CSCI-GA.3205 Applied Cryptography & Network Security

Department of Computer Science
New York University

PRESENTED BY DR. MAZDAK ZAMANI
mazdak.zamani@NYU.edu

Stream Cipher
Block Cipher

03

NYU

Stream ciphers

- *Stream cipher* is one type of cipher that allows us to build secure ciphers (semantic security) that use reasonably short keys.

Pseudo-random generators

Recall the one-time pad. Here, keys, messages, and ciphertexts are all L-bit strings. However, we would like to use a key that is much shorter. So the idea is to instead use a short, l-bit "seed" s as the encryption key, where l is much smaller than L, and to "stretch" this seed into a longer, L-bit string that is used to mask the message (and unmask the ciphertext). The string s is stretched using some efficient, deterministic algorithm G that maps (l -bit strings to L-bit strings. Thus, the key space for this modified one-time pad is $\{0,1\}^{l,}$ while the message and ciphertext spaces are $\{0,1\}^{L}$. For $s \in \{0,1\}^{l}$ and $m, c \in \{0,1\}^{L}$, encryption and decryption are defined as follows:

$$E(s, m) := G(s) \oplus m \text{ and } D(s, c):=G(s) \oplus c$$

This modified one-time pad is called a **stream cipher**, and the function G is called **a pseudo-random generator**.

Pseudo-random generators – con'd

- If $l < L$, then by Shannon's Theorem, this stream cipher cannot achieve perfect security; however, if G satisfies an appropriate security property, then this cipher is semantically secure.
- Suppose s is a random l-bit string and r is a random L-bit string.
- Intuitively, if an adversary cannot effectively tell the difference between G(s) and r, then he should not be able to tell the difference between this stream cipher and a one-time pad.

Statistical test

- An algorithm that is used to distinguish a pseudo-random string G(s) from a truly random string r is called a statistical test.
- It takes a string as input, and outputs 0 or 1.
- Such a test is called effective if the probability that it outputs 1 on a pseudo-random input is significantly different than the probability that it outputs 1 on a truly random input.

Designing an effective statistical test

- Given an L-bit string, calculate some statistic, and then see if this statistic differs greatly from what one would expect if the string were truly random.

Example of an effective statistical test

For example, a very simple statistic that is easy to compute is the number k of 1's appearing in the string. For a truly random string, we would expect $k \approx L/2$. If the PRG G had some bias towards either 0-bits or 1-bits, we could effectively detect this with a statistical test that, say, outputs 1 if $|k - 0.5L| < 0.01L$, and otherwise outputs 0. This statistical test would be quite effective if the PRG G did indeed have some significant bias towards either 0 or 1.

Definition of a pseudo-random generator

A pseudo-random generator, or PRG for short, is an efficient, deterministic algorithm G that, given as input a seed s, computes an output r.
The seed s comes from a finite seed space S and the output r belongs to a finite output space R.
Typically, S and R are sets of bit strings of some prescribed length (for example, in the discussion above, we had $S = \{0,1\}^l$ and $R = \{0,1\}^L$).
We say that G is a PRG defined over (S, R).
Our definition of security for a PRG captures the intuitive notion that if s is chosen at random from S and r is chosen at random from R, then no efficient adversary can effectively tell the difference between G(s) and r: the two are computationally indistinguishable.
The definition is formulated as an attack game.

Attack Game

For a given PRG G, defined over (S, R), and for a given adversary A, we define two experiments, Experiment 0 and Experiment 1. For b =0,1, we define:

Experiment b:

- if b = 0: $s \xleftarrow{R} S$, $r \leftarrow G(s)$;
- if b = 1: $r \xleftarrow{R} R$.

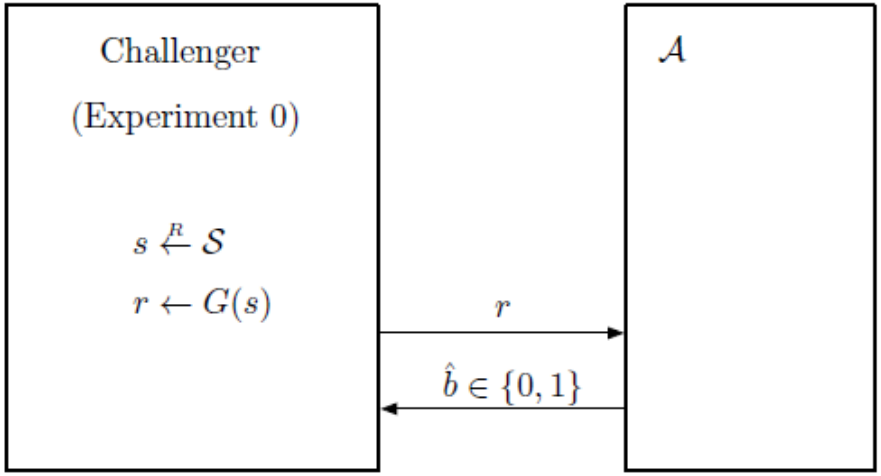
and sends r to the adversary.

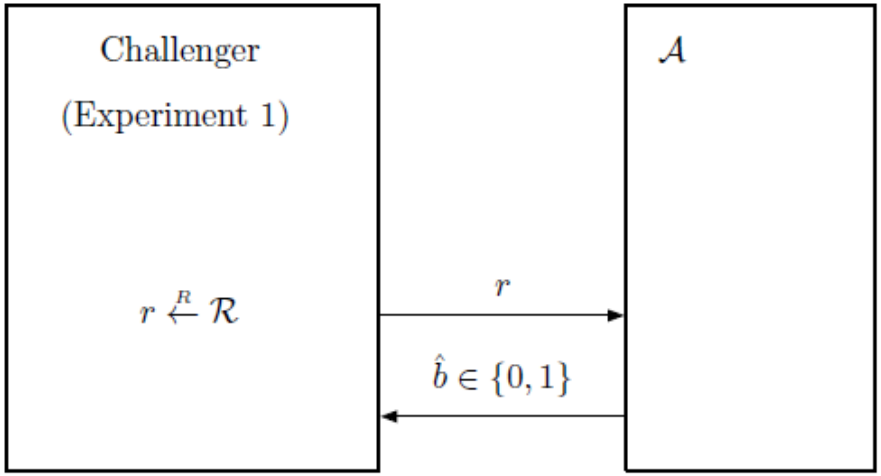
Given r, the adversary computes and outputs a bit $\hat{b} \in \{0,1\}$.

For b = 0, 1, let W_b be the event that A outputs 1 in Experiment b. We define A's advantage with respect to G as

$$PRGadv[A,G] := |Pr[W_0] - Pr[W_1]|$$

Experiments 0 and 1 of Attack Game 3.1





Stream ciphers: encryption with a PRG

Let G be a PRG defined over $(\{0,1\}^{l}, \{0,1\}^{L})$ that is, G stretches an l-bit seed to an L-bit output. The **stream cipher** $\varepsilon = (E, D)$ **constructed from G** is defined over $(\{0,1\}^{l}, \{0,1\}^{\leq L}, \{0,1\}^{\leq L})$ for $s \in \{0,1\}^{l}$ and $m,c \in \{0,1\}^{\leq L}$, encryption and decryption are defined as follows: if $|m| = v$, then

$$E(s, m) := G(s)[0 \ldots v\text{-}1] \oplus m,$$

and if $|c| = v$, then

$$D(s,c) = G(s)[0..v\text{-}1] \oplus c.$$

NYU

Stream cipher limitations: attacks on the one-time pad

- Although stream ciphers are semantically secure, they are highly brittle and become totally insecure if used incorrectly.
  - The two-time pad is insecure
  - The one-time pad is malleable

The two-time pad is insecure

A stream cipher is well equipped to encrypt a single message from Alice to Bob. Alice, however, may wish to send several messages to Bob. For simplicity suppose Alice wishes to encrypt two messages m_1 and m_2. The naive solution is to encrypt both messages using the same stream cipher key s:

$$c_1 \leftarrow m_1 \oplus G(s) \text{ and } c_2 \leftarrow m_2 \oplus G(s) \quad (3.8)$$

A moment's reflection shows that this construction is insecure in a very strong sense. An adversary who intercepts c_1 and c_2 can compute

$$\Delta := c_1 \oplus c_2 = (m_1 \oplus G(s)) \oplus (m_2 \oplus G(s)) = m_1 \oplus m_2$$

and obtain the XOR of m_1 and m_2. Not surprisingly, English text contains enough redundancy that given $\Delta = m_1 \oplus m_2$ the adversary can recover both m_1 and m_2 in the clear. Hence, the construction in (3.8) leaks the plaintexts after seeing only two sufficiently long ciphertexts.

The one-time pad is malleable

Although semantic security ensures that an adversity cannot read the plaintext, it provides no guarantee for integrity. When using a stream cipher an adversary can change the ciphertext and the modification will never be detected by the decryptor. Even worse, let us know that by changing the ciphertext, the attacker can control how the decrypted plaintext will change.

Suppose and attacker intercepts a ciphertext $c := E(s,m) = m \oplus G(s)$.
The attacker changes c to $c' := c \oplus \Delta$ for some Δ of the attacker's choice.
Consequently, the decryptor receives the modified message

$$D(s, c') = c' \oplus G(s) = (c \oplus \Delta) \oplus G(s) = m \oplus \Delta$$

Linear-Feedback Shift Registers

- The linear-feedback shift registers (LFSRs) have been used historically for pseudorandom-number generation, as they are extremely efficient to implement in hardware, and generate output having good statistical properties.
- By themselves, however, they do not give cryptographically strong pseudorandom generators, and in fact we will show an easy key recovery attack on LFSRs. Nevertheless, LFSRs can be used as a component in building stream ciphers with better security.

A linear-feedback shift register

Linear feedback shift registers (LFSR). The CSS stream cipher is built from two LFSRs.
An n-bit LFSR is defined by a set of integers $V = \{v_1, \ldots, v_d\}$ where each v_i is in the range $\{0, \ldots, n-1\}$. The elements of V are called **tap positions**. An LFSR gives a PRG as follows:

Input: $s = (b_{n-1}, \ldots, b_0) \in \{0,1\}^n$ *and* $s \neq 0^n$

Output: $y \in \{0,1\}^l$ where $l > n$

for i← 1... l do

Output b_0 //output one bit

$b \leftarrow b_{v1} \oplus \ldots \oplus b_{vd}$ //compute feedback bit

$s \leftarrow (b, b_{n-1}, \ldots, b_1)$ //shift register bits to the right

A linear-feedback shift register

The LFSR outputs one bit per clock cycle. Note that if an LFSR is started in state $s = 0^n$ then its output is degenerate, namely all 0. For this reason, one of the seed bits is always set to 1.

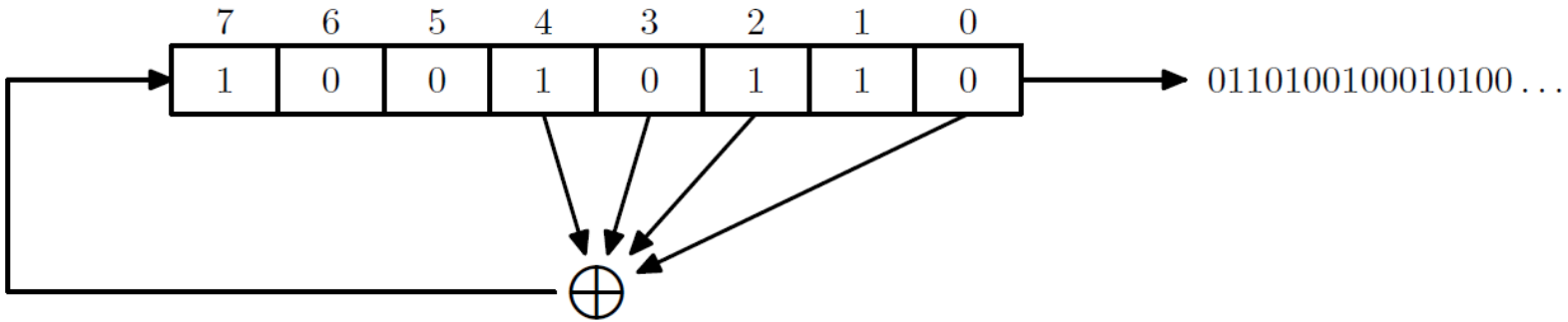


Figure 3.10: The 8 bit linear feedback shift register $\{4, 3, 2, 0\}$

Block ciphers

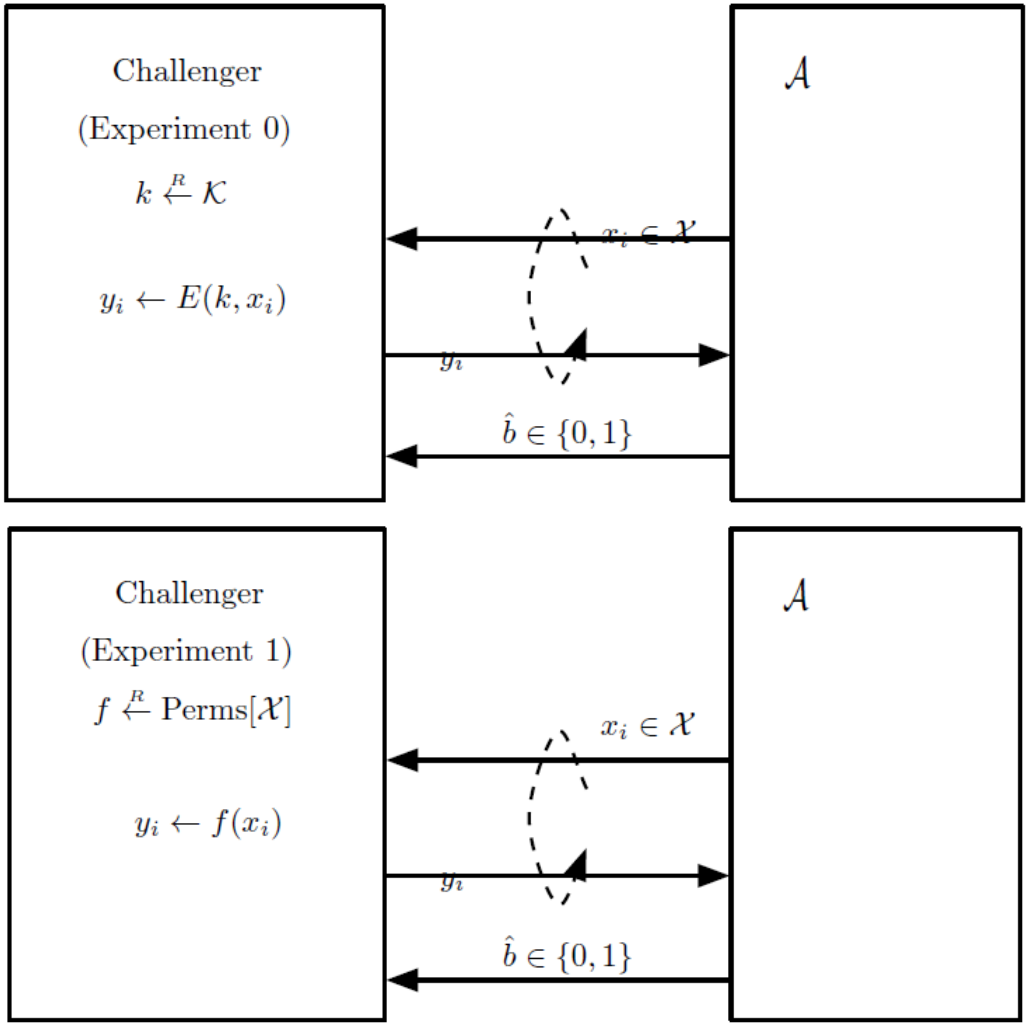
Functionally, a block cipher is a deterministic cipher ε =(E,D) whose message space and ciphertext space are the same (finite) set X.

If the key space of ε is K, we say that ε is a block cipher defined over (K, X). We call an element x ∈ X a data block and refer to X as the data block space of ε.

For every fixed key k ∈ K, we can define the function $f_k := E(k,.)$; that is, f_k: X → X sends x ∈ X to E(k, x) ∈ X.

The usual correctness requirement for any cipher implies that for every fixed key k, the function f_k is one-to-one, and as X is finite, f_k must be onto as well. Thus, f_k is a permutation on X, and D(k,.) is the inverse permutation f_k^{-1}.

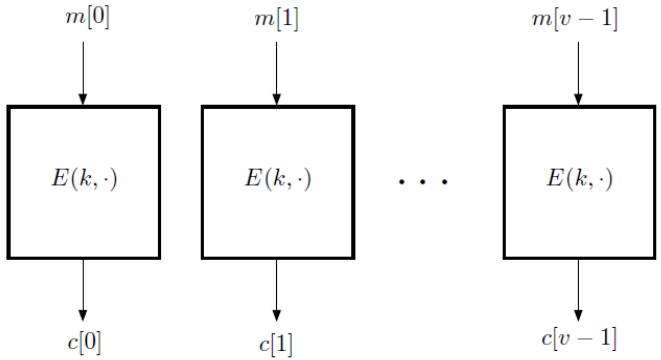
A secure block cipher is unpredictable

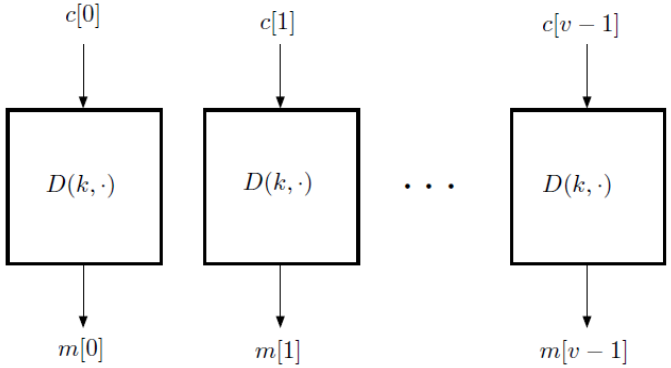


NYU

Electronic codebook mode

- If we want to encrypt longer messages, a natural idea would be to break up a long message into a sequence of data blocks and encrypt each data block separately.
- This use of a block cipher to encrypt long messages is called Electronic Codebook Mode, or ECB mode for short.





Pseudo Random Permutation

- Pseudo Random Function (**PRF**) defined over (K,X,Y):

  $F: K \times X \rightarrow Y$

  such that exists "efficient" algorithm to evaluate F(k,x)
- Pseudo Random Permutation (**PRP**) defined over (K,X):

  $E: K \times X \rightarrow X$

such that:

1. Exists "efficient" deterministic algorithm to evaluate E(k,x)
2. The function $E(k, \cdot)$ is one-to-one
3. Exists "efficient" inversion algorithm D(k,y)

Constructing block ciphers in practice

Block ciphers are a basic primitive in cryptography from which many other systems are built.
Virtually all block ciphers used in practice use the same basic framework called the **iterated cipher paradigm**.

To construct an iterated block cipher the designer makes two choices:

- First, she picks a simple block cipher $\hat{\varepsilon} := (\hat{E}, \hat{D})$ that is clearly insecure on its own. We call $\hat{\varepsilon}$ the **round cipher**.

- Second, she picks a simple (not necessarily secure) PRG G that is used to expand the key k into d keys $k_1, \ldots, k_d$ for $\hat{\varepsilon}$. We call G the key **expansion function**.

Algorithm E(k, x):

- step 1. **key expansion**: use the key expansion function G to stretch the key k of ε to d keys of ξ:

$$(k_1,.., k_d) \leftarrow G(k)$$

- step 2. **iteration**: for i = 1,.., d apply $\widehat{E}(k_i, .)$, namely:

$$y \leftarrow \widehat{E}(k_d, \widehat{E}(k_{d-1},.., \widehat{E}(k_2, \widehat{E}(k_1, x))...))$$

Each application of $\widehat{E}$ is called a **round** and the total number of rounds is d.

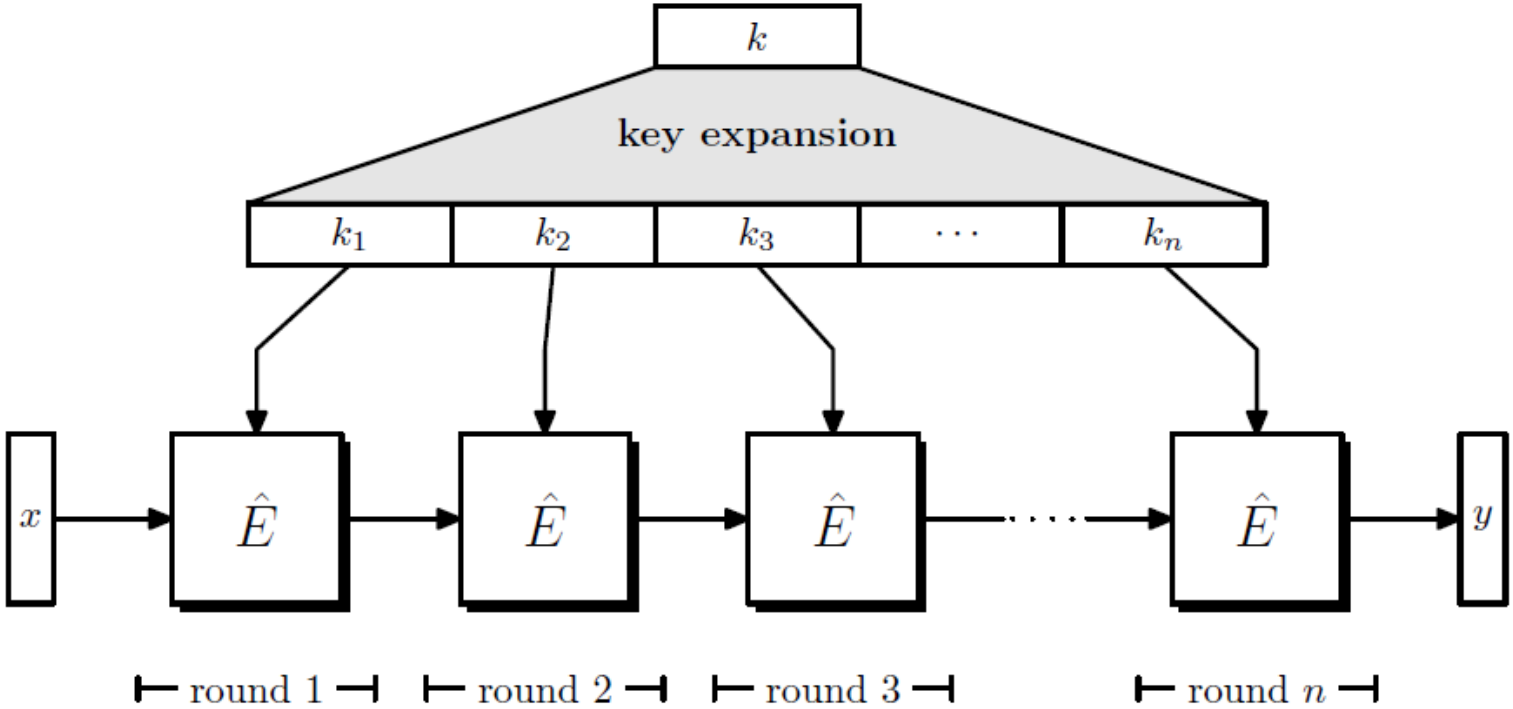
The keys $k_1,..., k_d$ are called **round keys**.

The decryption algorithm D{k, y) is identical except that the round keys are applied in reverse order. D(k,y) is defined as:

$$x \leftarrow \widehat{D}(k_1, \widehat{D}(k_2,.., \widehat{D}(k_{d-1}, \widehat{D}(k_d, y))...))$$

NYU

Block Ciphers Built by Iteration



Sample block ciphers

	key size (bits)	block size (bits)	number of rounds	performance[1] (MB/sec)
DES	56	64	16	80
3DES	168	64	48	30
AES-128	128	128	10	163
AES-256	256	128	14	115

NYU

The DES algorithm

The Data Encryption Standard (DES) algorithm consists of 16 iterations of a simple round cipher. To describe DES it suffices to describe the DES round cipher and the DES key expansion function. We describe each in turn.

The Feistel permutation. One of the key innovations in DES, invented by Horst Feistel at IBM, builds a permutation from an arbitrary function.

Let f: X→X be a function. We construct a permutations π: X^2 → X^2 follows:

$$\pi\,(x,y) := (y,\ x \oplus f(y))$$

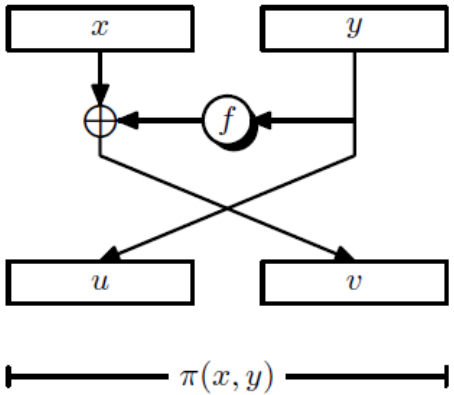
To show that π is one-to-one we construct its inverse, which is given by:

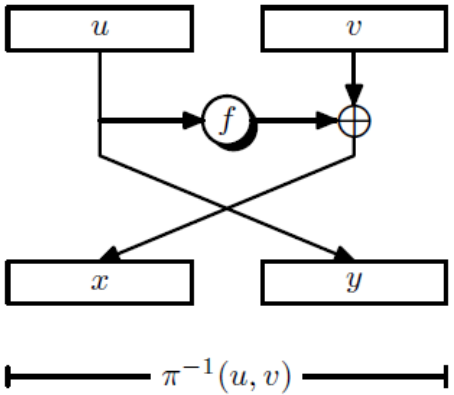
$$\pi^{-1}\,(u,v) = (v \oplus f(u),\ u)$$

Feistel permutation

The function π is called a **Feistel permutation** and is used to build the DES round cipher. The composition of n Feistel permutations is called an n-round Feistel network. Block ciphers designed as a Feistel network are called **Feistel ciphers**. For DES, the function f takes 32-bit inputs and the resulting permutation π operates on 64-bit blocks.

Note that the Feistel inverse function π^{-1} is almost identical to π. As a result, the same hardware can be used for evaluating both π^{-1} and π . This in turn means that the encryption and decryption circuits can use the same hardware.





NYU

Feistel permutation – cont'd

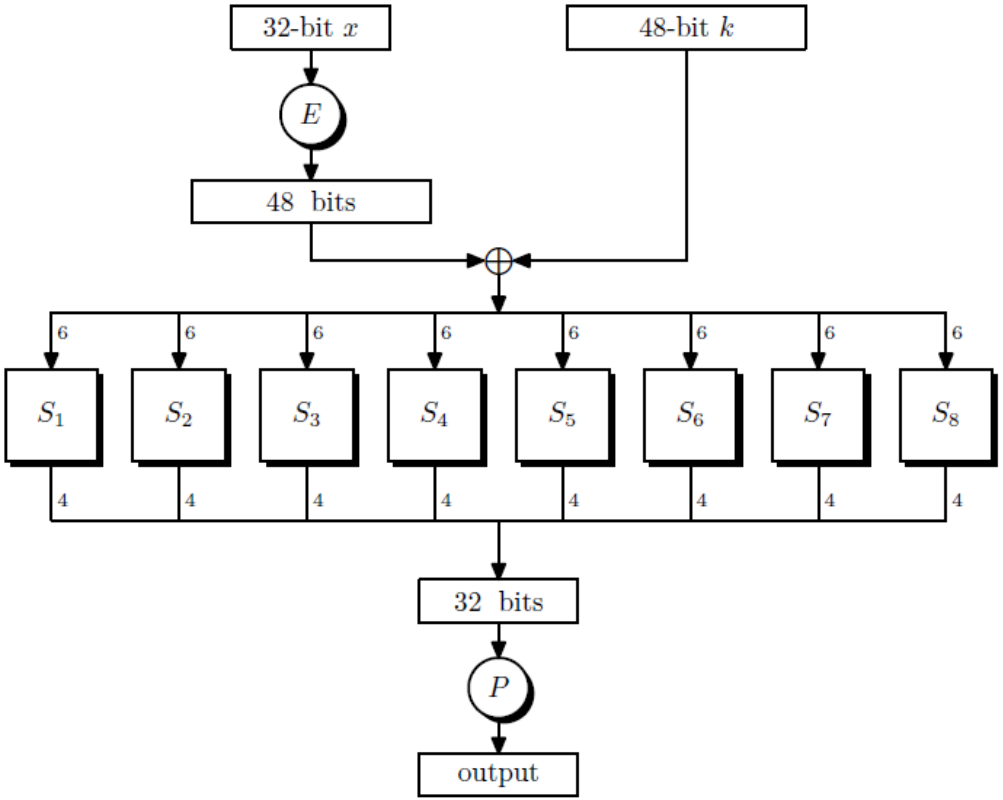
The DES round function F(k,x). The DES encryption algorithm is a 16-round Feistel network where each round uses a different function f: X → X. ln round number i the function f is defined as:

$$f(x) := F(k_i, x)$$

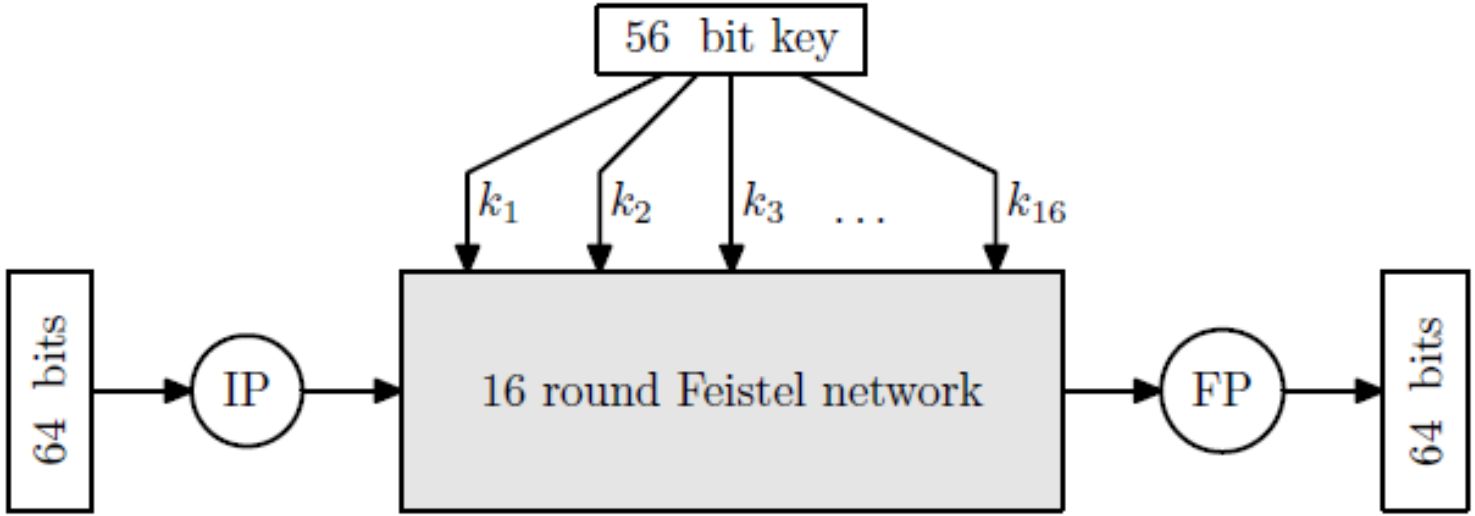
where k is a 48-bit key for round number i and F is a fixed function called the **DES round function**. The function F is the centerpiece of the DES algorithm, and it uses several auxiliary functions E, P, and S1,.. . , S8 defined as follows:

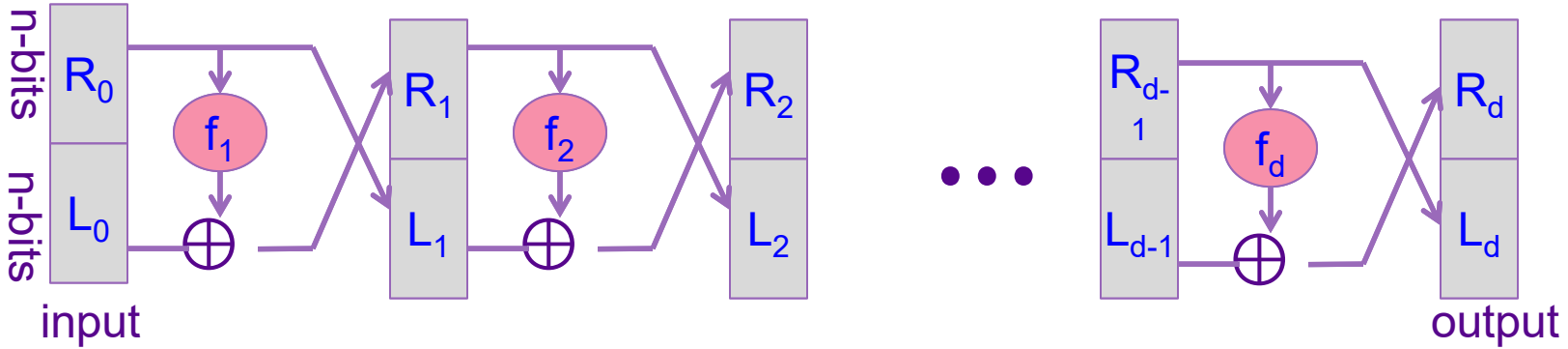
- The function E expands a 32-bit input to a 48-bit output by rearranging and replicating the input bits. For example, E maps input bit number 1 to output bits 2 and 48; it maps input bit 2 to output bit number 3, and so on.
- The function P, called the mixing permutation, maps a 32-bit input to a 32-bit output by rearranging the bits of the input. For example, P maps input bit umber 1 to output bit number 9; input bit number 2 to output number 15, and so on.
- At the heart of the DES algorithm are the functions S1,.. . , S8 called S-boxes. Each S-box S_i maps a 6-bit input to a 4-bit output by a lookup table. The DES standard lists these 8 look-up tables, where each table contains 64 entries.

The DES round function F(k; x)



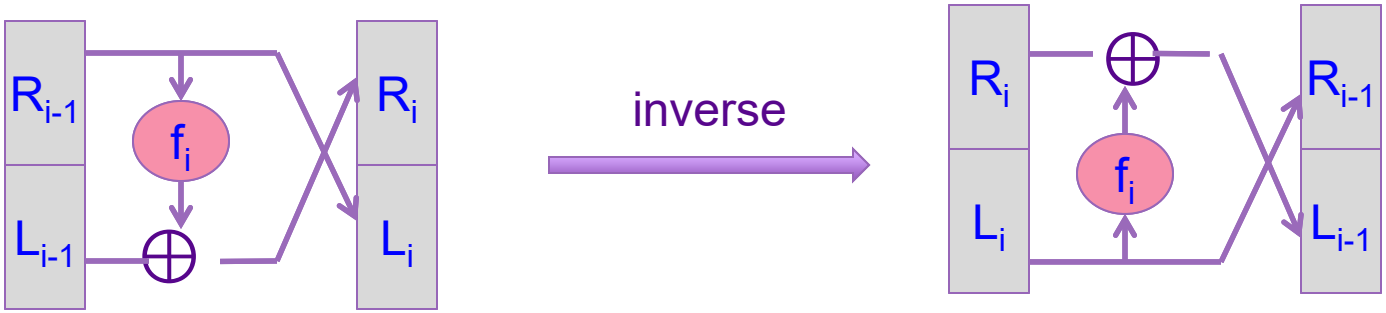
The complete DES circuit





For all $f_1, \ldots, f_d$: $\{0,1\}^n \longrightarrow \{0,1\}^n$

Feistel network $F: \{0,1\}^{2n} \longrightarrow \{0,1\}^{2n}$ is invertible



The S-boxes

- S_i: $\{0,1\}^6 \longrightarrow \{0,1\}^4$

S_5		Middle 4 bits of input															
		0000	**0001**	**0010**	**0011**	**0100**	**0101**	**0110**	**0111**	**1000**	**1001**	**1010**	**1011**	**1100**	**1101**	**1110**	**1111**
Outer bits	**00**	0010	1100	0100	0001	0111	1010	1011	0110	1000	0101	0011	1111	1101	0000	1110	1001
	01	1110	1011	0010	1100	0100	0111	1101	0001	0101	0000	1111	1010	0011	1001	1000	0110
	10	0100	0010	0001	1011	1010	1101	0111	1000	1111	1001	1100	0101	0110	0011	0000	1110
	11	1011	1000	1100	0111	0001	1110	0010	1101	0110	1111	0000	1001	1010	0100	0101	0011

NYU

NYU

The AES block cipher

- Like many real-world block ciphers, Advanced Encryption Standard (AES) is an iterated cipher that iterates a simple round cipher several times. The number of iterations depends on the size of the secret key:

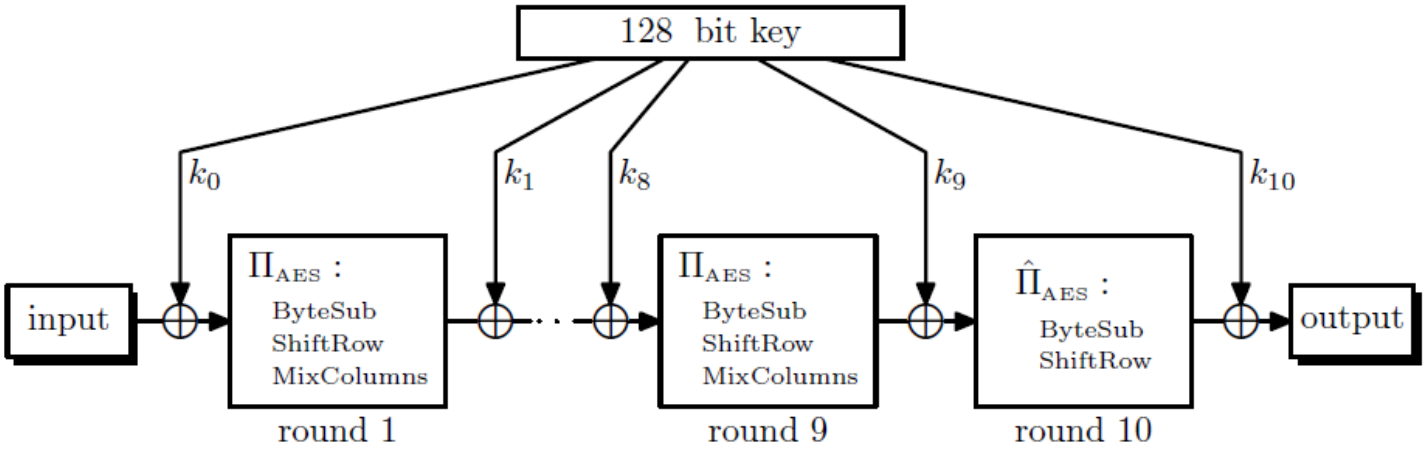
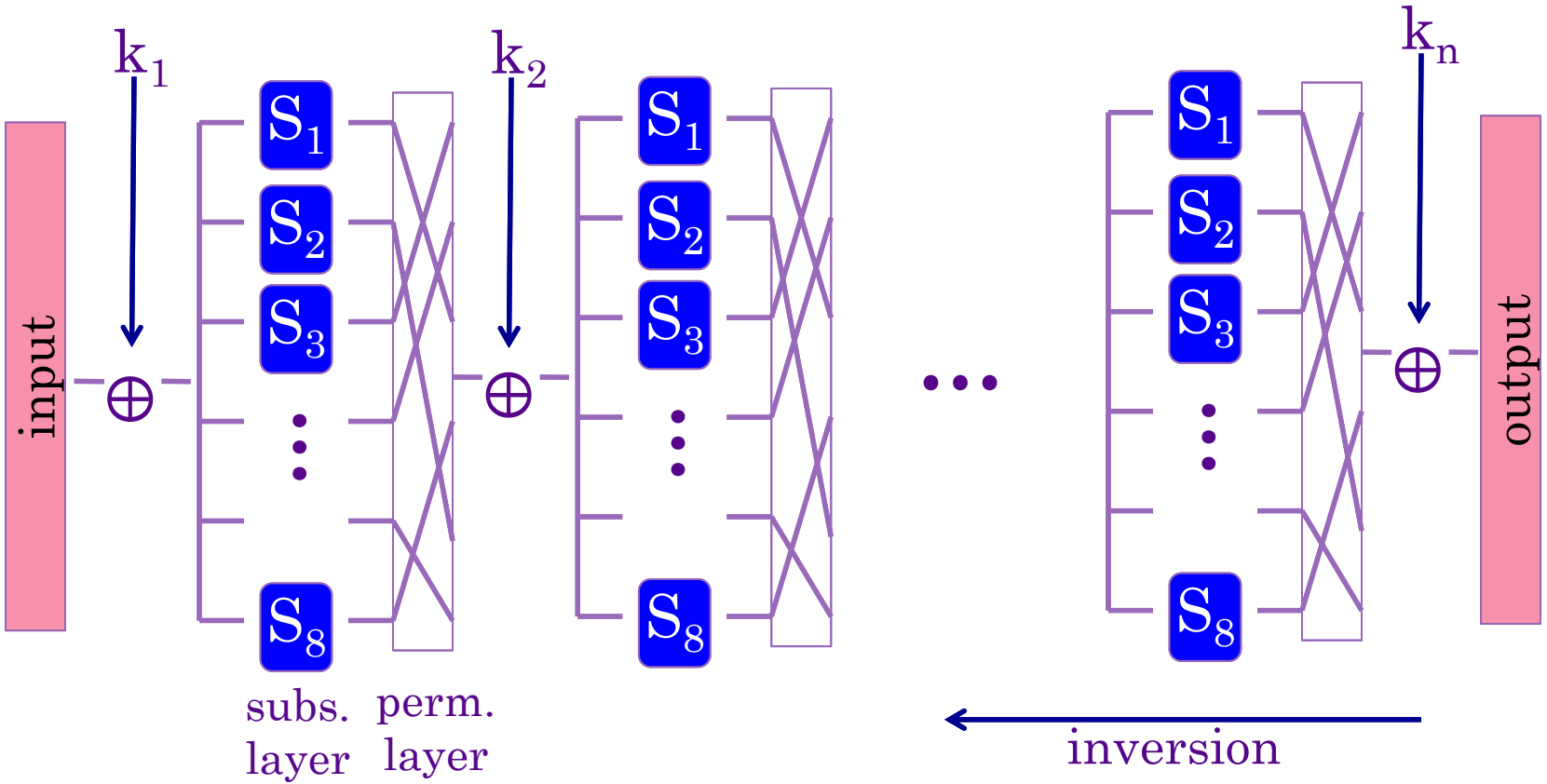
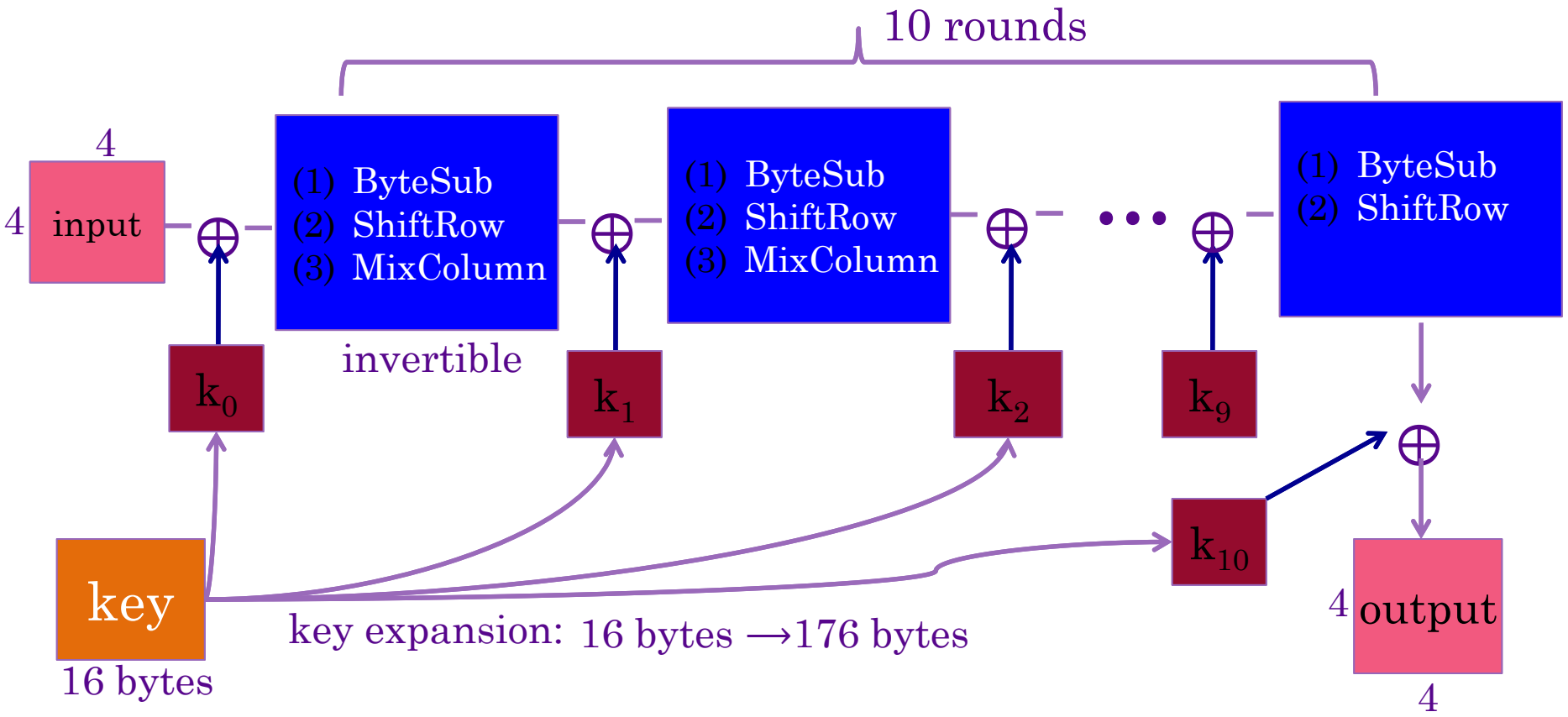


Figure 4.11: Schematic of the AES-128 block cipher

AES is a Subs-Perm network (not Feistel)

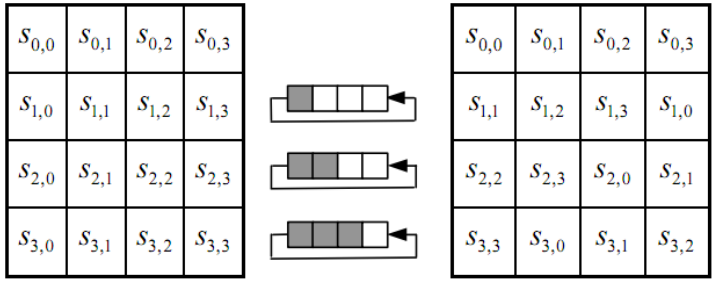


AES-128 schematic

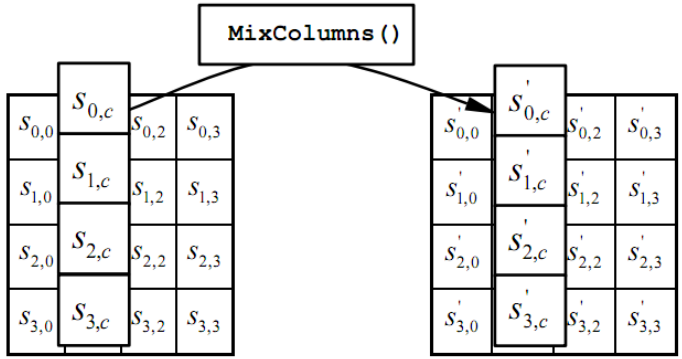


The round function

- **ByteSub**: a 1 byte S-box. 256 byte table (easily computable)
- **ShiftRows**:



- **MixColumns**:



AES Key Sizes

cipher name	key-size (bits)	block-size (bits)	number of rounds
AES-128	128	128	10
AES-192	192	128	12
AES-256	256	128	14

NYU

The AES round permutation

The AES round permutation. The permutation Π_{AES} is made up of a sequence of three invertible operations on the set $\{0,1\}^{128}$, The input 128-bits is organized as a 4x4 array of cells, where each cell is eight bits, The following three invertible operations are then carried out in sequence, one after the other, on this 4 x 4 array:

1. SubBytes: Let S: $\{0,1\}^8 \rightarrow \{0,1\}^8$ be a fixed permutation (a one-to-one function). This permutation is applied to each of the 16 cells, one cell at a time. The permutation S is specified in the AES standard as a hard-coded table of 256 entries. It is designed to have no fixed points, namely $S(x) \neq x$ for all $x \in \{0,1\}^8$ and no inverse fixed points, namely $S(x) \neq \bar{x}$ where $\bar{x}$ is the bit-wise complement of x. These requirements are needed to defeat certain attacks discussed in Section 4.3.1.

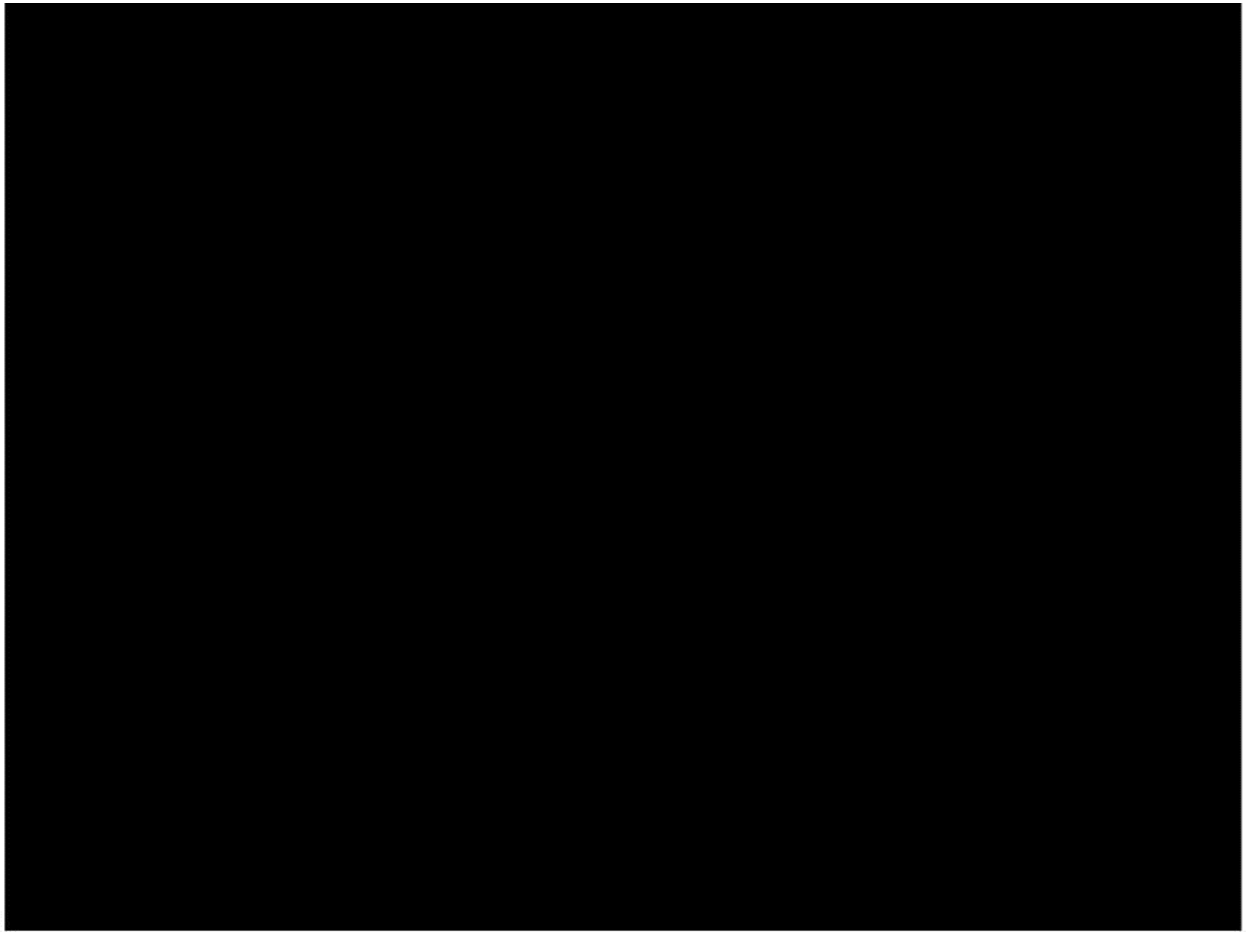
The AES round permutation – cont'd

2. ShiftRows: this type of forms of cyclic shift on the four rows of the input 4*4 array: the first row is unchanged, the second row is cyclically shifted one byte to the left, the third row is cyclically shifted two bytes and the fourth row cyclically shifted three bytes. In a diagram this step performs the following transformation

$$\begin{pmatrix} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \\ a_8 & a_9 & a_{10} & a_{11} \\ a_{12} & a_{13} & a_{14} & a_{15} \end{pmatrix} \Longrightarrow \begin{pmatrix} a_0 & a_1 & a_2 & a_3 \\ a_5 & a_6 & a_7 & a_4 \\ a_{10} & a_{11} & a_8 & a_9 \\ a_{15} & a_{12} & a_{13} & a_{14} \end{pmatrix} \tag{4.11}$$

3. MixColumns: ln this step the 4*4 array is treated as a matrix and this matrix is multiplied by a fixed matrix. Specifically, the MixColumns transformation does:

$$\begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \times \begin{pmatrix} a_0 & a_1 & a_2 & a_3 \\ a_5 & a_6 & a_7 & a_4 \\ a_{10} & a_{11} & a_8 & a_9 \\ a_{15} & a_{12} & a_{13} & a_{14} \end{pmatrix} \Longrightarrow \begin{pmatrix} a'_0 & a'_1 & a'_2 & a'_3 \\ a'_4 & a'_5 & a'_6 & a'_7 \\ a'_8 & a'_9 & a'_{10} & a'_{11} \\ a'_{12} & a'_{13} & a'_{14} & a'_{15} \end{pmatrix} \tag{4.12}$$



NYU