otherwise.} \end{cases} \tag{3}$$

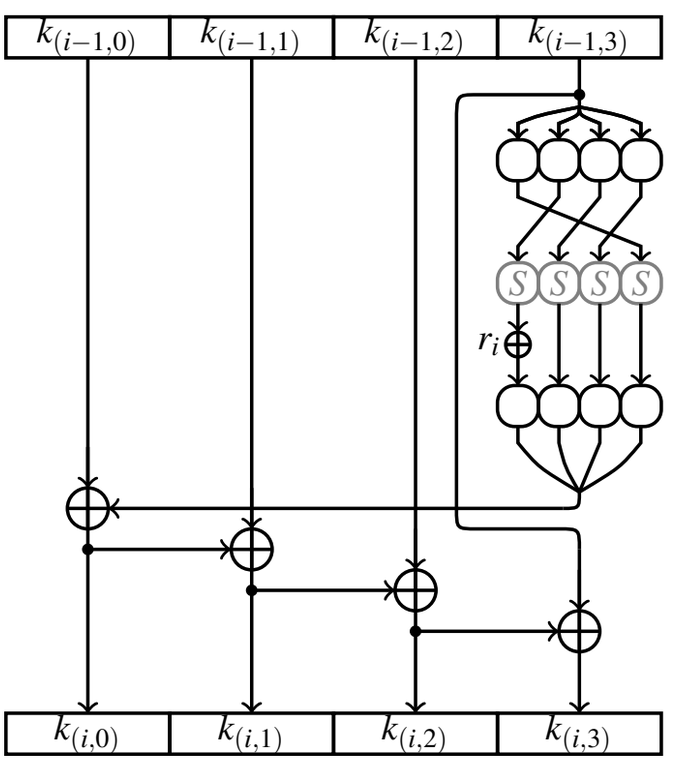


Figure 4: Key expansion for AES-128 [KR11, Fig. 3.1]: $k(0, 0{:}3) \to k(0{:}N_r, 0{:}3)$.

In Eq. (3) ☝,

- `w[0]` and `w[1]` (128 bits) are filled with the original key.
- Every following word, `w[i]`, is set to the XOR of the previous word, `w[i-1]`, and the word `n` positions earlier, `w[i-n]`.
- For words in positions that are a multiple of `n`, a transformation is applied to `w[i-1]` prior to the XOR, followed by an XOR with a round constant.
- The transformation steps, namely `RotWord` and `SubWord`, on `w[i-1]`, as well as the update step `XtimeWord` on the round constant are subsequently discussed.

👈 In Figure 4, 1️⃣ $k(i, j)k(i, j+1)$, where j is even, is equivalent to $w[i]$ in Eq. (3) and `w[i]` in Listing 1; 2️⃣ S represents `SubWord`; and 3️⃣ r_i is exactly $rc_{i/n}$ in Eq. (5).

The function `RotWord`

- Performs a byte-wise leftward rotation (cyclic shift) on a word [NIS01, Sec. 5.2], i.e., $b_0 b_1 b_2 b_3$ becomes $b_1 b_2 b_3 b_0$. 👉
- If the input word is represented as a fourth-order polynomial whose coefficients are the input bytes, then `RotWord` essentially multiplies the input polynomial with x^3 modulo $(x^4 + 1)$ [NIS01, Sec. 4.3].

Listing 2: `RotWord`.

```
static void RotWord(u32 *x)
{
  unsigned char *w0;
  unsigned char tmp;

  w0    = (unsigned char *)x;
  tmp   = w0[0];
  w0[0] = w0[1];
  w0[1] = w0[2];
  w0[2] = w0[3];
  w0[3] = tmp;
}
```

The function `SubWord`

- Substitutes each of the four bytes in a word with its affine transformation in $\mathrm{GF}(2^8)$.
- If a byte is treated as a pair of nibbles xy (i.e., 4-bit x and 4-bit y), then the affine transformation takes the form [DR20, (3.9)]:

$$\mathrm{Aff}_8(\mathrm{Inv}_8(xy)) = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x_3 \\ x_2 \\ x_1 \\ x_0 \\ y_3 \\ y_2 \\ y_1 \\ y_0 \end{bmatrix}^{-1} + \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}. \tag{4}$$

  Inversion can be done by the extended Euclidean algorithm, which was covered in an earlier tutorial, but we shall revisit it via the `galois` Python package in the current tutorial.

  The Galois-field arithmetic in Eq. (4) ☝ follows the description in Sec. 2.1, i.e., while addition is XOR, multiplication is done modulo the irreducible polynomial $m_{\mathrm{AES}}(x)$ in (2).
- Since there are only 256 possible output values from Eq. (4), the lookup table or so-called *S-box* (short for "substitution box") in Table 2 can be used. In the tutorial, we shall learn how to obtain Table 2 using the `galois` Python package.
- Eq. (4) is the only *nonlinear* transformation among the building blocks of the AES [DR20, Sec. 3.4.1]. 👈 It keeps the maximum input-output correlation small to help resist linear and differential cryptanalyses.
- The OpenSSL code for `SubWord` is too lengthy to be included here.

		y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
x	0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
	1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
	2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
	3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
	4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
	5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
	6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
	7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
	8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
	9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
	A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
	B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
	C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
	D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
	E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
	F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Table 2: Practical means for implementing Eq. (4): S-box for looking up output values $\mathrm{Aff}_8(\mathrm{Inv}_8(xy))$ in hexadecimals corresponding to input nibbles x and y [DR20, Table A.1]. Same table as [NIS01, Figure 7].

The function `XtimeWord` updates the 32-bit round constant by recursion:

$$\mathtt{rcon}_{i/n} = [rc_{i/n}, 00_{16}, 00_{16}, 00_{16}], \tag{5a}$$

$$rc_{i/n} = \begin{cases} 1 & \text{if } i = 1, \\ 02_{16} \times rc_{(i-1)/n} & \text{if } i > 1 \text{ and } rc_{(i-1)/n} < 80_{16}, \\ 02_{16} \times rc_{(i-1)/n} \oplus 011B_{16} & \text{if } i > 1 \text{ and } rc_{(i-1)/n} \geq 80_{16}, \end{cases} \tag{5b}$$

$$\Updownarrow$$

$$rc_{i/n} = x^{i/n-1}. \tag{5c}$$

- Eqs. (5b) and (5c) are equivalent but the former uses the hexadecimal representation while the latter uses the polynomial representation; recall 02_{16} = polynomial x in GF(2^8).
- The label `xtime` refers to multiplication by polynomial x in GF(2^8) [DR20, Sec. 4.1.1].

  Suppose $b(x) = \sum_{i=0}^{7} b_i x^i$, then

$$
\begin{aligned}
& x \times b(x) \bmod m_{\mathrm{AES}}(x) = \sum_{i=0}^{7} b_i x^{i+1} \bmod m_{\mathrm{AES}}(x) \\
&= b_7(x^8 + x^4 + x^3 + x + 1) + b_6 x^7 + b_5 x^6 + b_4 x^5 \\
&\quad + (b_7 + b_3)x^4 + (b_7 + b_2)x^3 + b_1 x^2 + (b_7 + b_0)x \\
&\quad + b_7 \bmod m_{\mathrm{AES}}(x) \\
&= \underbrace{\sum_{i=0}^{6} b_i x^{i+1}}_{\text{Left shift of } b_6 b_5 \cdots b_0} + \underbrace{b_7(x^4 + x^3 + x + 1)}_{\begin{cases} 1B_{16} & \text{if } b_7 = 1 \\ 00_{16} & \text{otherwise} \end{cases}}.
\end{aligned}
\tag{6}
$$

  Thus, `xtime` can be implemented with a left shift and a conditional XOR, consistent with Eq. (5b) and Listing 3.

Listing 3: `XtimeWord`.

```
static void XtimeWord(u32 *w)
{
  u32 a, b;

  a = *w;
  b = a & 0x80808080u;
  a ^= b;
  b -= b >> 7;
  b &= 0x1B1B1B1Bu;
  b ^= a << 1;
  *w = b;
}
```

Constant-time code ☝ avoids `if-then-else` to prevent timing attacks [DR20, Sec. 4.1.1].

Computational efficiency can be improved using a look-up table [DR20, Table A.6] at the expense of memory and storage.

Finally, in terms of key selection, the first four words of the expanded key are used for round zero, the next four words are used for round one, and so on. 👈 This explains why `rk` in Listing 1 is an array of $4(N_r + 1)$ words.

Quiz 4

Referring to `crypto/aes/aes_core.c`, what is the difference between `AES_set_decrypt_key` and `AES_set_encrypt_key`?

2.3 Encryption/decryption

Let us begin by inspecting the OpenSSL file `crypto/aes/aes_ecb.c`, which suggests the function `AES_ecb_encrypt` is used for encryption and decryption, depending on the value of the integer parameter `enc`.

- For encryption, the function `AES_encrypt` defined in `crypto/aes/aes_core.c` is called.
- For decryption, the function `AES_decrypt` defined in `crypto/aes/aes_core.c` is called. The decryption process reverses the encryption process, and is not discussed here.

In `crypto/aes/aes_core.c`, there are two versions of `AES_encrypt`: a textbook version and an optimised version. Listing 4 shows the textbook version, where `Cipher` matches the pseudocode in [NIS01, Figure 5] closely.

Listing 4: `AES_encrypt` and `Cipher`.

```
void AES_encrypt(const unsigned char *in, unsigned char *out,
                 const AES_KEY *key)
{
```

```
  const u64 *rk;

  assert(in && out && key);
  rk = (u64*)key->rd_key;

  Cipher(in, out, rk, key->rounds);
}

static void Cipher(const unsigned char *in, unsigned char *out,
                   const u64 *w, int nr)
{
  u64 state[2];
  int i;

  memcpy(state, in, 16);

  AddRoundKey(state, w);
  for (i = 1; i < nr; i++) {
      SubLong(&state[0]); SubLong(&state[1]);
      ShiftRows(state); MixColumns(state); AddRoundKey(state, w + i*2);
  }
  SubLong(&state[0]); SubLong(&state[1]);
  ShiftRows(state); AddRoundKey(state, w + nr*2);

  memcpy(out, state, 16);
}
```

In Listing 4 ☝,

- The 128-bit `state` is the column-first flattened version of the 4×4 state-byte array specified in [NIS01, Sec. 3.4]. More on this later.
- `state` is initialised to the plaintext.
- In round zero, the function `AddRoundKey`, as implemented in Listing 5 ☝, XORs the current state with the current round key.
- The first to penultimate rounds involve `SubLong`, `ShiftRows` and `MixColumns` — to be subsequently discussed — prior to the application of `AddRoundKey`.
- The last round is the same as the preceding rounds, except it does not involve `MixColumns`.
- Figure 5 illustrates the encryption process.

Listing 5: `AddRoundKey`.

```
static void AddRoundKey(u64 *state,
                        const u64 *w)
{
  state[0] ^= w[0]; state[1] ^= w[1];
}
```

The function `SubLong` implements `SubBytes` in FIPS 197 [NIS01]:

- Same as the function `SubWord` in Listing 1, except `SubLong` operates on four words rather than one word at a time.
- The OpenSSL code for `SubLong` is too lengthy to be included here.

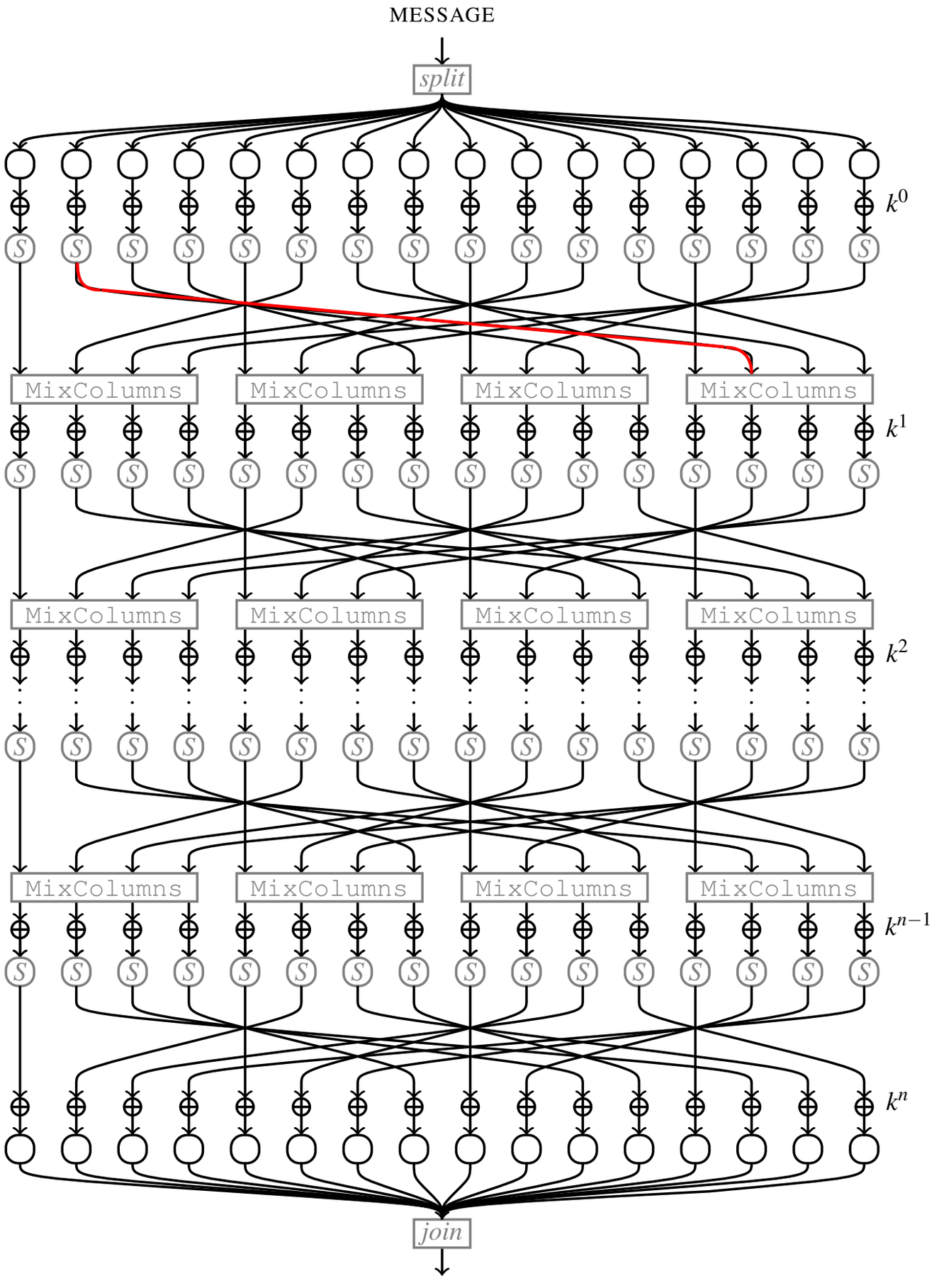


Figure 5: AES encryption [KR11, Fig. 3.4]: Each arrow is one byte wide. k^i denotes the subkey for round i. S is short for `SubLong` and `ShiftRows`. $n = N_r$ is number of rounds and should not be confused with n in Listing 1. Red line highlights a sample effect of `ShiftRows`: 2nd byte of the state shifted to the 14th byte of the state.

The function `ShiftRows` 👉

- Treats the state array, s, as a 4×4 matrix and shifts its rows as follows:

s_0	s_4	s_8	s_{12}
s_1	s_5	s_9	s_{13}
s_2	s_6	s_{10}	s_{14}
s_3	s_7	s_{11}	s_{15}

↓

s_0	s_4	s_8	s_{12}
s_5	s_9	s_{13}	s_1
s_{10}	s_{14}	s_2	s_6
s_{15}	s_3	s_7	s_{11}

- Together with `MixColumn`, achieves diffusion [KR11, Sec. 3.1.5.2].

Listing 6: `ShiftRows`.

```
static void ShiftRows(u64 *state)
{
  unsigned char s[4];
  unsigned char *s0;
  int r;

  s0 = (unsigned char *)state;
  for (r = 0; r < 4; r++) {
    s[0] = s0[0*4 + r];
    s[1] = s0[1*4 + r];
    s[2] = s0[2*4 + r];
    s[3] = s0[3*4 + r];
    s0[0*4 + r] = s[(r+0) % 4];
    s0[1*4 + r] = s[(r+1) % 4];
    s0[2*4 + r] = s[(r+2) % 4];
    s0[3*4 + r] = s[(r+3) % 4];
  }
}
```

The function `MixColumns` (see Listing 7)

- Does *not* mix columns together.
- But treats the state array, s, as a 4×4 matrix, and each jth column of the matrix as a four-term polynomial in GF(2^8):

$$s_{0,j}x^3 + s_{1,j}x^2 + s_{2,j}x + s_{3,j}, \qquad j = 1, \ldots, 4, \tag{7}$$

with irreducible polynomial $x^4 + 1$. ☝ This is different from the treatment in Sec. 2.1.

Each column is multiplied by the constant polynomial [NIS01, Sec. 5.1.3]:

$$a(x) \triangleq 03_{16}x^3 + 01_{16}x^2 + 01_{16}x + 02_{16}, \tag{8}$$

which was chosen because it has an inverse (making decryption possible). Specifically, the inverse of $a(x)$ is $a(x)^3$ [DR20, (3.13)].

Without derivation, multiplying the polynomial in (7) with $a(x)$ in (8) is equivalent to the linear transformation:

$$\begin{bmatrix} s_{0,j} \\ s_{1,j} \\ s_{2,j} \\ s_{3,j} \end{bmatrix} \leftarrow \begin{bmatrix} 02_{16} & 03_{16} & 01_{16} & 01_{16} \\ 01_{16} & 02_{16} & 03_{16} & 01_{16} \\ 01_{16} & 01_{16} & 02_{16} & 03_{16} \\ 03_{16} & 01_{16} & 01_{16} & 02_{16} \end{bmatrix} \begin{bmatrix} s_{0,j} \\ s_{1,j} \\ s_{2,j} \\ s_{3,j} \end{bmatrix} \tag{9}$$

and equivalently,

$$s_{0,j} \leftarrow (02_{16} \times s_{0,j}) \oplus (03_{16} \times s_{1,j}) \oplus s_{2,j} \oplus s_{3,j}, \tag{10a}$$
$$s_{1,j} \leftarrow s_{0,j} \oplus (02_{16} \times s_{1,j}) \oplus (03_{16} \times s_{2,j}) \oplus s_{3,j}, \tag{10b}$$
$$s_{2,j} \leftarrow s_{0,j} \oplus s_{1,j} \oplus (02_{16} \times s_{2,j}) \oplus (03_{16} \times s_{3,j}), \tag{10c}$$
$$s_{3,j} \leftarrow (03_{16} \times s_{0,j}) \oplus s_{1,j} \oplus s_{2,j} \oplus (02_{16} \times s_{3,j}). \tag{10d}$$

- The matrix in Eq. (9) is called a *D-box* (short for "diffusion box"). Implementation of Eq. (10) can be seen in Listing 7.

Listing 7: `MixColumns`. Compare `XtimeLong` with `XtimeWord` in Listing 3.

```
typedef union { unsigned char b[8]; u32 w[2]; u64 d; } uni;

static void XtimeLong(u64 *w)
{
  u64 a, b;

  a = *w;
  b = a & U64(0x8080808080808080);
  a ^= b;
  b -= b >> 7;
  b &= U64(0x1B1B1B1B1B1B1B1B);
  b ^= a << 1;
  *w = b;
}

static void MixColumns(u64 *state)
{
  uni s1; uni s; int c;

  for (c = 0; c < 2; c++) { /* works on 2 cols at once */
    s1.d = state[c];        /* 2 cols x 4 bytes/col */
    s.d = s1.d;
    s.d ^=((s.d & U64(0xFFFF0000FFFF0000))>>16) | ((s.d & U64(0x0000FFFF0000FFFF))<<16);
    /* result: s =
       s0 + s2
       s1 + s3
       s2 + s0
       s3 + s1 */
    s.d ^=((s.d & U64(0xFF00FF00FF00FF00))>> 8) | ((s.d & U64(0x00FF00FF00FF00FF))<< 8);
    /* result: s =
       s0 + s2 + s1 + s3
       s1 + s3 + s0 + s2
       s2 + s0 + s3 + s1
       s3 + s1 + s2 + s0 */
    s.d ^= s1.d;
    /* result: s =
       s0 + s2 + s1 + s3 + s0 = s2 + s1 + s3
       s1 + s3 + s0 + s2 + s1 = s3 + s0 + s2
       s2 + s0 + s3 + s1 + s2 = s0 + s3 + s1
       s3 + s1 + s2 + s0 + s3 = s1 + s2 + s0 */
    XtimeLong(&s1.d);
    s.d ^= s1.d;
    /* result: s =
       s2 + s1 + s3 + 2*s0
       s3 + s0 + s2 + 2*s1
       s0 + s3 + s1 + 2*s2
       s1 + s2 + s0 + 2*s3 */
    s.b[0] ^= s1.b[1]; s.b[4] ^= s1.b[5];
    s.b[1] ^= s1.b[2]; s.b[5] ^= s1.b[6];
    s.b[2] ^= s1.b[3]; s.b[6] ^= s1.b[7];
    s.b[3] ^= s1.b[0]; s.b[7] ^= s1.b[4];
    /* result: s =
       s2 + s1 + s3 + 2*s0 + 2*s1
       s3 + s0 + s2 + 2*s1 + 2*s2
       s0 + s3 + s1 + 2*s2 + 2*s3
       s1 + s2 + s0 + 2*s3 + 2*s0 */
    state[c] = s.d;
  }
}
```

Listing 8 shows a speed-optimised but "rolled-up" version of `AES_encrypt` using lookup

tables:

- The arrays `Te0`, `Te1`, `Te2` and `Te3` capture simultaneously the results of (4) and `MixColumns`.
- `ShiftRows` is accomplished using shifting, e.g., `t0` stores the transformed version of 1️⃣ the first element of `s0`, 2️⃣ the second element of `s1`, 3️⃣ the third element of `s2`, and 4️⃣ the last element of `s3`.

Listing 8: Speed-optimised `AES_encrypt` using lookup tables.

```
void AES_encrypt(const unsigned char *in, unsigned char *out,
                 const AES_KEY *key)
{
  const u32 *rk;
  u32 s0, s1, s2, s3, t0, t1, t2, t3;

  int r;

  assert(in && out && key);
  rk = key→rd_key;

  /* map byte array block to cipher state
   * and add initial round key: */
  s0 = GETU32(in     ) ^ rk[0]; /* 1st column */
  s1 = GETU32(in +  4) ^ rk[1]; /* 2nd column */
  s2 = GETU32(in +  8) ^ rk[2]; /* 3rd column */
  s3 = GETU32(in + 12) ^ rk[3]; /* 4th column */

  /* Nr - 1 full rounds: */
  r = key→rounds >> 1; /* because one iteration below = two rounds */
  for (;;) {
    t0 =Te0[(s0 >> 24)       ] ^ Te1[(s1 >> 16) & 0xff] ^
        Te2[(s2 >>  8) & 0xff] ^ Te3[(s3      ) & 0xff] ^ rk[4];
    t1 =Te0[(s1 >> 24)       ] ^ Te1[(s2 >> 16) & 0xff] ^
        Te2[(s3 >>  8) & 0xff] ^ Te3[(s0      ) & 0xff] ^ rk[5];
    t2 =Te0[(s2 >> 24)       ] ^ Te1[(s3 >> 16) & 0xff] ^
        Te2[(s0 >>  8) & 0xff] ^ Te3[(s1      ) & 0xff] ^ rk[6];
    t3 =Te0[(s3 >> 24)       ] ^ Te1[(s0 >> 16) & 0xff] ^
        Te2[(s1 >>  8) & 0xff] ^ Te3[(s2      ) & 0xff] ^ rk[7];

    rk += 8;
    if (--r == 0) { break; }

    s0 =Te0[(t0 >> 24)       ] ^ Te1[(t1 >> 16) & 0xff] ^
        Te2[(t2 >>  8) & 0xff] ^ Te3[(t3      ) & 0xff] ^ rk[0];
    s1 =Te0[(t1 >> 24)       ] ^ Te1[(t2 >> 16) & 0xff] ^
        Te2[(t3 >>  8) & 0xff] ^ Te3[(t0      ) & 0xff] ^ rk[1];
    s2 =Te0[(t2 >> 24)       ] ^ Te1[(t3 >> 16) & 0xff] ^
        Te2[(t0 >>  8) & 0xff] ^ Te3[(t1      ) & 0xff] ^ rk[2];
    s3 =Te0[(t3 >> 24)       ] ^ Te1[(t0 >> 16) & 0xff] ^
        Te2[(t1 >>  8) & 0xff] ^ Te3[(t2      ) & 0xff] ^ rk[3];
  }

  /* apply last round and
   * map cipher state to byte array block: */
  s0 =(Te2[(t0 >> 24)       ] & 0xff000000) ^ (Te3[(t1 >> 16) & 0xff] & 0x00ff0000) ^
      (Te0[(t2 >>  8) & 0xff] & 0x0000ff00) ^ (Te1[(t3      ) & 0xff] & 0x000000ff) ^
      rk[0];
  PUTU32(out     , s0);
  s1 =(Te2[(t1 >> 24)       ] & 0xff000000) ^ (Te3[(t2 >> 16) & 0xff] & 0x00ff0000) ^
      (Te0[(t3 >>  8) & 0xff] & 0x0000ff00) ^ (Te1[(t0      ) & 0xff] & 0x000000ff) ^
      rk[1];
```

```
  PUTU32(out +  4, s1);
  s2 =(Te2[(t2 >> 24)       ] & 0xff000000) ^ (Te3[(t3 >> 16) & 0xff] & 0x00ff0000) ^
      (Te0[(t0 >>  8) & 0xff] & 0x0000ff00) ^ (Te1[(t1      ) & 0xff] & 0x000000ff) ^
      rk[2];
  PUTU32(out +  8, s2);
  s3 =(Te2[(t3 >> 24)       ] & 0xff000000) ^ (Te3[(t0 >> 16) & 0xff] & 0x00ff0000) ^
      (Te0[(t1 >>  8) & 0xff] & 0x0000ff00) ^ (Te1[(t2      ) & 0xff] & 0x000000ff) ^
      rk[3];
  PUTU32(out + 12, s3);
}
```

Quiz 5

Referring to `crypto/aes/aes_core.c`, how many bytes of storage are needed for `Te0`, `Te1`, `Te2` and `Te3`? Compare this with the size of L1 data cache on an ARM Cortex-A53, a widely used microprocessor.

3 References

[Aum18] J.-P. Aumasson, *Serious Cryptography: A Practical Introduction to Modern Encryption*, No Starch Press, 2018.

[Bar20] E. Barker, *Guideline for Using Cryptographic Standards in the Federal Government: Cryptographic Mechanisms*, Special Publication 800-175B Revision 1, NIST, 2020. https://doi.org/10.6028/NIST.SP.800-175Br1.

[BR19] E. Barker and A. Roginsky, Transitioning the use of cryptographic algorithms and key lengths, NIST Special Publication 800-131A Revision 2, 2019. https://doi.org/10.6028/NIST.SP.800-131Ar2.

[CMR06] C. Cid, S. Murphy, and M. Robshaw, *Algebraic Aspects of the Advanced Encryption Standard*, Springer New York, NY, 2006. https://doi.org/10.1007/978-0-387-36842-9.

[CRY22] CRYPTREC, 電子政府における調達のために参照すべき暗号のリスト（cryptrec 暗号リスト）, LS-0001-2012R7, 2022, created 2013, updated 2022. Available at https://www.cryptrec.go.jp/list/cryptrec-ls-0001-2012r7.pdf.

[DR20] J. Daemen and V. Rijmen, *The Design of Rijndael: The Advanced Encryption Standard (AES)*, 2nd ed., *Information Security and Cryptography*, Springer-Verlag, Heidelberg, 2020. https://doi.org/10.1007/978-3-662-60769-5.

[Gol01] O. Goldreich, *Foundations of Cryptography: Volume I Basic Tools*, Cambridge University Press, 2001. https://doi.org/10.1017/CBO9780511546891.

[Gol04] O. Goldreich, *Foundations of Cryptography: Volume II Basic Applications*, Cambridge University Press, 2004. https://doi.org/10.1017/CBO9780511721656.

[Gue09] S. Gueron, Intel's new AES instructions for enhanced performance and security, in *Fast Software Encryption* (O. Dunkelman, ed.), Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 51–66.

[HW03] D. W. Hardy and C. L. Walker, *Applied Algebra: Codes, Ciphers and Discrete Algorithms*, Pearson Education, Inc., 2003.

[KL21] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*, 3rd ed., CRC Press, 2021. Available at https://ebookcentral.proquest.com/lib/unisa/detail.action?docID=6425020.

[Kib17] M. R. Kibler, *Galois Fields and Galois Rings Made Easy*, ISTE Press Ltd, 2017.

[KR11] L. R. Knudsen and M. J. Robshaw, *The Block Cipher Companion*, *Information Security and Cryptography Texts and Monographs*, Springer Berlin, Heidelberg, 2011. https://doi.org/10.1007/978-3-642-17342-4.

[LN94] R. Lidl and H. Niederreiter, *Introduction to finite fields and their applications*, revised ed., Cambridge University Press, 1994.

[Lov22] S. Lovett, *Abstract Algebra: A First Course*, 2nd ed., Chapman and Hall/CRC, 2022. https://doi.org/10.1201/9781003299233.

[McN99] D. McNett, US government's encryption standard broken in less than a day, formal press release, 1999. Available at http://www.distributed.net/images/d/d7/19990119_-_PR_-_release-des3.pdf.

[Mou21] N. Mouha, Review of the Advanced Encryption Standard, NISTIR 8319, 2021. https://doi.org/10.6028/NIST.IR.8319.

[NIS01] NIST, Announcing the ADVANCED ENCRYPTION STANDARD (AES), Federal Information Processing Standards Publication 197, November 2001. https://doi.org/10.6028/NIST.FIPS.197.

[NIS21] NIST, Cryptographic Standards and Guidelines Share to Facebook: AES Development, 2021, Created 29 Dec 2016, updated 23 Aug 2021. Available at https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/archived-crypto-projects/aes-development.

[PBO+03] B. Preneel, A. Biryukov, E. Oswald, B. V. Rompay, L. Granboulan, E. Dottax, S. Murphy, A. Dent, J. White, M. Dichtl, S. Pyka, M. Schafheutle, P. Serf, E. Biham, E. Barkan, O. Dunkelman, J.-J. Quisquater, M. Ciet, F. Sica, L. Knudsen, M. Parker, and H. Raddum, *NESSIE Security Report*, Deliverable D20, NESSIE Consortium, February 2003, Version 2.0.

[Pre03] B. Preneel, NESSIE project announces final selection of crypto algorithms: An open competition for the crypto algorithms of the 21st century, Press Release, 2003. Available at https://www.cosic.esat.kuleuven.be/nessie/deliverables/press_release_feb27.pdf.

[Sch15] B. Schneier, *Applied Cryptography, Second Edition: Protocols, Algorithms, and Source Code in C*, 20th anniversary edition ed., Wiley, 2015. Available at https://learning.oreilly.com/library/view/applied-cryptography-protocols/9781119096726.

[vJ11] H. C. van Tilborg and S. Jajodia (eds.), *Encyclopedia of Cryptography and Security*, Springer, Boston, MA, 2011. https://doi.org/10.1007/978-1-4419-5906-5.