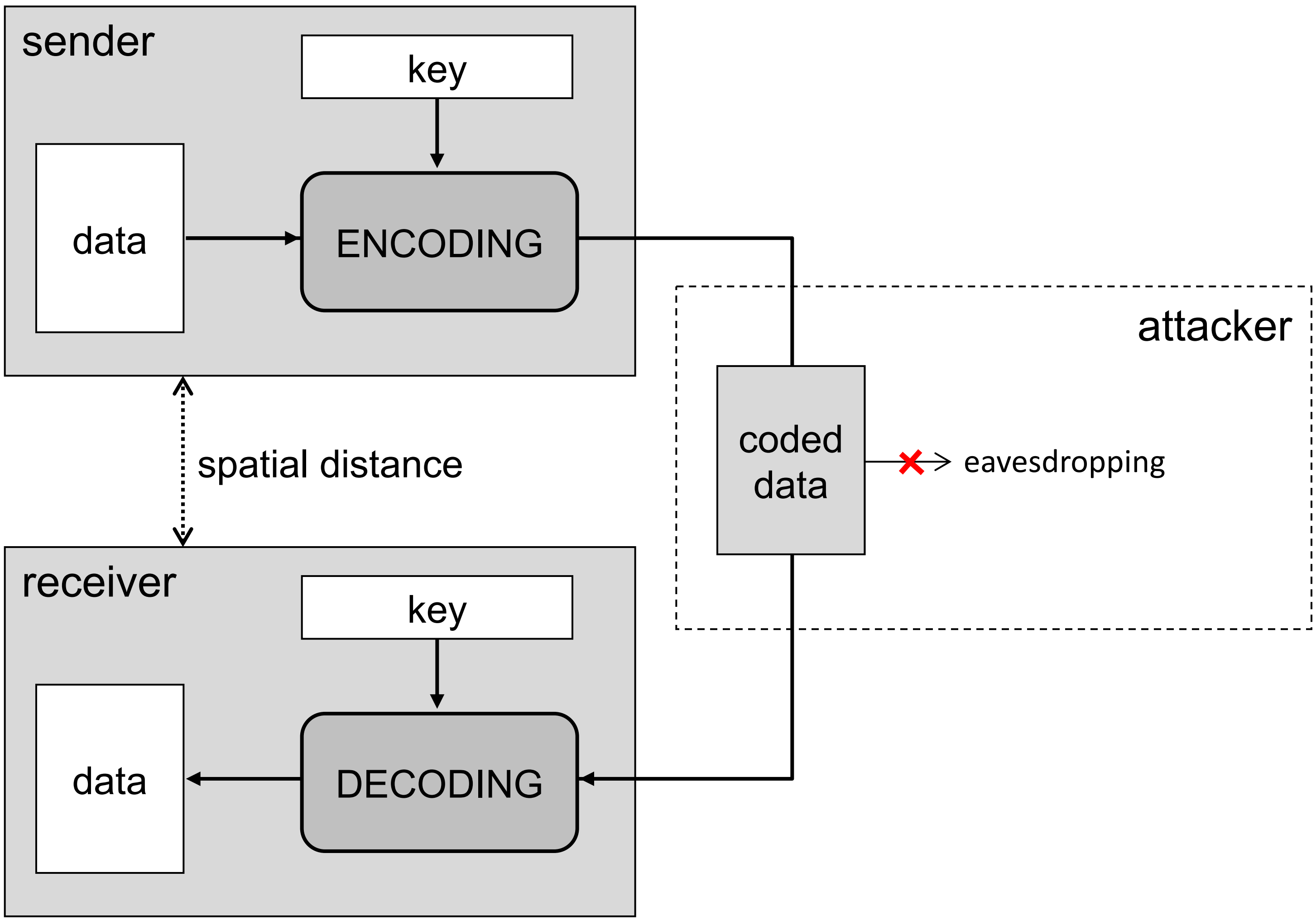
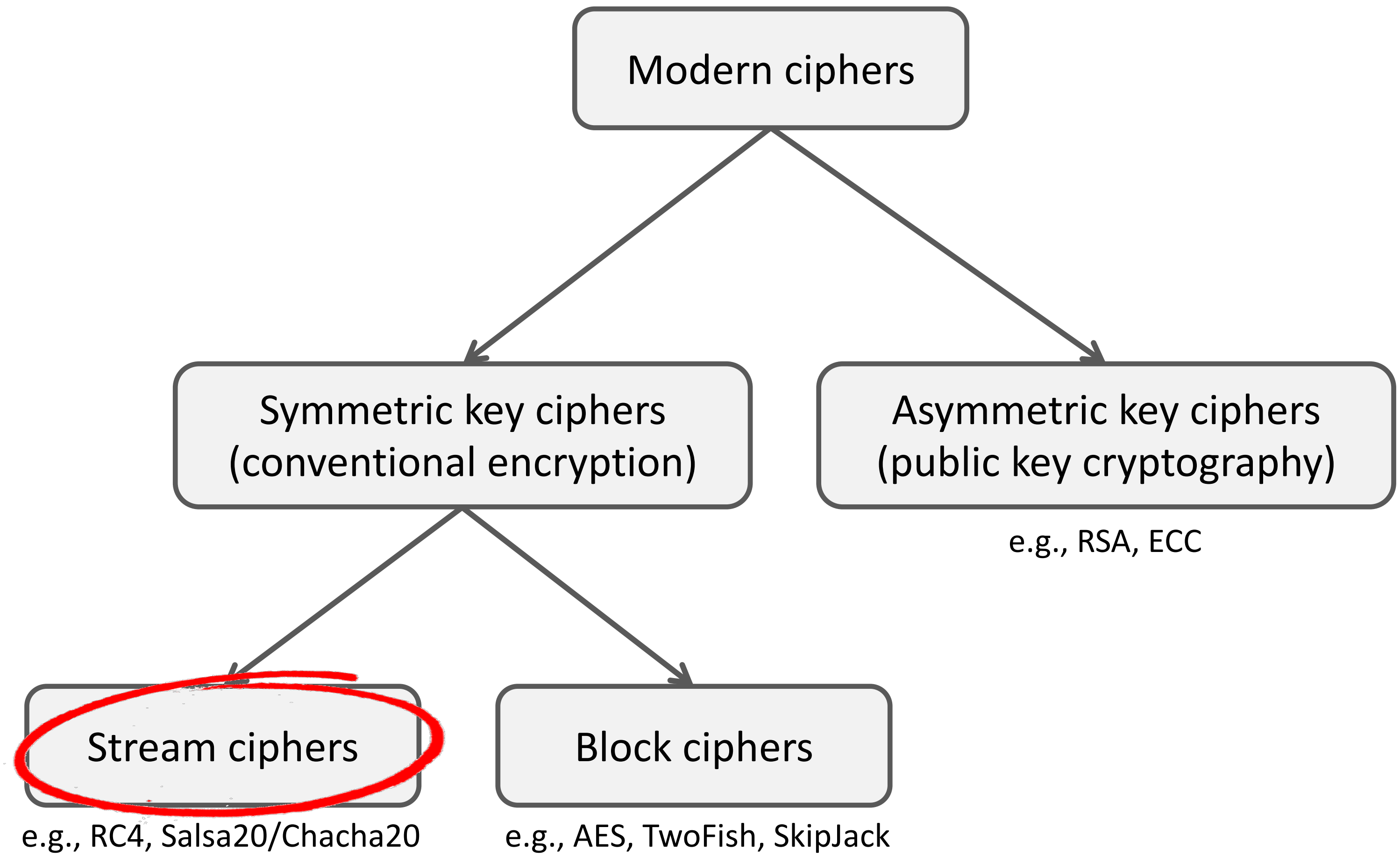
Symmetric Key Encryption
Stream Ciphers

Levente Buttyán
CrySyS Lab, BME
buttyan@crysys.hu

Basic model of symmetric key encryption



Classification of ciphers



Data representation and encodings

- the form in which data is stored, processed, and transmitted is called data representation
- at the lowest level, data is represented as bits and bytes
  - 1 byte = 8 bits
- at higher levels, we use different encodings for
  - numbers
    - unsigned integers
    - signed integers
    - floating-point numbers
  - text (string of characters)
  - binary data (string of arbitrary bytes)

Bits and bytes

- bits

  `10010011010110100001011101101110`

- bytes

  `10010011   01011010   00010111   01101110`

- hexadecimal notation

  `93 5A 17 6E`

  explanation:
  - 16 hexadecimal digits: 0 1 2 3 4 5 6 7 8 9 A B C D E F
  - 01011010 → 0101 1010 (half-bytes or nibbles)
  - 0101 → 0*8 + 1*4 + 0*2 + 1*1 = 5 → 5
  - 1010 → 1*8 + 0*4 + 1*2 + 0*1 = 10 → A
  - 10101010 → 5A

Numbers

- unsigned integers
  - positive integers (and 0)
  - typically encoded directly
    - e.g., 53 → `00110101`
- signed integers
  - positive and negative integers (and 0)
  - different encodings
    - signed-magnitude
    - ones' complement
    - two's complement
    - …

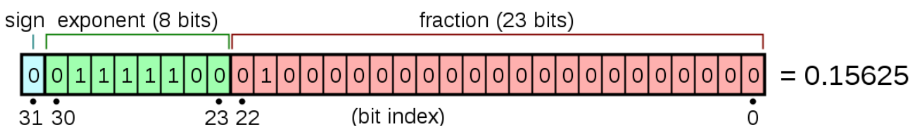
Binary	Unsigned	Sign-magnitude	Ones' complement	Two's complement
`00000000`	0	+0	+0	0
`00000001`	1	1	1	1
…	…	…	…	…
`01111110`	126	126	126	126
`01111111`	127	127	127	127
`10000000`	128	-0	-127	-128
`10000001`	129	-1	-126	-127
…	…	…	…	…
`11111110`	254	-126	-1	-2
`11111111`	255	-127	-0	-1

- integers are typically represented on 4 bytes in modern computers
  - e.g., 53 → `00000000 00000000 00000000 00110101`
  - unsigned range: 0 through 4,294,967,295 (2^{32} - 1)
  - two's complement range: -2,147,483,648 (-2^{31}) through 2,147,483,647 (2^{31} - 1)

Numbers

- floating-point numbers
  - can be non-integers (fractions)
  - different encodings…

    <u>example:</u> IEEE 754 single-precision binary floating-point format



The real value assumed by a given 32-bit *binary32* data with a given *sign*, biased exponent *e* (the 8-bit unsigned integer), and a *23-bit fraction* is

$$(-1)^{b_{31}} \times 2^{(b_{30}b_{29}\dots b_{23})_2 - 127} \times (1.b_{22}b_{21}\dots b_0)_2,$$

which yields

$$\text{value} = (-1)^{\text{sign}} \times 2^{(E-127)} \times \left(1 + \sum_{i=1}^{23} b_{23-i}2^{-i}\right).$$

In this example:

- $\text{sign} = b_{31} = 0,$
- $(-1)^{\text{sign}} = (-1)^0 = +1 \in \{-1, +1\},$
- $E = b_{30}b_{29}\dots b_{23} = \sum_{i=0}^{7} b_{23+i}2^{+i} = 124 \in \{1, \dots, (2^8-1)-1\} = \{1, \dots, 254\},$
- $2^{(E-127)} = 2^{124-127} = 2^{-3} \in \{2^{-126}, \dots, 2^{127}\},$
- $1.b_{22}b_{21}\dots b_0 = 1 + \sum_{i=1}^{23} b_{23-i}2^{-i} = 1 + 1 \cdot 2^{-2} = 1.25 \in \{1, 1+2^{-23}, \dots, 2-2^{-23}\} \subset [1; 2-2^{-23}] \subset [1; 2).$

thus:

- $\text{value} = (+1) \times 2^{-3} \times 1.25 = +0.15625.$

Source: https://en.wikipedia.org/wiki/Single-precision_floating-point_format

Text

- string of encoded characters
- different encodings
  - ASCII - American Standard Code for Information Interchange
    - » characters are encoded on 7 bits (use 1 byte with MSB = `0`)
    - » allows for a limited set of characters only
  - Unicode
    - » Unicode codepoints can represent a huge number of characters, symbols, emoji, and non-visual control and formatting codes
    - » codepoints can be encoded using different encoding standards, like UTF-8 and UTF-16 (where UTF stands for Unicode Transformation Format)
      - UTF-8: variable length encoding on 1, 2, 3, or 4 bytes (used by Linux)
      - UTF-16: variable length encoding on 2 or 4 bytes (used by Windows)

ASCII code table

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

UTF-8

- first 128 characters are encoded on 1 byte

  → compatible with ASCII

- next 1,920 characters are encoded on 2 bytes
  - they cover the remainder of almost all Latin-script alphabets, and also phonetic extensions, Greek, Cyrillic, Hebrew, Arabic, …
- Chinese, Japanese, Korean, ... characters are encoded on 3 bytes
- all the rest is encoded on 4 bytes

Code point <-> UTF-8 conversion

First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
U+0000	U+007F	0xxxxxxx			
U+0080	U+07FF	110xxxxx	10xxxxxx		
U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
U+10000	[nb 2]U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Source: https://en.wikipedia.org/wiki/UTF-8

Binary data

- raw binary

  <u>example:</u>

  ```
  41 70 70 6c 69 65 64 20 43 72
  79 70 74 6f 67 72 61 70 68 79
  ```

- Base64 encoding
  - converts raw binary data to a printable ASCII string
    - » splitting binary to 6-bit chunks
    - » replacing 6-bit chunks with characters of a pre-defined set
    - » padding to cope with potential short chunks at the end

  <u>example:</u>

  QXBwbGllZCBDcnlwdG9ncmFwaHk=

Index	Binary	Char	Index	Binary	Char	Index	Binary	Char	Index	Binary	Char
0	000000	A	16	010000	Q	32	100000	g	48	110000	w
1	000001	B	17	010001	R	33	100001	h	49	110001	x
2	000010	C	18	010010	S	34	100010	i	50	110010	y
3	000011	D	19	010011	T	35	100011	j	51	110011	z
4	000100	E	20	010100	U	36	100100	k	52	110100	0
5	000101	F	21	010101	V	37	100101	l	53	110101	1
6	000110	G	22	010110	W	38	100110	m	54	110110	2
7	000111	H	23	010111	X	39	100111	n	55	110111	3
8	001000	I	24	011000	Y	40	101000	o	56	111000	4
9	001001	J	25	011001	Z	41	101001	p	57	111001	5
10	001010	K	26	011010	a	42	101010	q	58	111010	6
11	001011	L	27	011011	b	43	101011	r	59	111011	7
12	001100	M	28	011100	c	44	101100	s	60	111100	8
13	001101	N	29	011101	d	45	101101	t	61	111101	9
14	001110	O	30	011110	e	46	101110	u	62	111110	+
15	001111	P	31	011111	f	47	101111	v	63	111111	/
Padding		=									

Source: https://en.wikipedia.org/wiki/Base64

Decoding

`3e 20 00 00` ← What is this?

Depends on how you interpret and decode…

- signed or unsigned integer: 1042284544
- single-precision floating-point number: 0.15625
- ASCII or UTF-8 text: ‘> ’ (i.e., ‘>’ + ‘ ’ (space) + 2 terminating 0’s)
- UTF-16 text: ‘‾’ + a terminating 0
- raw binary: `3e 20 00 00`

→ a useful on-line tool: https://kt.gy/

XOR operation

- XOR (+ or $\oplus$)
  - `0+0 = 0; 0+1 = 1; 1+0 = 1; 1+1 = 0`

- XOR of bit vectors (words)
  - we XOR each corresponding bit pairs
  - e.g., `0011 + 1010 = 1001`

- <u>exercise:</u>
  - `1101 + 1001 = ____ ?`
  - `1010 + 0001 = ____ ?`
  - `1101 + 1101 = ____ ?`
  - `1010 + 1111 = ____ ?`

Main properties to remember

1. $X + 0 = 0 + X = X$
2. $X + X = 0$
3. if $X + Y = Z$, then $X = Y + Z$ (and $Y = X + Z$)

XOR-ing hex strings

1. convert them to binary
2. XOR the binary values
3. convert back the binary result to hex

examples:

» `xB + x9 = b1011 + b1001 = b0010 = x2`

» `xC + xC = ____?`

» `xDEAD + xBEEF = ____?`

... or use a program or on-line tool:

– desktop calculator
– on-line calculator: https://cryptii.com/pipes/bitwise-calculator
– on-line XOR tool: https://xor.pw/

Simple XOR cipher

- encryption
  - represent the plaintext as a sequence of bytes (e.g., using ASCII encoding)
  - take a password and repeat it many times to get a byte string as long as the plaintext
  - obtain the ciphertext by XOR-ing together the plaintext and the password string

```
Lorem ipsum dolor sit amet, eu p
rima euismod mediocritatem sea,
sint aliquip est te, et quot sae
pe omittam sit. Id vel malis sum
mo dolores, pro odio dolorum ei.
 Eam inimicus tractatos partiend
o te, ex eum equidem delicata pr
incipes. Error conceptam vel ea,
 salutatus delicatissimi vituper
atoribus ut eam. Nam ne animal e
xpetenda, vide ubique convenire
qui ut. Ne aeque gloriatur nam,
sed alterum inimicus dissentias
te. Vel te cibo tibique.
```

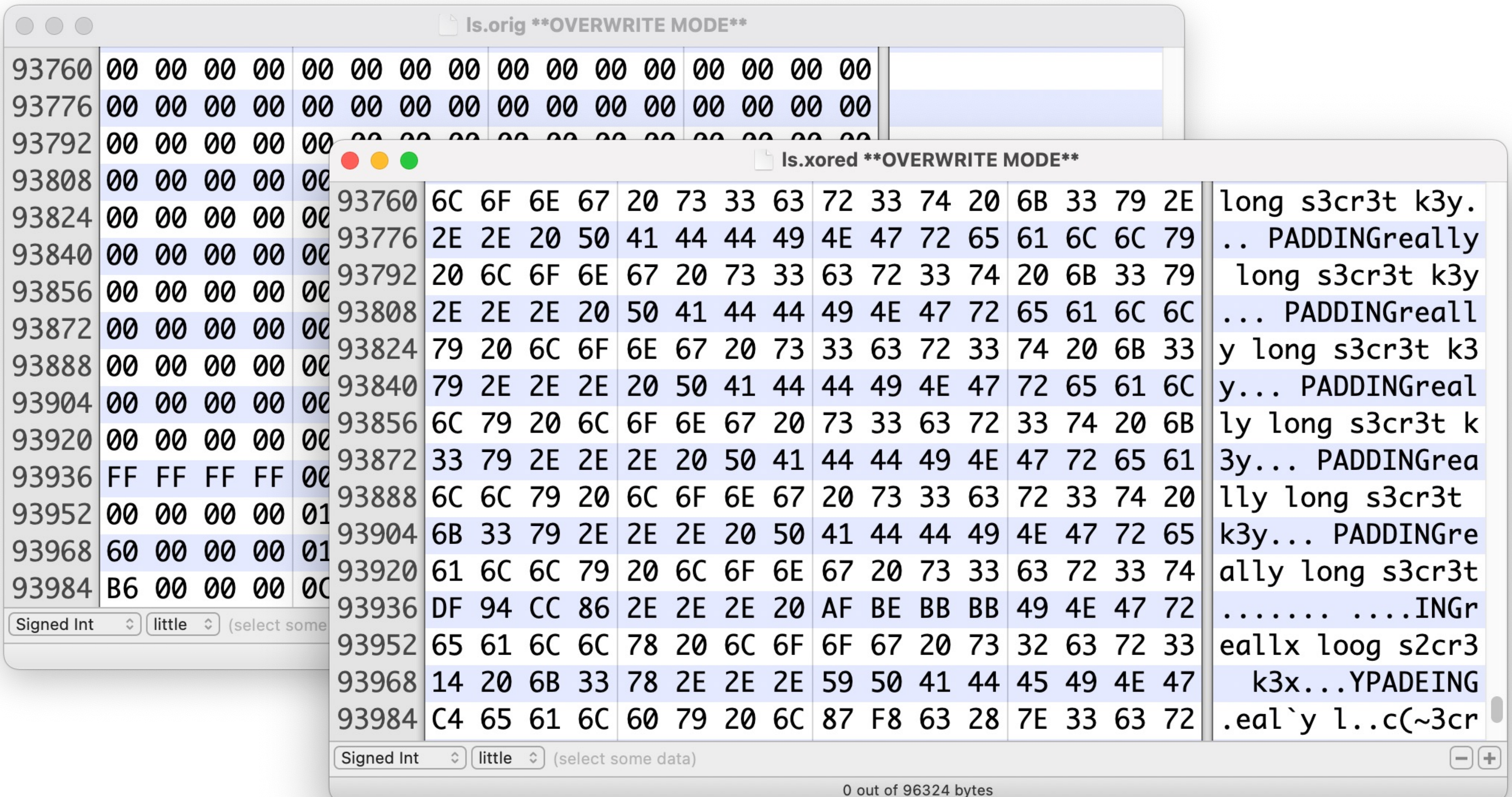
$\oplus$

```
TitanTitanTitanTitanTitanTitanTi
tanTitanTitanTitanTitanTitanTita
nTitanTitanTitanTitanTitanTitanT
itanTitanTitanTitanTitanTitanTit
anTitanTitanTitanTitanTitanTitan
TitanTitanTitanTitanTitanTitanTi
tanTitanTitanTitanTitanTitanTita
nTitanTitanTitanTitanTitanTitanT
itanTitanTitanTitanTitanTitanTit
anTitanTitanTitanTitanTitanTitan
TitanTitanTitanTitanTitanTitanTi
tanTitanTitanTitanTitanTitanTita
nTitanTitanTitanTitanTitanTitanT
itanTitanTitanTitanTitanT
```

- decryption
  - XOR the same password string to the ciphertext to recover the plaintext

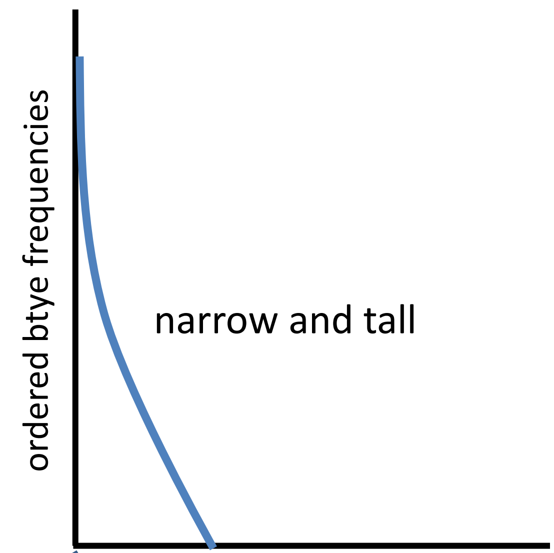
Breaking the simple XOR cipher

Sometimes, it can be trivial …



Breaking the simple XOR cipher

```
Lorem ipsum dolor sit amet, eu p
rima euismod mediocritatem sea, 
sint aliquip est te, et quot sae
pe omittam sit. Id vel malis sum
mo dolores, pro odio dolorum ei.
 Eam inimicus tractatos partiend
o te, ex eum equidem delicata pr
incipes. Error conceptam vel ea,
 salutatus delicatissimi vituper
atoribus ut eam. Nam ne animal e
xpetenda, vide ubique convenire 
qui ut. Ne aeque gloriatur nam, 
sed alterum inimicus dissentias 
te. Vel te cibo tibique.
```



bytes occuring in the text

most frequent char is ' ' (space)
ASCII code of ' ' is `x20`

Breaking the simple XOR cipher

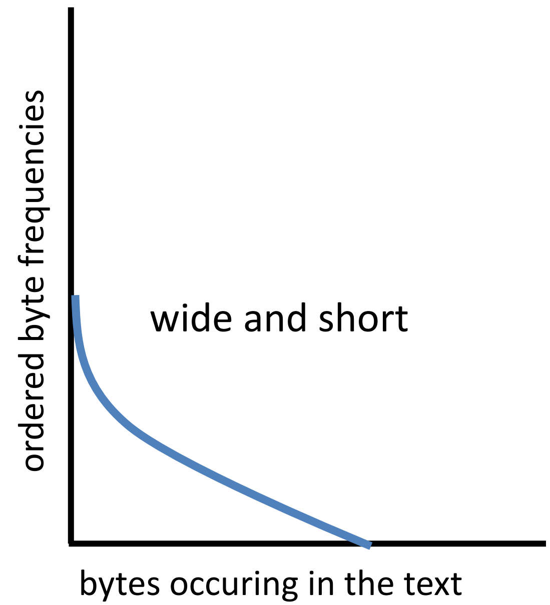
```
Titan …

⊕

Lorem ipsum dolor sit amet, eu p
rima euismod mediocritatem sea, 
sint aliquip est te, et quot sae
pe omittam sit. Id vel malis sum
mo dolores, pro odio dolorum ei.
 Eam inimicus tractatos partiend
o te, ex eum equidem delicata pr
incipes. Error conceptam vel ea,
 salutatus delicatissimi vituper
atoribus ut eam. Nam ne animal e
xpetenda, vide ubique convenire 
qui ut. Ne aeque gloriatur nam, 
sed alterum inimicus dissentias 
te. Vel te cibo tibique.
```

Breaking the simple XOR cipher

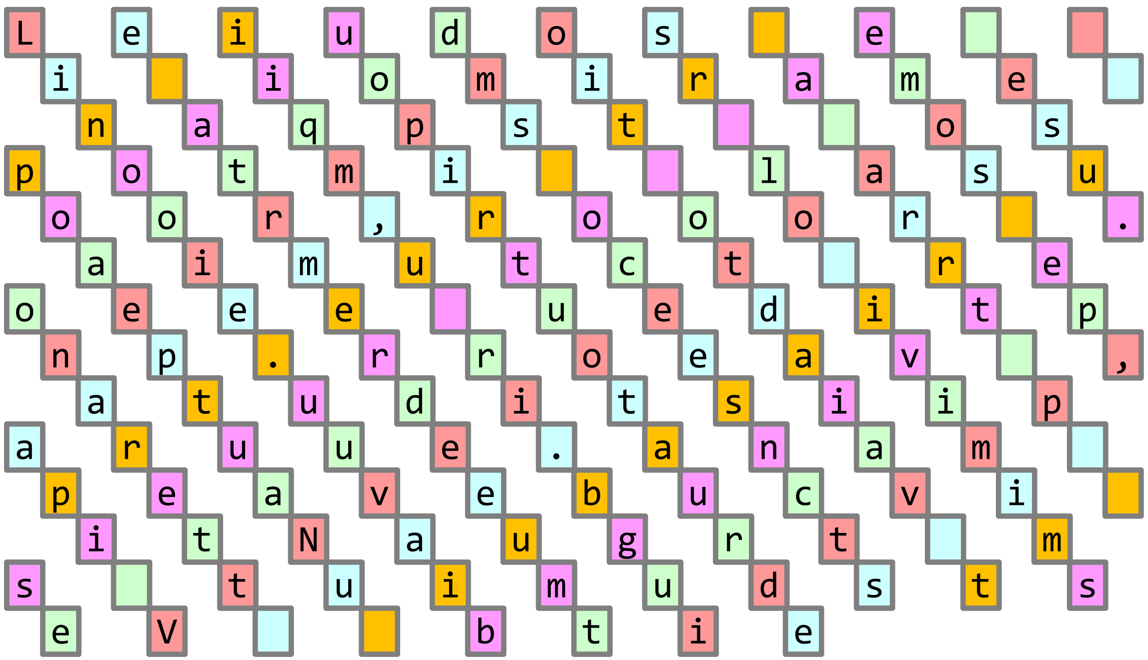
Let’s determine the length of the key ...

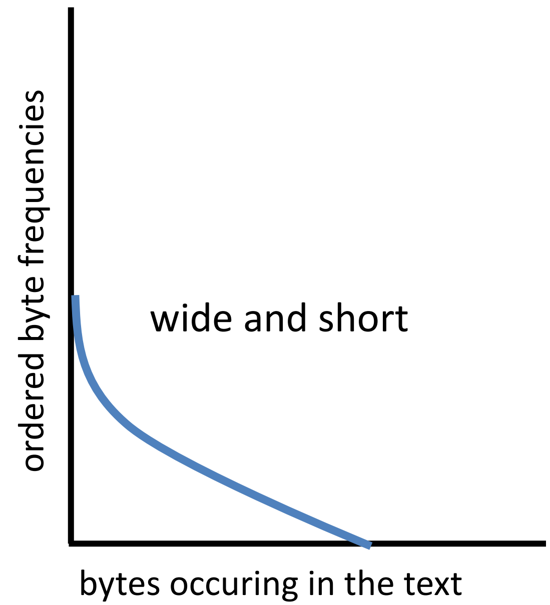


Breaking the simple XOR cipher

Let's determine the length of the key ...

e.g., keeping every 3rd byte...

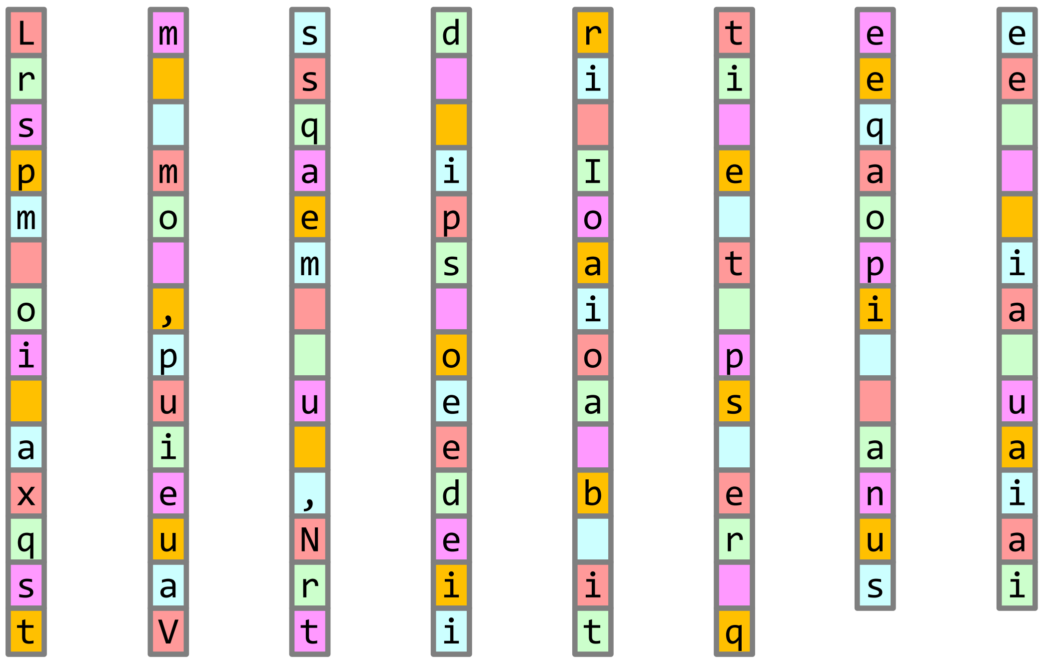


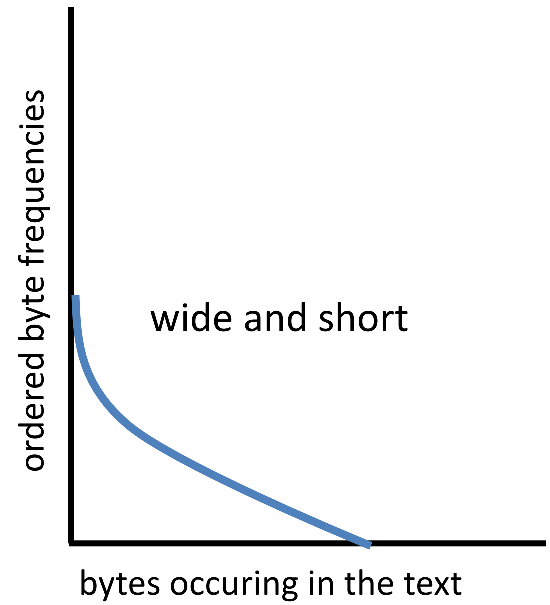


Breaking the simple XOR cipher

Let’s determine the length of the key ...

e.g., keeping every 4th byte...

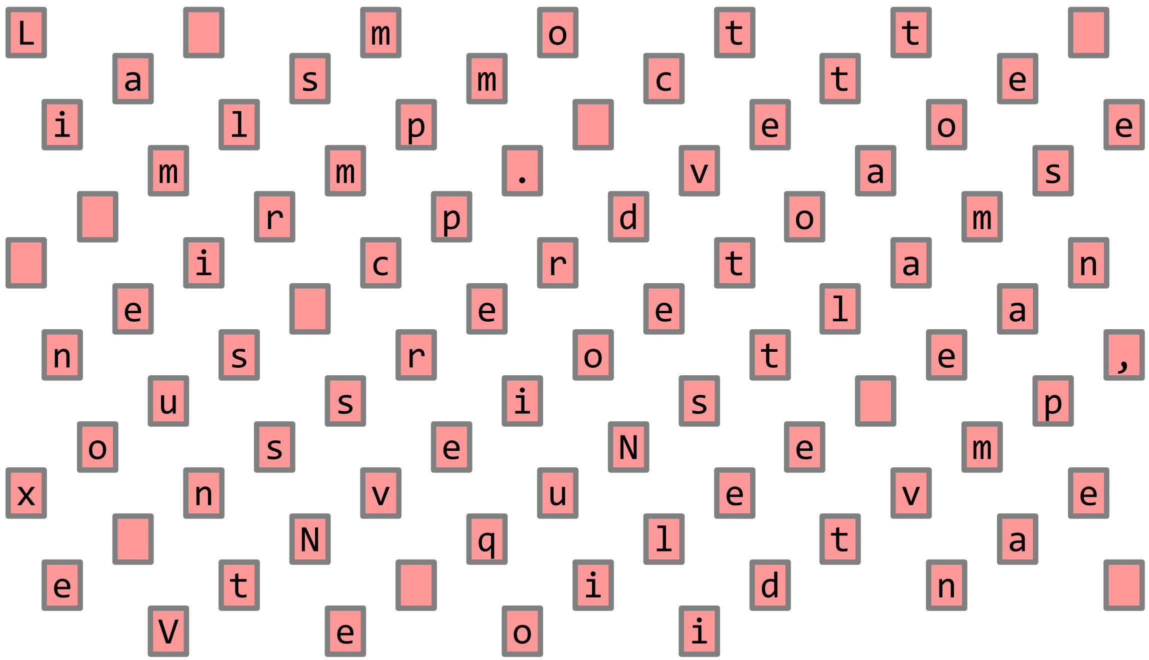


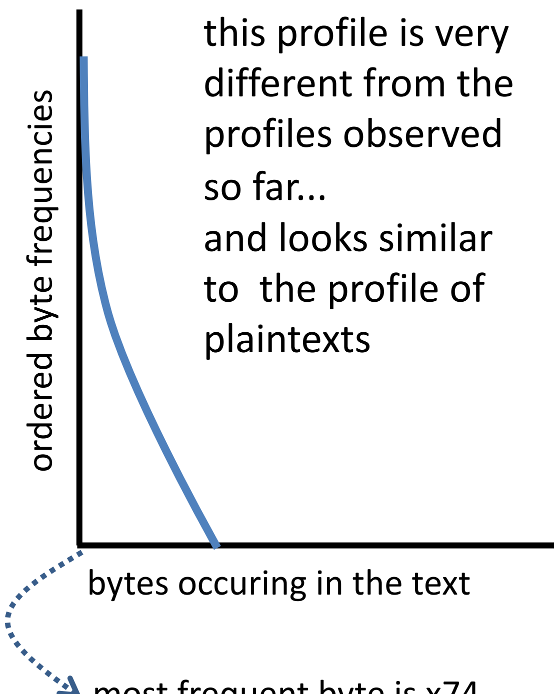


Breaking the simple XOR cipher

Let’s determine the length of the key ...

e.g., keeping every 5th byte...





this profile is very different from the profiles observed so far...
and looks similar to the profile of plaintexts

most frequent byte is x74

Let’s determine the characters of the key ...

’ ’ + $\boxed{}$ = x20 + $\boxed{}$ = x74

$\boxed{}$ = x74 + x20 = x54 = ’T’

One-time pad

- encryption
  - represent the message as a sequence of bytes: $m_1\ m_2\ ...\ m_L$
  - take a sequence of true random bytes: $k_1\ k_2\ ...\ k_L$
  - obtain the encrypted message by XOR-ing them together:

    $m_1\ m_2\ ...\ m_L + k_1\ k_2\ ...\ k_L = c_1\ c_2\ ...\ c_L$ where $c_i = m_i + k_i$ for all $i = 1, ..., L$

- decryption
  - XOR the same stream of key bytes to the encrypted message:

    $c_1\ c_2\ ...\ c_L + k_1\ k_2\ ...\ k_L = m_1\ m_2\ ...\ m_L$ (because $c_i + k_i = m_i + k_i + k_i = m_i$)

Properties of the one-time pad

- **perfect secrecy**
  - informally: observing the encrypted message provides no information (in an information theoretic sense) about the original message
  - illustration
    - » let the clear message be: `x41` (ASCII code for letter 'A')
    - » let the key be: `xAD`
    - » encrypted message observed by the attacker: `xEC` `(x41 + xAD = xEC)`
    - » from the attacker's point of view, the original message may be:
      - `x00` if the key was `xEC`
      - `x01` if the key was `xED`

        ...
      - `x41` ('A') if the key was `xAD`
      - `x42` ('B') if the key was `xAE`
      - `x43` ('C') if the key was `xAF`
      - ...
      - `xFF` if the key was `x13`

all these cases have equal probability, because the key is chosen randomly

Properties of the one-time pad

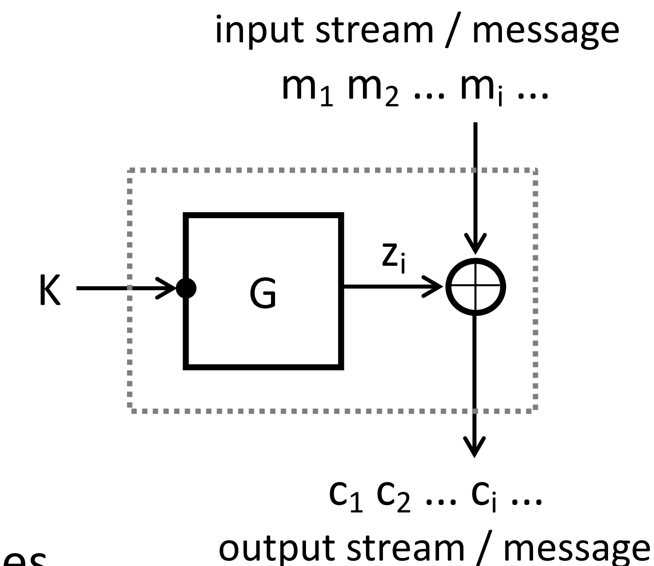
- large key size
  - needs a truly random key that has the same length as the (compressed) message

- impractical in many applications
  - how to send the key in a secure way to the recipient?
  - in practice, the only possibility is to exchange a large amount of truly random key material in an *out-of-band* manner (e.g., by physical meeting, via a quantum channel) before the communication takes place
  - we have to do this with all potential communication partners
  - key management becomes cumbersome

General model of stream ciphers

- idea: simulate the truly random key stream of the one-time pad with a pseudo-random sequence generated from a random seed

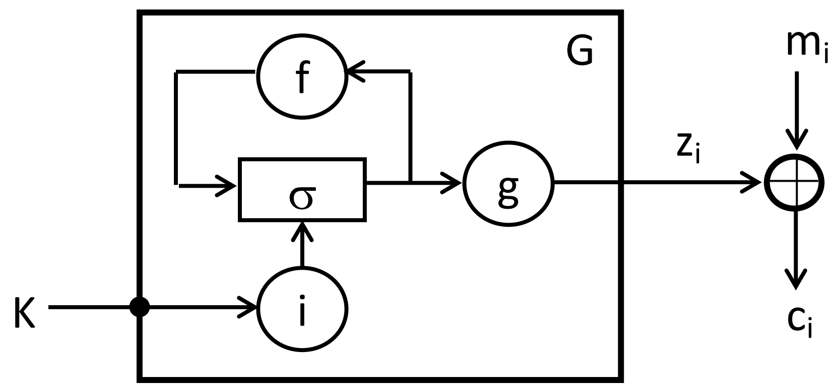
- terminology:
  - m_i – plaintext character
  - c_i – ciphertext character
  - z_i – key-stream character
  - K – key (seed)
  - G – key-stream generator



- application:
  - encryption of data → confidentiality services
  - PRNG (Pseudo-Random Number Generator)

- examples:
  - LFSR based (hardware), RC4, A5 (GSM), E0 (Bluetooth), Salsa20/Chacha

Inside the key stream generator

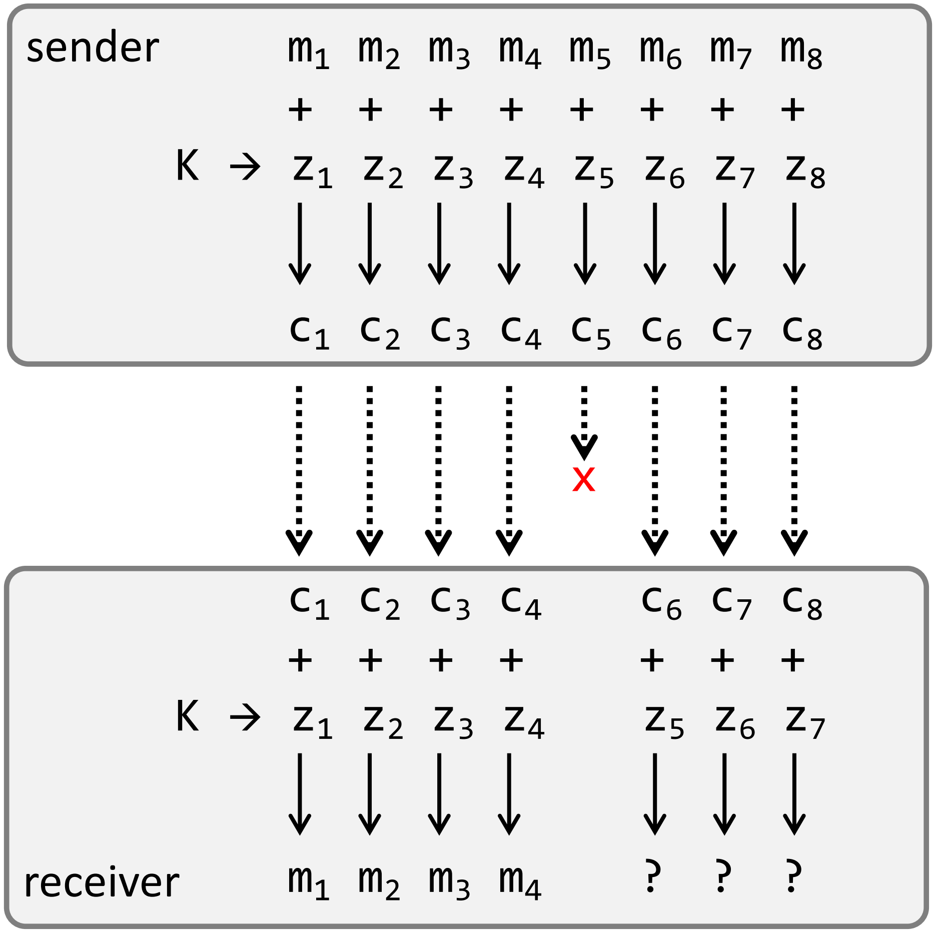


- the key stream is generated independently of the plaintext and of the ciphertext
  - key K is used to initialize an internal state σ (seed the generator)
  - once initialized, the state is updated using some function f after each step
  - the output is generated from the state with another function g
- effective size of the state space must be large
  - otherwise, the key stream starts repeating
  - if two plaintext characters m_i and m_j are encrypted with the same key character z, then the XOR sum of the ciphertext characters $(m_i + z) + (m_j + z) = m_i + m_j$

Other properties of stream ciphers

- stream ciphers are usually very efficient
  - fast (especially in hardware)
  - require small memory to store the internal state and the code of the generation and state update functions
- the ciphertext always has the same length as the plaintext (in some block encryption modes, the ciphertext is longer)
- synchronization is needed between the sender and the receiver
  - loss of synchrony needs to be detected and addressed
- stream ciphers do not provide any integrity protection !!!
  - an attacker can make changes to selected ciphertext characters and know exactly what effect these changes have on the plaintext
  - the receiver may not notice these changes

Loss of synchrony illustrated



$$c_6 + z_5 = m_6 + z_6 + z_5 \neq m_6$$

$$c_7 + z_6 = m_7 + z_7 + z_6 \neq m_7$$

…

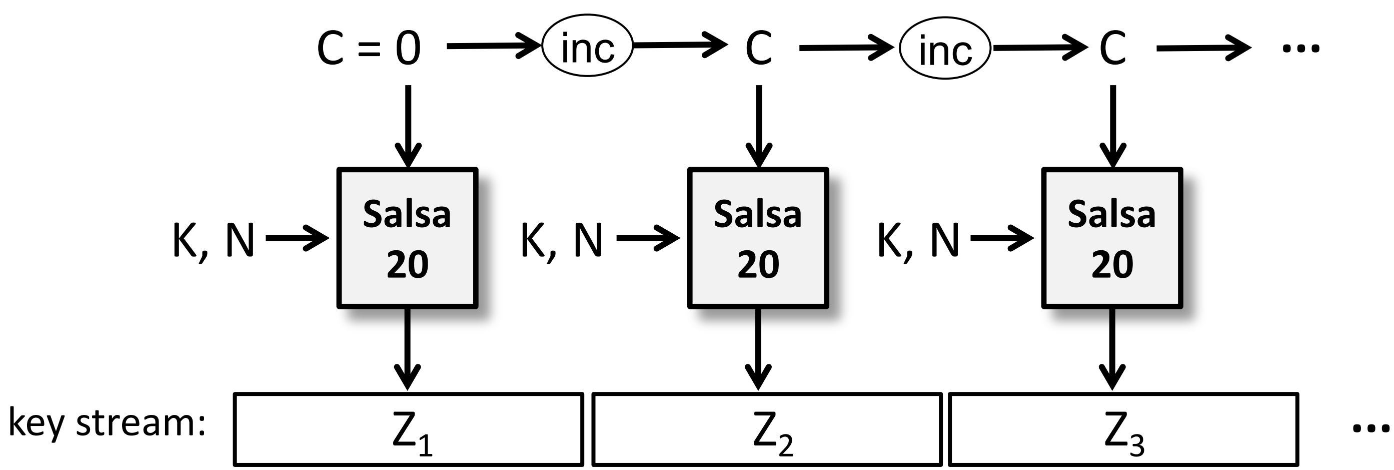
Controlled modification of the plaintext

- an attacker can add any value Δ to a ciphertext character c_i
- the receiver will decode: $(c_i + \Delta) + z_i = (m_i + z_i + \Delta) + z_i = m_i + \Delta$
- hence, the plaintext character will be modified by Δ
  - bits of m_i will be flipped in every position where Δ has a bit 1
  - bits of m_i will be unchanged in every position where Δ has a bit 0

$$
\begin{array}{rl}
m_i = & 00101100 \\
 & ++++++++ \\
\Delta = & 10000100 \\
\hline
m_i' = & 10101000
\end{array}
$$

Salsa20

- produces 512 pseudo-random bits in one step from
  - a 256-bit key K
  - a 64-bit nonce N (unique to each message)
  - a 64-bit counter C



Salsa20

- internal state:
  - 16 words $S_{0..15}$ of 32 bits each arranged in a 4x4 matrix

S_0	S_1	S_2	S_3
S_4	S_5	S_6	S_7
S_8	S_9	S_{10}	S_{11}
S_{12}	S_{13}	S_{14}	S_{15}

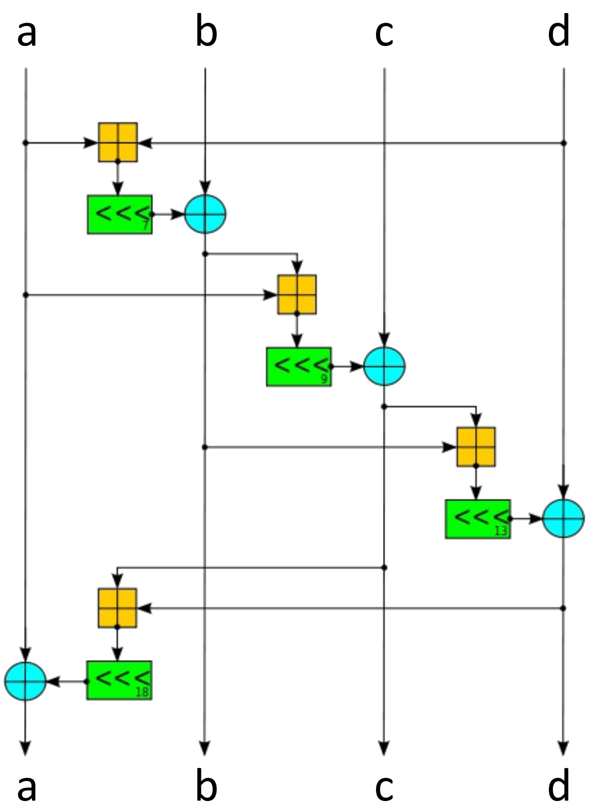
- initial state:

A_0	K_0	K_1	K_2
K_3	A_1	N_0	N_1
C_0	C_1	A_2	K_4
K_5	K_6	K_7	A_3

- $K_{0..7}$ is the key K
- $N_{0..1}$ is the nonce N
- $C_{0..1}$ is the counter C
- $A_{0..4}$ is a constant

Salsa20

- quarter-round (QR):



- Salsa20 block:
  - $S_{0..15}$ = initial_state
  - repeat 10 times:

    // Odd round
    QR(S_0, S_4, S_8, S_{12}) // column 1
    QR(S_5, S_9, S_{13}, S_1) // column 2
    QR(S_{10}, S_{14}, S_2, S_6) // column 3
    QR(S_{15}, S_3, S_7, S_{11}) // column 4
    // Even round
    QR(S_0, S_1, S_2, S_3) // row 1
    QR(S_5, S_6, S_7, S_4) // row 2
    QR(S_{10}, S_{11}, S_8, S_9) // row 3
    QR(S_{15}, S_{12}, S_{13}, S_{14}) // row 4

  - output = $S_{0..15}$ + initial_state

Summary

- representation of data and different encodings
- XOR operation on bits and bit vectors, 3 main properties of XOR
- hexadecimal numbers and ASCII codes of characters
- the simple XOR cipher and how to break it
- the one-time pad and its properties
  - perfect secrecy
  - the key must be as long as the plaintext (impractical)
- stream ciphers
  - try to simulate the one-time pad, use a pseudo-random key stream
  - general structure and operation
  - properties
    - » must have a large state space
    - » need for detecting loss of synchrony
    - » no integrity protection at all
  - examples: LFSR-based, RC4, Salsa20

Control questions

- What are the three main properties of the XOR operation?
- Why the simple XOR cipher be broken?
- How does the one-time pad work?
- What does perfect secrecy mean intuitively?
- What are the disadvantages of the one-time pad?
- How stream ciphers work? (general internal structure)
- Why should the state space of a stream cipher be large?
- What are the advantages of stream ciphers?
- What are the disadvantages of stream ciphers?