Sicherheit:
Fragen und Lösungsansätze

im Wintersemester 2012 / 2013

Prof. Dr. Jan Jürjens

TU Dortmund, Fakultät Informatik, Lehrstuhl XIV

Teil 7: Symmetric Encryption
v. 15.01.2013

technische universität
dortmund

fakultät für
informatik

Themen der Vorlesung

Encryption mechanism: functionality

- underlying sets:
  - D domain set of (possible) *plaintexts*
  - R range set of (possible) *ciphertexts*
  - $K = EK \quad DK$ set K of (possible) *keys*, each of which comprises
    - $ek \in EK$ *encryption key*
    - $dk \in DK$ *decryption key*
- $Gen : \rightarrow K$ *key generation* algorithm,
  might take a natural number l as a *security parameter*
- $Enc : EK \quad D \rightarrow R$ *encryption* algorithm,
  transforms a plaintext $x \in D$
  into a ciphertext $y = Enc(ek, x) \in R$
  using an encryption key $ek \in EK$
- $Dec : DK \quad R \rightarrow D$ *decryption* algorithm,
  transforms a ciphertext $y \in R$
  into a plaintext $x = Dec(dk, y) \in D$
  using a decryption key $dk \in DK$

Encryption mechanism: properties

- ***correctness***

  using a generated key pair,

  any encryption can be reversed by the corresponding decryption, i.e.,

  for all keys $(ek, dk) \in EK \quad DK$ generated by *Gen*,

  for all plaintexts $x \in D$:

  $$Dec\ (\ dk,\ Enc(ek,\ x))\ =\ x$$

- ***secrecy*** **(*naive* version)**

  without knowing the pertinent decryption key *dk*,

  an (unauthorized) observer of a ciphertext $y = Enc\ (\ ek\ ,\ x\ )$

  cannot "determine" the corresponding plaintext *x*

  (*semantic* version: such an observer

  can "determine" only those properties of the corresponding plaintext *x*

  that he could "determine" without knowing the ciphertext *y* at all)

- *efficiency*

  algorithms *Gen*, *Enc* and *Dec* are efficiently computable

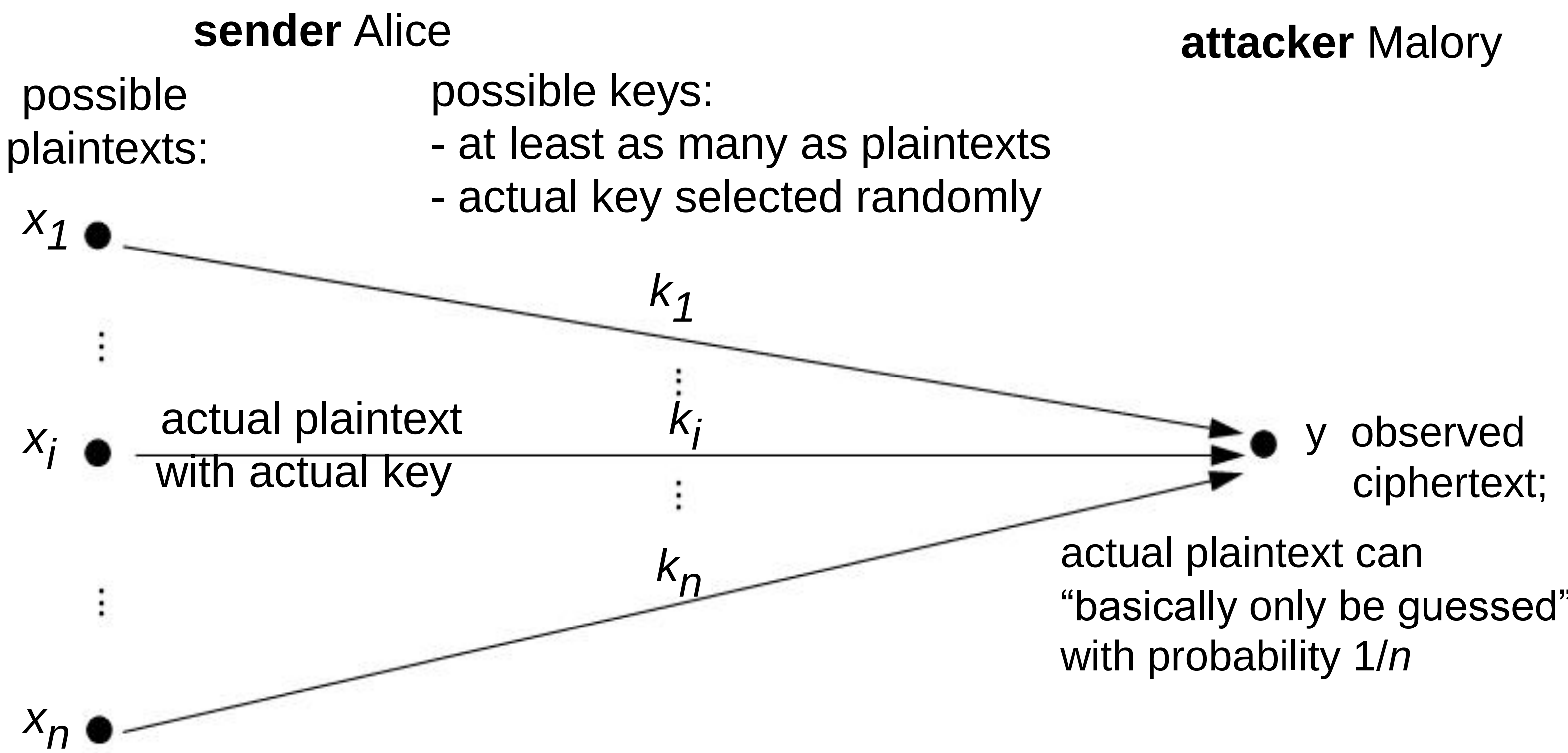
Classification

- ***mode of operation***:
  blockwise or *streamwise*

- ***relationship between keys***:
  symmetric or *asymmetric*

- ***justification of a secrecy property***:
  one-time key or *one-way function* or *chaos*

Probability-theoretic secrecy property (one-time key approach)



Complexity-theoretic secrecy property (one-way function approach)

sender Alice

attacker Malory

possible plaintexts:

⋮

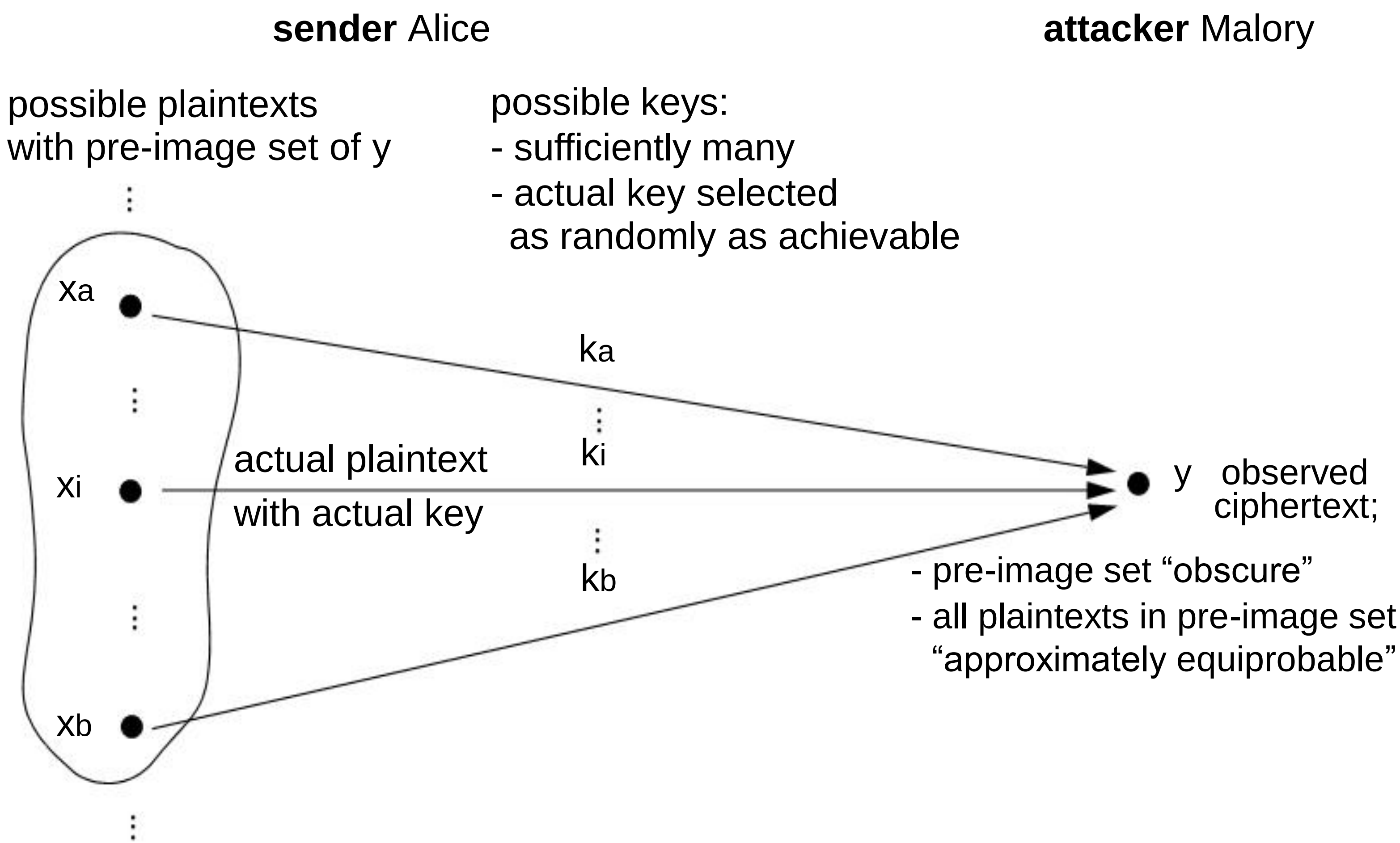
public encryption key

x_i → y observed ciphertext;

⋮

matching plaintext x_i
cannot be “feasibly determined”
since computational effort is too high
(without knowledge of the private key)

Empirical secrecy property (chaos approach / confusion and diffusion)



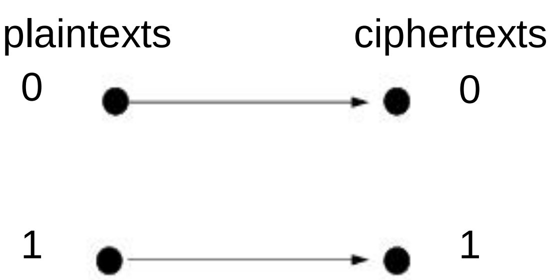
One-time keys and perfect ciphers (Vernam)

- are based on
  - a sufficient (and „nearly necessary“) condition for *perfectness*, achieving *probability-theoretic secrecy*
  - the resulting group-based construction
- are *symmetric*, having *identical* encryption key and decryption key
- are restricted to a *single* key usage
- operate *streamwise* by considering a plaintext as a sequence of bits, each of which is treated separately

One-time keys: treating a single bit

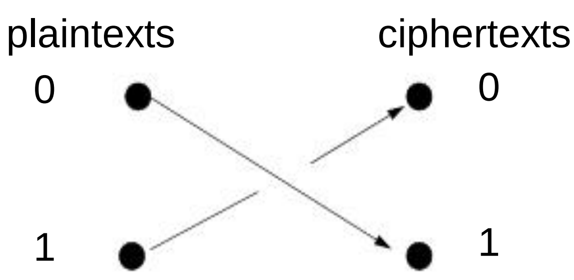
- *plaintext* domain, *ciphertext* range and *key* set are chosen as { 0 , 1}
- set { 0 , 1} is seen as the carrier of the *group* ($\mathbf{Z}_2$, + , 0) of *residue classes* modulo 2, where the residue classes are identified with their representatives 0 and 1
- *group operation* of addition modulo 2 is identical to the Boolean operation XOR (exclusive or, denoted by the operator $\oplus$)

key 0



Enc(0,*x*) = 0 $\oplus$ *x* = *x*
delivers the identity permutation

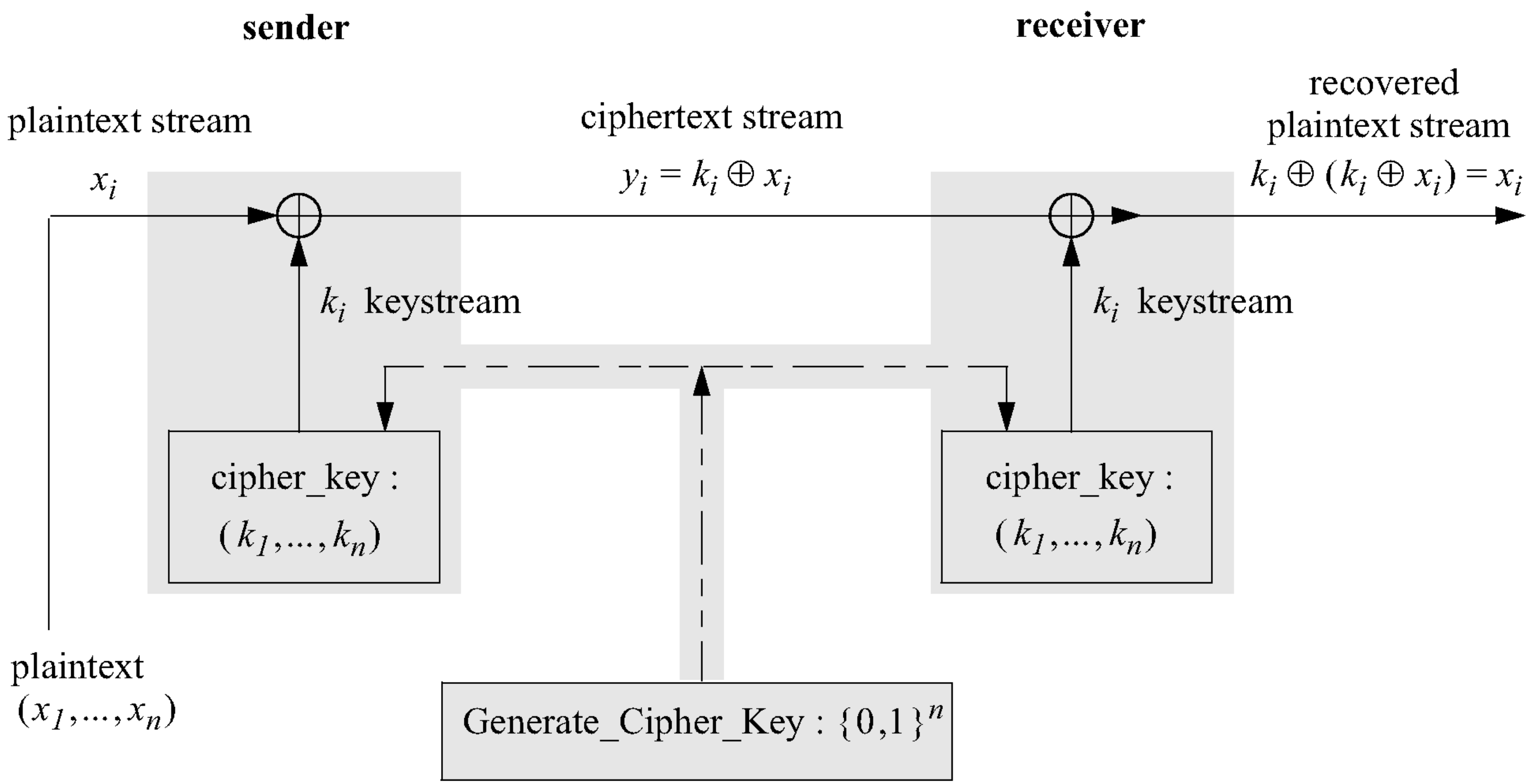
key 1



Enc(1,*x*) = 1 $\oplus$ *x* = 1 - *x*
delivers the exchanging permutation

One-time keys: handling bit strings of length n

- employ the corresponding *product group*:
  - take the group (Z_2 , + , 0) *n* times
  - define the *group operation* componentwise



One-time keys: underlying sets

- plaintexts: bit strings of length n, i.e.,
  "streams" (x_1 ,… , x_n) of length n over the set { 0 , 1}

- ciphertexts: bit strings of the same length n, i.e.,
  "streams" (y_1 ,… , y_n) of length n over the set { 0 , 1}

- keys: bit strings of the same length n, i.e.,
  "streams" (k_1 ,… , k_n) of length n over the set { 0 , 1}

One-time keys: algorithms

- ***key generation*** algorithm *Gen(erate_Cipher_Key)*
selects a "truly random" *cipher key* $(k_1 ,\dots , k_n)$

- ***encryption*** algorithm *Enc*
handles the plaintext $(x_1 ,\dots , x_n)$ and the cipher key $(k_1 ,\dots , k_n)$ as streams;
treats each corresponding pair of a plaintext bit x_i and a cipher key bit k_i
as input for a XOR operation, yielding a ciphertext bit
$y_i = k_i \oplus x_i$

- ***decryption*** algorithm *Dec*
handles the ciphertext $(y_1 ,\dots , y_n)$ and the cipher key $(k_1 ,\dots , k_n)$ as streams;
treats each corresponding pair of a ciphertext bit y_i and a cipher key bit k_i
as input for a XOR operation,
yielding the original plaintext bit x_i *correctly*:

$$k_i \oplus y_i = k_i \oplus (k_i \oplus x_i) = (k_i \oplus k_i) \oplus x_i = 0 \oplus x_i = x_i$$

One-time keys: applications

- restriction to using a key *only once* is crucial:
  observing a ciphertext/plaintext pair, an attacker achieves complete success:
  solve, for each position *i*, the equation $y_i = k_i \oplus x_i$
  regarding the secret key bit as
  $k_i = y_i \oplus x_i$
- considering the transmission of a *single* message:
  qualified to the best possible extent regarding *secrecy* and *efficiency*
- as a trade-off for the best secrecy - proved to be inevitable:
  - secret cipher key can be used only once
  - secret cipher key must be as long as the anticipated plaintext
- as a *stand-alone* mechanism,
  pure one-time key encryption is practically employed
  only in dedicated applications with extremely high secrecy requirements
- however, basic approach is widely exploited in
  - variants
  - subparts of other mechanisms

Stream ciphers with pseudorandom sequences (Vigenère)

- are a variant of the *one-time key* encryption mechanism

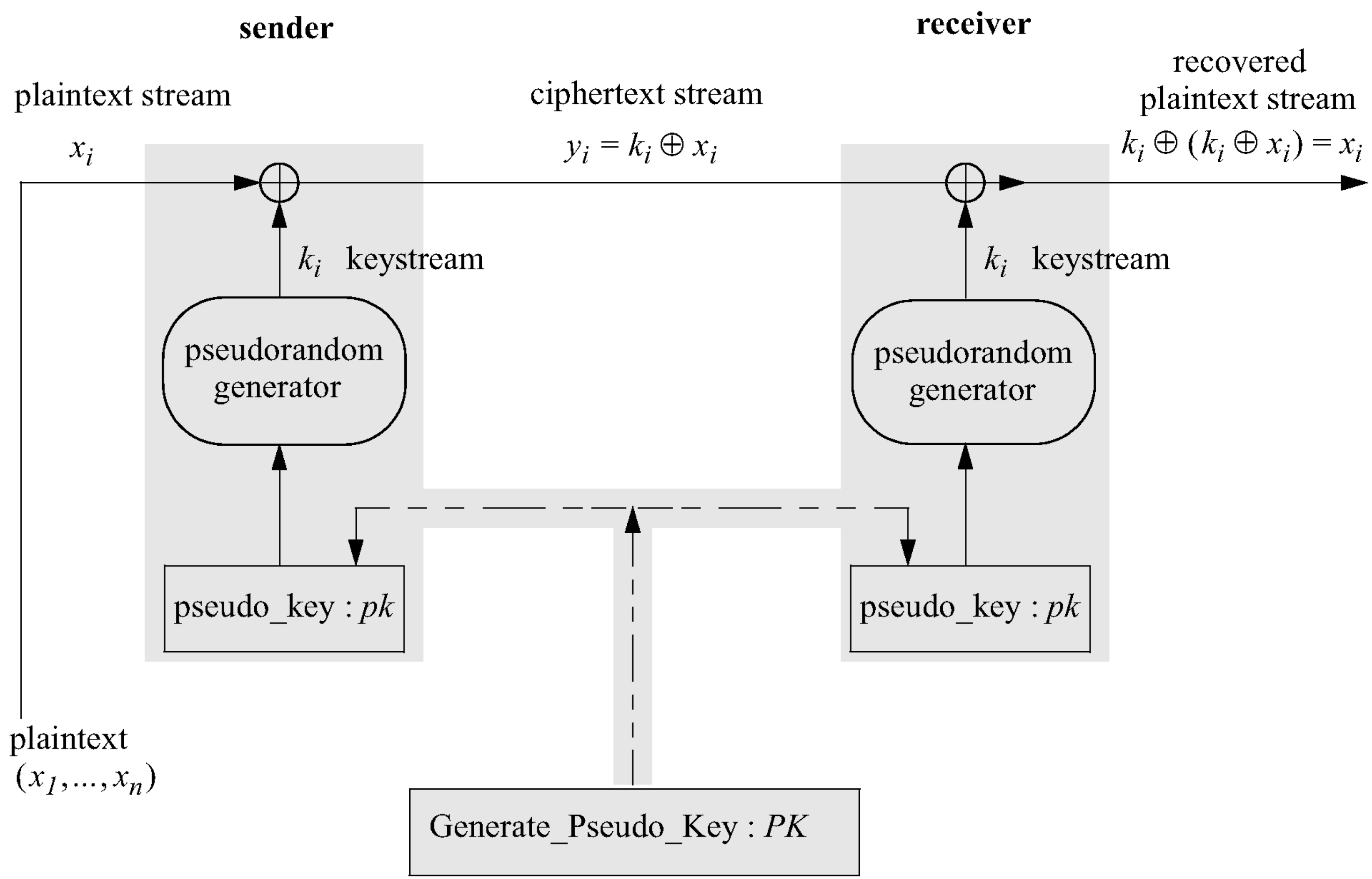
- are obtained by replacing the “truly random” *cipher key* by a *pseudorandom* one that is determined by a short(er) *pseudo-key*

- are *symmetric*

- operate *streamwise* by considering a plaintext as a sequence of bits, each of which is treated separately

- cannot be *perfect* or *probability-theoretically secure* in practice, since the pseudo-key is often substantially shorter than the generated cipher key

Vigenère: overall structure



DES (Data Encryption Standard)

- has been a most influential example of the *chaos* approach, used worldwide
- designed by IBM and the National Security Agency (NSA) of the USA
- standardized by the National Bureau of Standards (NBS)
in 1976/77 for “unclassified government communication”
- adopted by the American National Standards Institute (ANSI)
in 1981 for commercial and private applications
- is a *symmetric* mechanism, admitting *multiple* key usage
- operates *blockwise*, where the block length is 64 bits
- has a key length of 56 bits:
  today, the pure form of this mechanism is considered to be outdated,
  as it suffers from a too short key length
- has a still useful variant: *Triple-DES*

Triple-DES

inputs:

- a plaintext x / a ciphertext y
- three different keys k_1, k_2, k_3

encryption algorithm: successively perform

- an encryption with k_1,
- a decryption with k_2
- another encryption with k_3

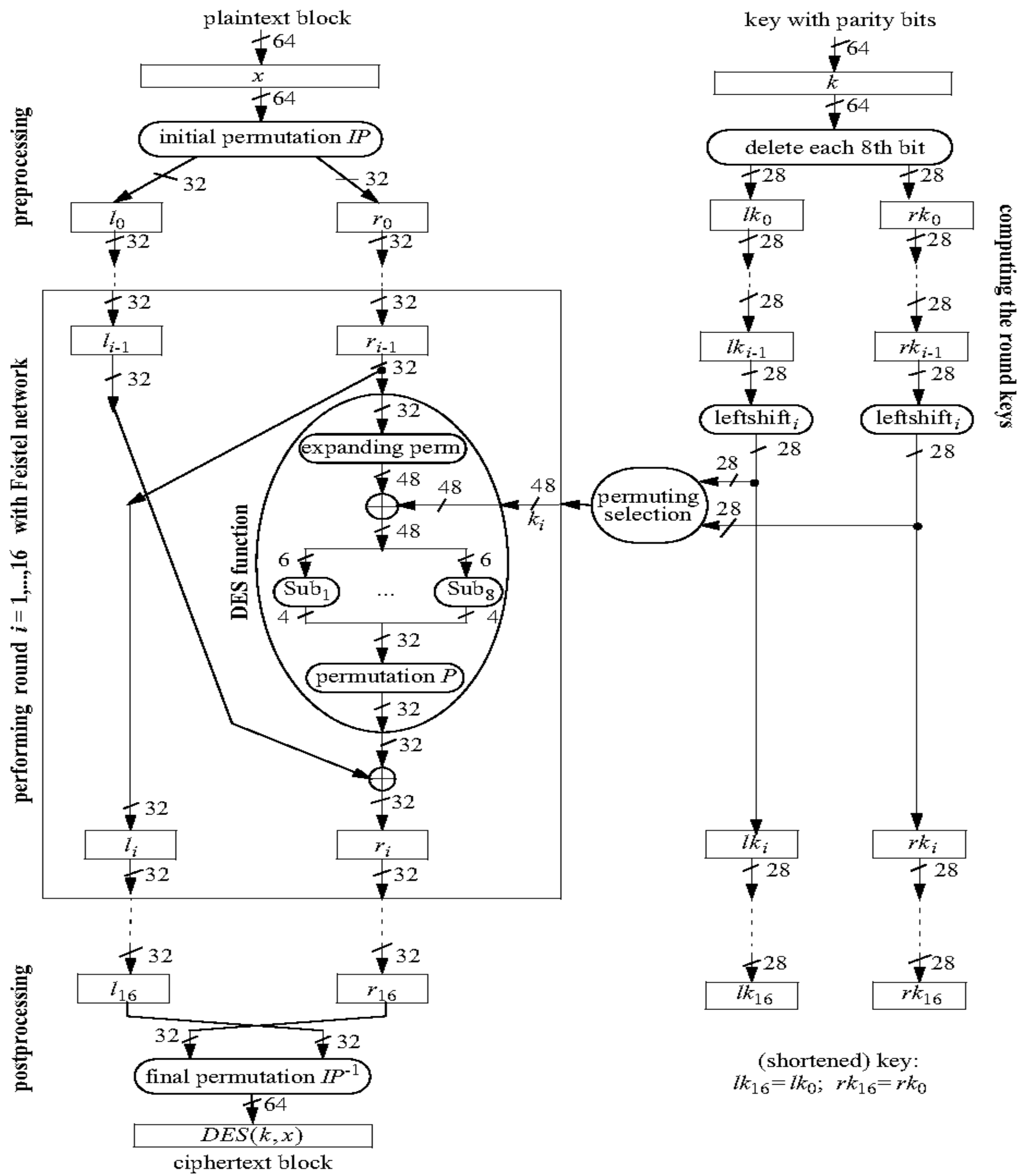
yielding the ciphertext y as

$$Enc\,(\,k_3\,,\,Dec\,(\,k_2\,,\,Enc\,(\,k_1\,,\ x\,)\,)\,)$$

decryption algorithm: perform corresponding inverse algorithms to obtain

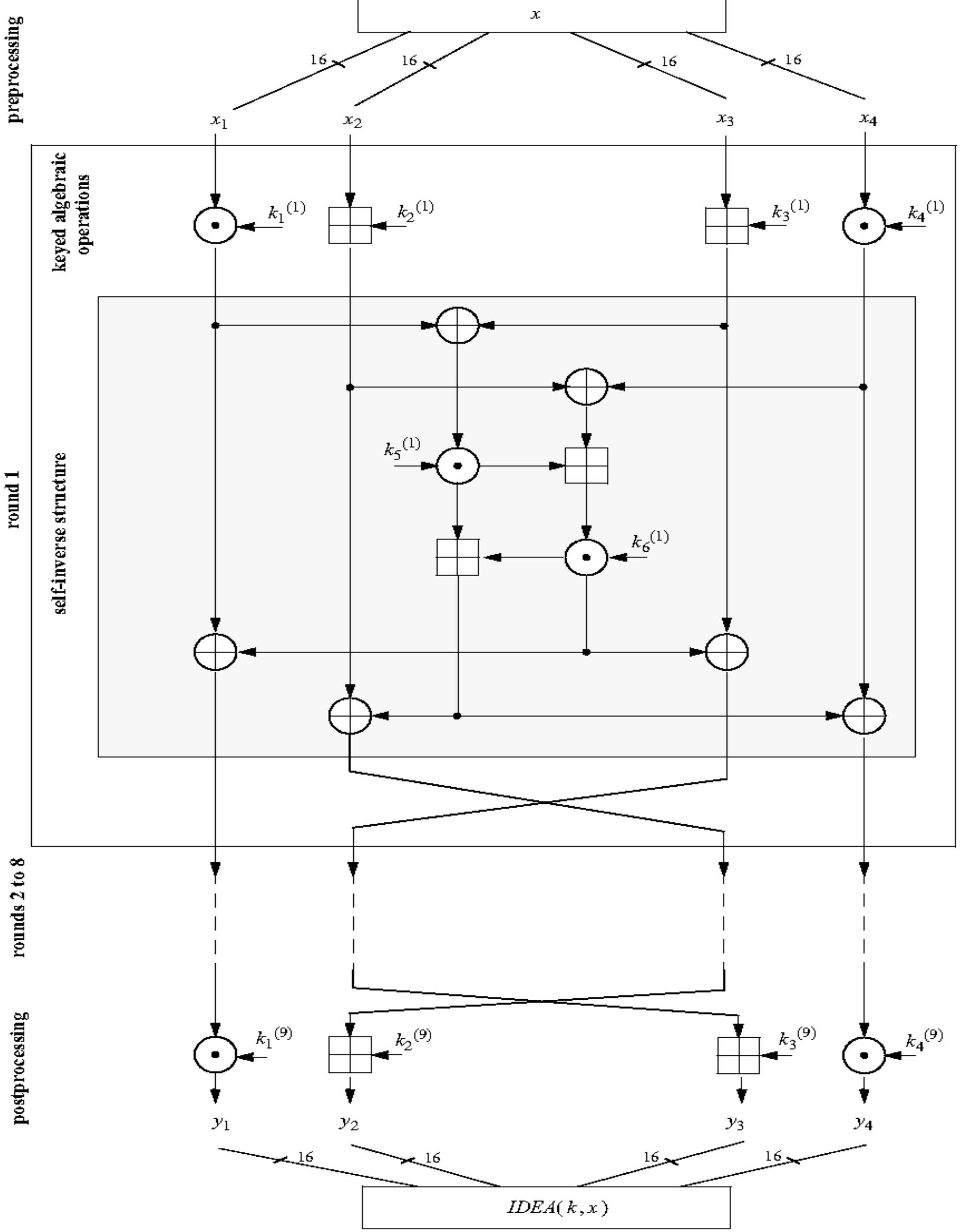
$$Dec\,(\,k_1,\,Enc\,(\,k_2,\,Dec\,(\,k_3,\,y)\,)\,)$$

DES: overall structure



IDEA (International Data Encryption Algorithm)

- was developed as an alternative to DES
- is a further example of the *chaos* approach
- combines
  - a DES-like round structure operating on block parts and round keys
  - algebraic group operations
- was adopted for Pretty Good Privacy (PGP),
but never reached common acceptance
- is *symmetric*, admitting *multiple* key usage
- operates *blockwise*, where the block length is 64 bits
- has a key length of 128 bits, still sufficient from today's perspective



IDEA: overall structure

AES-Rijndael (Advanced Encryption Standard)

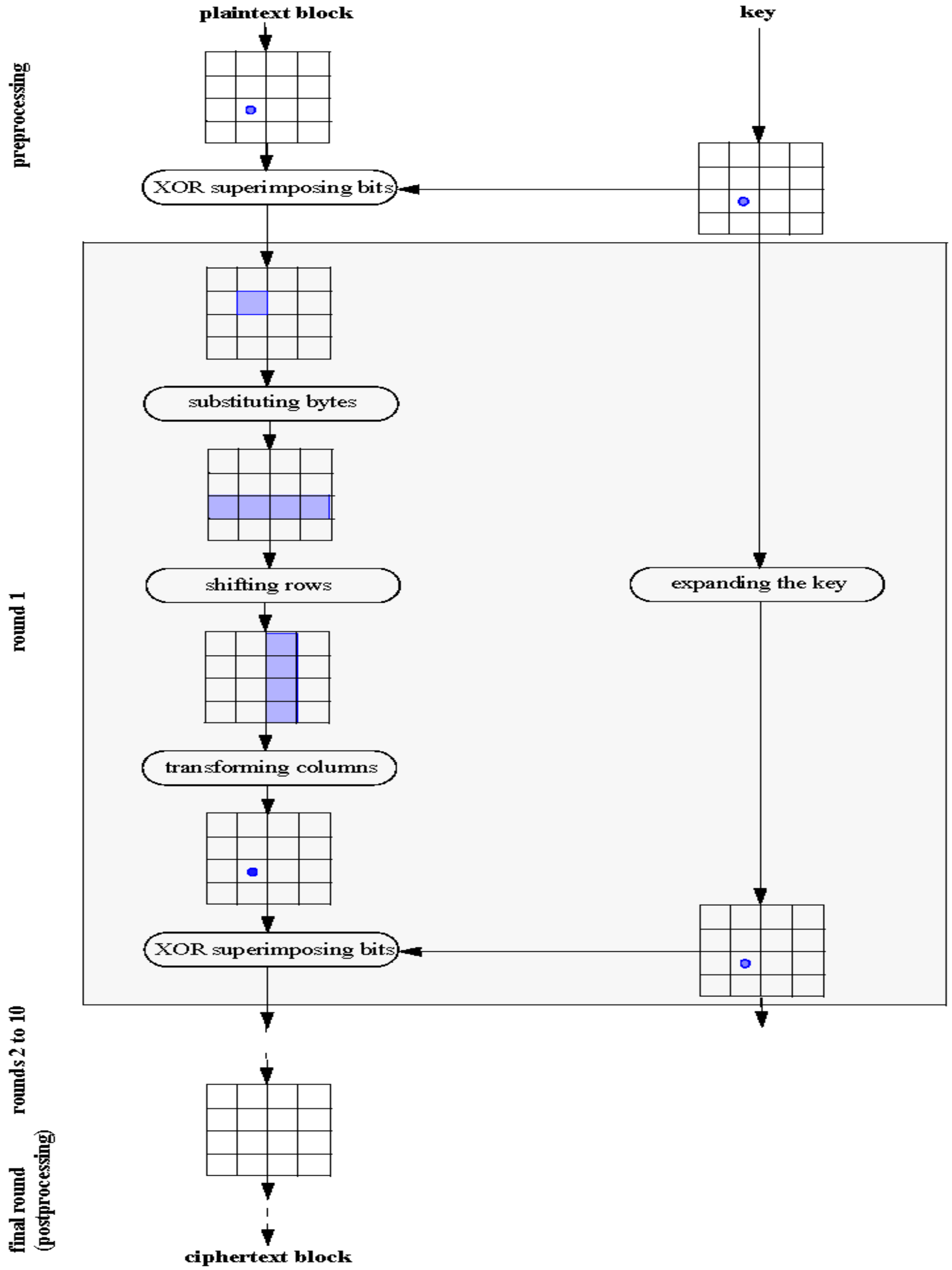
- was designed by the Belgian researchers J. Daemen and V. Rijmen, winner of a public competition and evaluation, organized by the NIST
- follows the *chaos* approach, producing *confusion* and *diffusion*
- is *symmetric*, admits *multiple* key usage, operates *blockwise*
- permits block lengths varying from 128 bits to any larger multiple of 32 bits
- permits key length varying from 128 bits to any larger multiple of 32 bits
- is somehow restricted for standardization:
  - block length is fixed at 128 bits
  - key length is restricted to be 128, 192 or 256 bits, today regarded as sufficient to resist *exhaustive search* and *trial attacks*
- combines several long-approved techniques
  - operating *roundwise* on block parts and round keys
  - *superimposing* the randomness of the key on the blocks using *XOR*
  - *permuting* the positions of a block or a key
  - *employing* of advanced algebraic operations showing *one-way behavior*

AES-Rijndael (Advanced Encryption Standard)

- operates on the following sets:
  - ***plaintexts*:**
    bit strings (blocks over { 0 , 1}) of length 128 (or a larger multiple of 32),
    represented as a byte matrix of 4 rows and 4 columns,
    thus having 16 entries of 8 bits each
  - ***ciphertexts*:**
    bit strings (blocks) of the same length as the plaintext blocks
  - ***keys*:**
    bit strings of length 128 (or a larger multiple of 32),
    again represented as a byte matrix like the plaintexts
- employs three algorithms as follows
  - ***key generation*:** select a “truly random” bit string of length 128
  - ***encryption*:** perform byte matrix transformations, see next pages
  - ***decryption*:** invert the byte matrix transformations in reverse order, employing the round keys accordingly

Encryption algorithm AES (k, x)

- takes a key *k* and a plaintext *x* as input
- represents them as byte matrices
- operates on the current byte matrices
- uses some preprocessing and postprocessing
- performs 10 (or more for larger block or key lengths) uniform *rounds*
- executes four steps in one round:
  (1) bytewise substitutions
  (2) permutations that shift positions within a row
  (3) transformations on columns and
  (4) bitwise XOR operations with the round key



Structure of the AES-Rijndael symmetric block cipher

AES-step (1): bytewise substitutions

- step (1) is defined by a *non-linear*, *invertible* function *SRD* on bytes, i.e., each byte of the current matrix is independently substituted by applying *SRD*
- *invertibility* ensures that a *correct* decryption is possible just by applying the inverse function SRD^{-1}
- *non-linearity* is aimed at achieving *confusion*, in terms of both
  - algebraic complexity
  - small statistical correlations between argument and value bytes
- the substitution function *SRD* has two convenient representations:
  - tabular representation organized as a lookup table of size 16x16
  - algebraic representation

Tabular representation of the substitution function

argument byte a: seen as composed of two hexadecimal symbols li and co
value byte v: table entry for line li and column co

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
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

Algebraic representation of the substitution function

- the representation treats a byte as an element of the finite field (Galois field) $GF(2^8)$, where each *bit* of a *byte* is seen as a coefficient of a polynomial with degree at most 7

- the multiplicative structure is defined by
  the usual *multiplication of polynomials*, followed by a *reduction*
  modulo the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$

- the function S_{RD} has a representation of the form

  $S_{RD}(a) = f(a^{-1})$, where

  - the inversion operation refers to the multiplicative structure of $GF(2^8)$
  - f is an affine function in $GF(2^8)$, basically described by
    - a suitable 8 8 bit matrix F
    - a suitable constant byte c

    such that
    $f(a) = (F \times a) \oplus c$

AES-step (2): permutations shifting positions within a row

- step (2) is defined by the offsets to be used for each of the rows: the offsets are 0, 1, 2 and 3 byte positions, meaning that
  - the first row remains invariant
  - the second, third and fourth rows are shifted by 8, 16 and 24 bit positions, respectively, to the left

- the shiftings are aimed at achieving good *diffusion*, and can be easily redone for a *correct* decryption

AES-step (3): transformations on columns

- step (3) is defined by a linear, invertible function MC_{RD} on "columns": each column of the current matrix is considered as an element of $\{0,1\}^{32}$ and independently substituted by applying MC_{RD}
- invertibility ensures that a *correct* decryption is possible
- the specific selection of MC_{RD} is aimed mainly at achieving *diffusion*, now regarding the rows of the byte matrices
- additionally, the selection was influenced by efficiency reasons
- MC_{RD} admits an algebraic definition in terms of polynomial multiplication:

$$MC_{RD}\left(\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}\right) = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

AES-step (4): bitwise XOR operations with the round key

- XOR *superimposes* the *randomness* of the sophisticatedly manipulated key on the intermediate state of the byte matrix
- effects of the superimposition can be *correctly* undone
  by applying these XOR operations with the same key arguments
- round keys are inductively computed
  by employing complex algebraic operations,
  while at the same time achieving an acceptable efficiency
- for the given block length and key length of 128 bits each (or suitably adapted for other possible lengths), the initial 4x4 byte matrix for the key *k* given as input is *expanded* into a 4x(1 + 10)·4 byte matrix, i.e., for each of the 10 rounds,
  four new columns are generated and taken as the *round key*

AES: key expansion

- the *key expansion* scheme distinguishes between
  the first column of a new round key and the remaining columns,
  but each column *i* is defined in terms of the
  - corresponding column *i* - 4 of the preceding round key
  - the immediately preceding column *i* - 1

- remaining columns:
  the column *i* is computed by directly applying the bitwise *XOR operation*

- first column:
  the preceding column is first transformed by a non-linear function
  that is a suitable composition of
  - the bytewise application of the substitution function *SRD*
  - a permutation that shifts the positions in a column
  - the addition of a round constant

AES: decryption

- there is a *straightforward decryption algorithm*:
  basically, it performs the inverses of all byte matrix transformation
  in reverse order, employing the round keys accordingly

- the design also includes an equivalent decryption algorithm:
it maintains the sequence of steps within a round,
replacing the steps by their respective inverses

AES: efficiency

- NIST requirements:
  successor of DES should enable an efficient implementation on *smartcards*, which could, for example, be used as *personal computing devices*
- the Rijndael proposal:
  the community was convinced regarding efficiency for implementations in both hardware and software
- the construction as a whole:
  high efficiency is enabled even though it operates on structures consisting of 128 bits (or even more)
- in combination with some block mode:
  transmission rates are suitable for large multimedia objects
- like any other symmetric block cipher:
  usage as part of a *hybrid encryption method* is possible

Question

- Why does it make sense to have a fixed, publically known bijection in step 1 of AES, given that applying a bijection does not change the information content of the data ?

General answer (independently of AES):

Imagine an encryption function E(x,k) which satisfies the following property:

E(x1::x2,k) = E(x1,k) :: E(x2,k)

(where :: is concatenation of bit-strings).

- Why is this property potentially dangerous ? (hint: integrity of encrypted data in transit when message structure is known)

- What can one do against this danger ? (hint: two alternatives)

Question

- Why does it make sense to have a fixed, publically known bijection in step 1 of AES, given that applying a bijection does not change the information content of the data ?

General answer (independently of AES):

Imagine an encryption function E(x,k) which satisfies the following property:

E(x1::x2,k) = E(x1,k) :: E(x2,k)

(where :: is concatenation of bit-strings).

- Why is this property potentially dangerous ? (hint: integrity of encrypted data in transit when message structure is known)

=> Example: What could the attacker do with: E("I owe you 100 EUR.",k) ?

=> E("I owe you 1000 EUR.",k) = E("I owe you 100",k):: E("0",k):: E("EUR.",k)

(and the latter three fragments can be extracted from E("I owe you 100 EUR.",k) using the equation above.

- What can one do against this danger ? (hint: two alternatives)

=> Apply bijection which breaks the equation. (Alternative: secure hash)

Stream ciphers using block modes

- *underlying block cipher*
  encrypts plaintext blocks and decrypts ciphertext blocks of a fixed length l_B

- *fragmentation*
  - divides a longer message into appropriate *fragments*
  - treats the resulting stream of fragments
    by using the block cipher in what is known as
    a *block mode* (*mode of operation*)

Two basic approaches to fragmentation

I) (1) the original message is divided into fragments of length equal to exactly the block length l_B of the underlying block cipher

(2) the block cipher treats the fragments
- either separately (electronic codebook)
- or in a suitably chained way (cipher block chaining)

II) (1) the original message is divided into fragments of length $l \leq l_B$ (typically, $l = 1$ or $l = 8$)

such that a plaintext stream of bits or bytes results

(2) the underlying block cipher is used

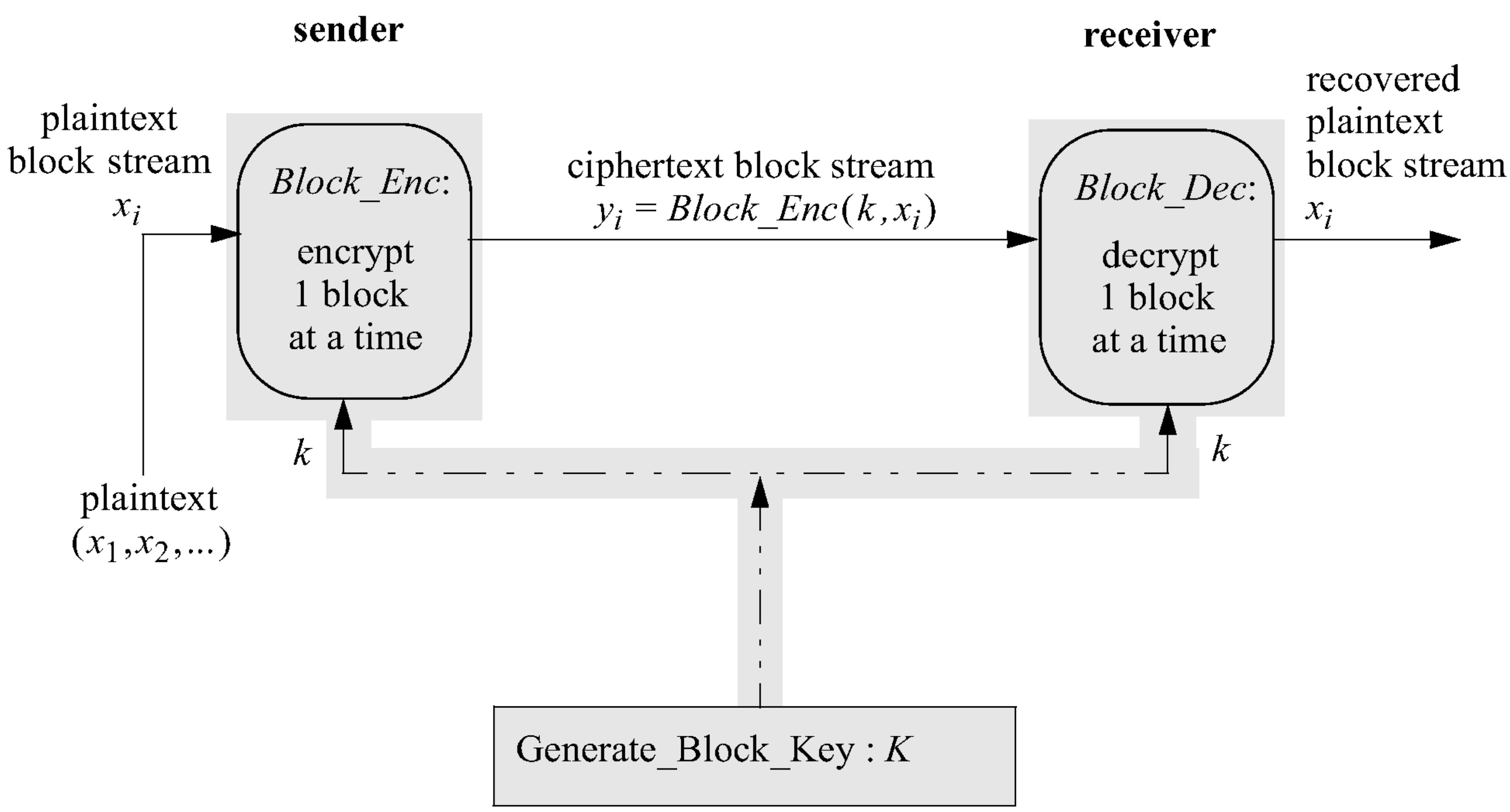
to generate a corresponding (apparently pseudorandom) cipher key stream that is superimposed on the plaintext stream

by using the *XOR operation*

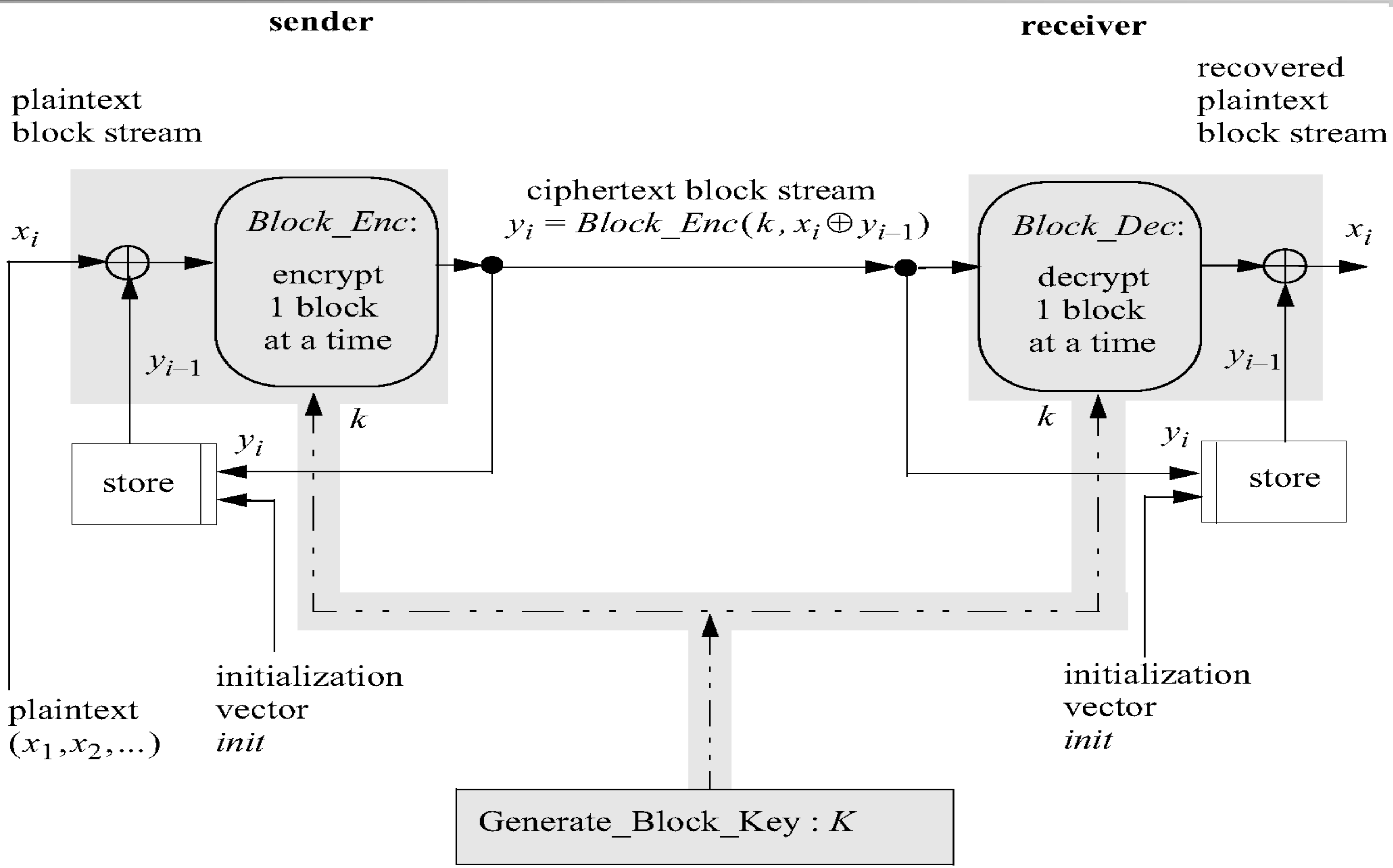
(cipher feedback, output feedback, counter-with-cipher-block-chaining)

can be seen as a variant of the *one-time key* encryption mechanism, where *perfectness* is abandoned for the sake of a reusable, short key as demanded by the underlying block cipher

Electronic Codebook (ECB) Mode



Cipher Block Chaining (CBC) Mode



CBC: correctness

- **encryption** algorithm *Enc*:
  - for the first block x_1,

    $Enc\ (k\ ,\ x_1\ ) := Block\_Enc\ (k\ ,x_1 \oplus init\ )$

  - for all further blocks x_i with $i > 1$,

    $Enc\ (k\ ,x_i\ ) := Block\_Enc\ (k\ ,x_i \oplus Enc\ (k, x_{i-1})\ )$
- **decryption** algorithm *Dec*:
  - for $i = 1$,

    $Dec\ (k\ ,y_1\ ) := Block\_Dec\ (k\ ,y_1\ ) \oplus init$

    $= Block\_Dec\ (k\ ,Block\_Enc\ (k\ ,x_1 \oplus init\ )\ ) \oplus init$

    $= (x_1 \oplus init\ ) \oplus init = x_1$
  - for $i > 1$,

    $Dec\ (k\ ,y_i\ )$

    $:= Block\_Dec\ (k\ ,y_i\ ) \oplus y_{i-1}$

    $= Block\_Dec\ (k\ ,Enc\ (k\ ,x_i)\ ) \oplus Enc\ (k\ ,x_{i-1}\ )$

    $= Block\_Dec\ (k\ ,Block\_Enc\ (k\ ,x_i \oplus Enc\ (k\ ,x_{i-1})\ )\ ) \oplus Enc\ (k\ ,x_{i-1}\ )$

    $= (x_i \oplus Enc\ (k\ ,x_{i-1}\ )\ ) \oplus Enc\ (k\ ,x_{i-1}\ ) = x_i$

CBC: producing a message digest

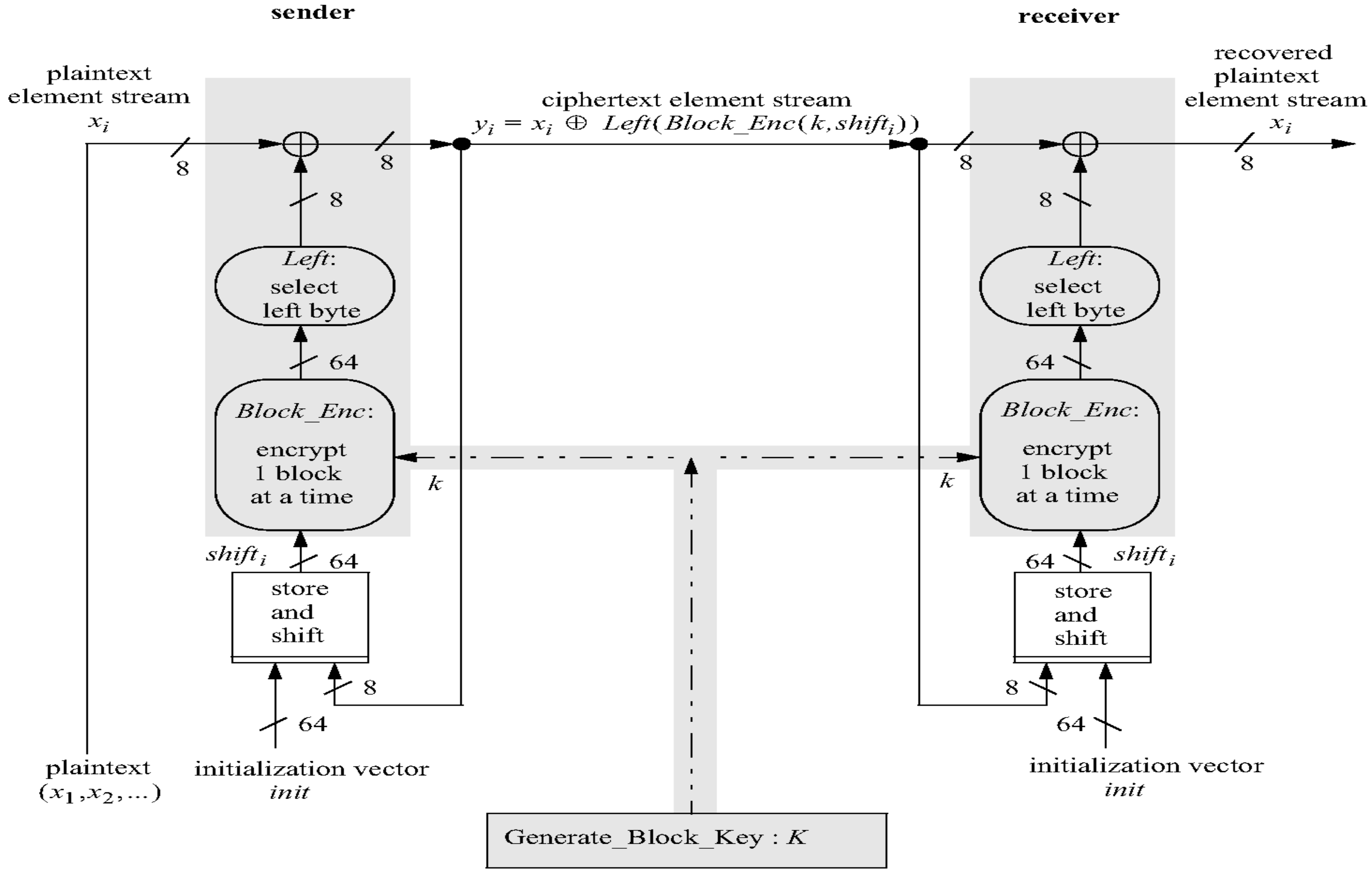
- characteristic feature of the cipher block chaining mode:
  all blocks are treated in a connected way
  requiring strict serialization

- the last resulting ciphertext block seen as a *message digest*:
  this block can be employed as a
  piece of *cryptographic evidence* (a *cryptographic exhibit*)
  for an *authenticity verification* algorithm

Cipher Feedback (CFB) Mode

- follows the second basic approach,
  achieving a variant of the one-time key encryption mechanism
- generates the required pseudorandom *cipher key* stream
  by means of the encryption algorithm *Block_Enc*(*ryption*)
  of the underlying block cipher
- does not employ the corresponding decryption algorithm,
  and thus *cannot* be used for an asymmetric block cipher
- extracts the cipher key stream from the outputs of the block cipher encryption,
  whose inputs are taken as a feedback from the ciphertext stream
- uses an *initialization vector init* as a *seed*,
  which must be used only once
  but can be communicated to the receiver without protection
- example: fragment length $l = 8$
  block size of the underlying block cipher $l_B = 64$

CFB: overall structure



CFB: correctness

- **encryption** algorithm *Enc*:
for each plaintext byte x_i,
$Enc\ (\ k\ ,\ x_i\ ) := \ x_i\ \ \oplus\ \ Left\ (\ Block\_Enc\ (\ k\ ,\ shift\_sender_i\ )\ ).$

- **decryption** algorithm *Dec*
for each ciphertext byte y_i,

$Dec\ (\ k\ ,\ y_i\ ) := \ y_i\ \ \oplus\ \ Left\ (\ Block\_Enc\ (\ k\ ,\ shift\_receiver_i\ )\ )$
$=(\ x_i \oplus\ Left\ (\ Block\_Enc\ (\ k\ ,\ shift\_sender_i\ )\ )\ )$
$\oplus\ \ Left\ (\ Block\_Enc\ (\ k\ ,\ shift\_receiver_i\ )\ ) = \ \ x_i\ ,$
provided $shift\_sender_i = shift\_receiver_i$

- required equality of the *shifti* inputs on both sides is achieved
by using the same initialization vector *init* and then, inductively,
by employing the same operations and inputs to generate them

CFB: producing a message digest

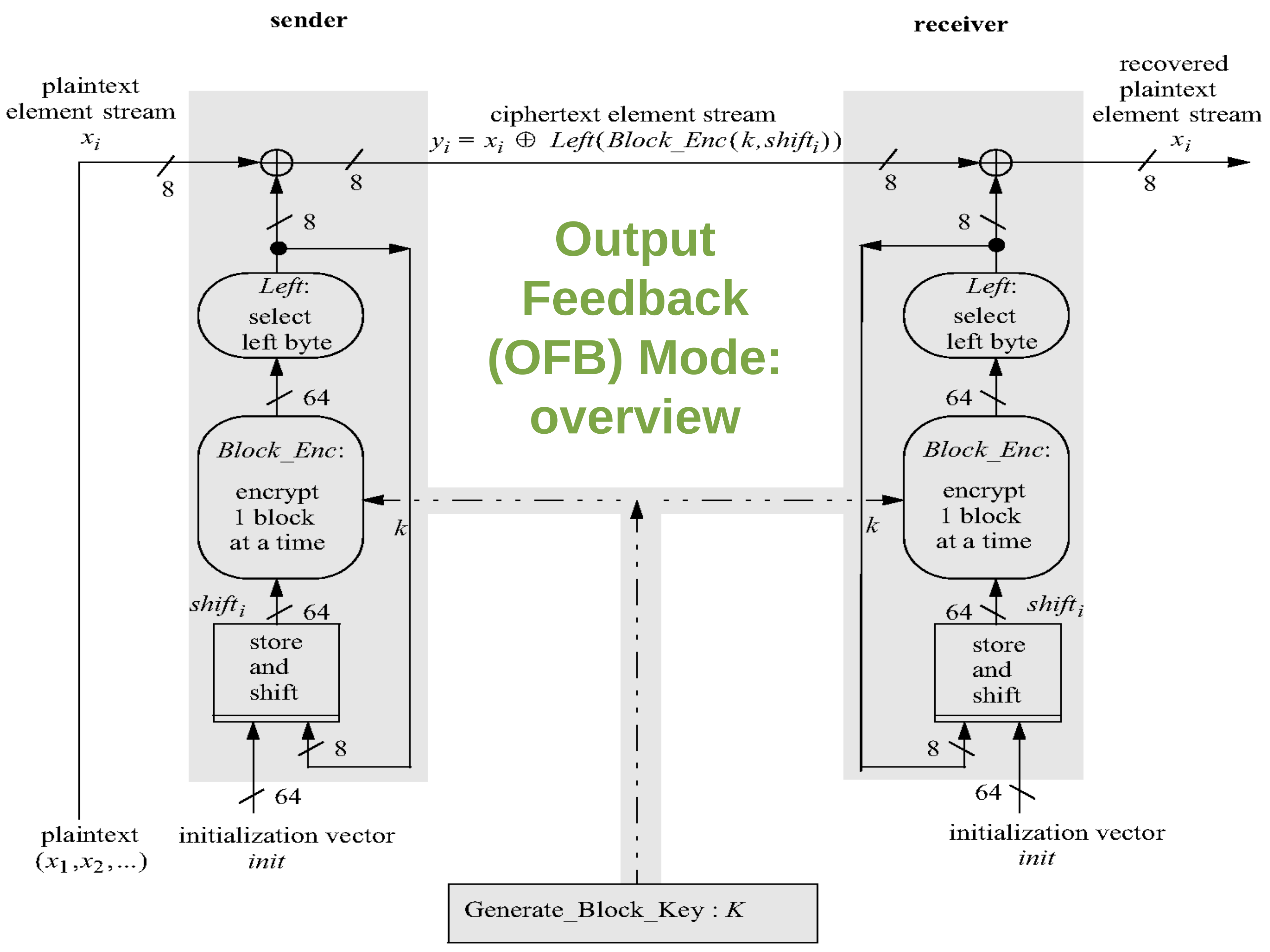
- characteristic feature of the cipher feedback mode:
  the last resulting ciphertext block
  depends potentially on the full plaintext stream

- the last resulting ciphertext block seen as a *message digest*:
  this block can be employed as a
  piece of *cryptographic evidence* (a *cryptographic exhibit*)
  for an *authenticity verification* algorithm

Output Feedback (OFB) Mode

- follows the second basic approach
- required pseudorandom *cipher key* stream is generated as for the cipher feedback mode, except of the following
- the block cipher encryption takes the feedback directly from its own outputs
- since only the encryption algorithm of the underlying block cipher is involved, this mode cannot be used for an asymmetric block cipher
- example:
  - fragment length: $l = 8$
  - block size of the underlying block cipher: $l_B = 64$

Output Feedback (OFB) Mode: overview



Counter-with-Cipher-Block-Chaining Mode (CCM)

- generates a pseudorandom *cipher key* stream by encrypting a sequence of *counters* $count_i$ using the underlying block encryption
- computes the counters by

  $$count_i := init + i \bmod 2^{l_B},$$

  assuming a block size l_B of the block cipher and taking an *initialization vector init* of that size
- cannot be used for an asymmetric block cipher
- exploits that for each $i = 1, 2, \ldots$:
  - the pair of the counter $count_i$ and the corresponding plaintext block x_i can be treated independently of all other pairs, as for ECB
  - the counter $count_i$ is independent of the ciphertext stream (and thus of the plaintext stream), as for OFB

Counter-with-Cipher-Block-Chaining Mode (CCM)

- achieves *authenticated encryption*:
  - additionally performs CBC encryption without transmitting the resulting ciphertext blocks
  - superimposes the last resulting CBC ciphertext block y_{fin} on the counter $count_0 = init$
  - appends the resulting block $y_{fin} \oplus count_0$ as a *message digest*

Features of block modes

- *initialization vector*:
  - some computational overhead is necessary
  - a *parameterization* of the encryption is achieved:
    if the initialization vector is varied for identical messages and kept secret,
    then the encryption could even be seen as *probabilistic*
- *fault tolerance*, for the sake of *availability*:
  propagation of a *modification error* is considered:
  - in the *plaintext* stream
  - during transmission, in the *ciphertext* stream:
    - all modes recover shortly after a modification error
    - OFB and CCM even behave optimally
      (only the error position is affected)

Features of block modes

- *modification error* in the *plaintext* stream:
  - ECB, OFB and the main part of CCM recover
    shortly after the error position or totally prevent propagation
  - for CBC, CFB and the digest production part of CCM,
    an error might "diffuse" through the full succeeding cipher stream:
    accordingly, the resulting final cipher block can be seen as a
    message digest and can thus be employed as
    a piece of *cryptographic evidence* (a *cryptographic exhibit*)

- *synchronization errors* owing to *lost fragments*:

  for all modes, additional measures must be employed, e.g.,
  by suitably inserting separators at agreed fragment borders

Rudimentary comparison of block modes

	ECB	CBC	CFB	OFB	CCM
Initialization vector / parameterization	no	yes	yes	yes	yes
Propagation of error in plaintext fragment	limited to block	unlimited up to end of stream	unlimited up to end of stream	limited to error position	limited to error position, except for superimposed last CBC cipher block
Suitable for producing a message digest	no	by last cipher block	by last cipher block	no	by superim-posed last CBC cipher block
Propagation of error in ciphertext fragment	limited to block	limited to block and succeeding block	limited to block and succeeding block	limited to error position	limited to error position

Some rough advice to a security administrator

- **electronic codebook mode**
  is suitable for short, randomly selected messages
  such as nonces or cryptographic keys of another mechanism

- **cipher block chaining mode**
  might be employed for long files with any non-predictable content

- **cipher feedback mode, output feedback mode and counter mode**
  support the transmission of a few bits or bytes,
  e.g., as needed for connections between a central processing unit
  and external devices such as a keyboard and monitor

- **output feedback mode and counter mode**
  might be preferred for highly failure-sensitive applications,
  since modification errors are not propagated at all
  (except for the added message digest)