Lecture 6: Symmetric Cryptography

CS 181S

Spring 2024

The Big Picture Thus Far…

Attacks
are perpetrated by
threats
that inflict
harm
by exploiting
vulnerabilities
which are controlled by
countermeasures.

Dolev-Yao Threat Model (1983)

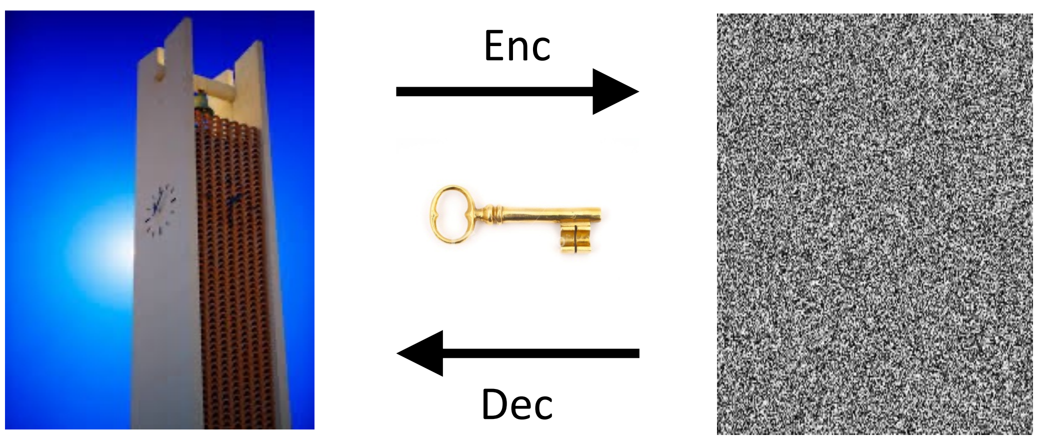
- Assume an attacker with network access and the following capabilities:
  - Can read all messages on the network
  - Can write messages to the network
  - Can block any messages sent over the network (i.e., cause them to be dropped)

Purpose of encryption

- **Threat:** Dolev-Yao attacker
- **Vulnerability:** communication channel between sender and receiver can be read by other principals
- **Harm:** messages containing secret information disclosed to attacker (violating confidentiality)
- **Countermeasure:** **encryption**

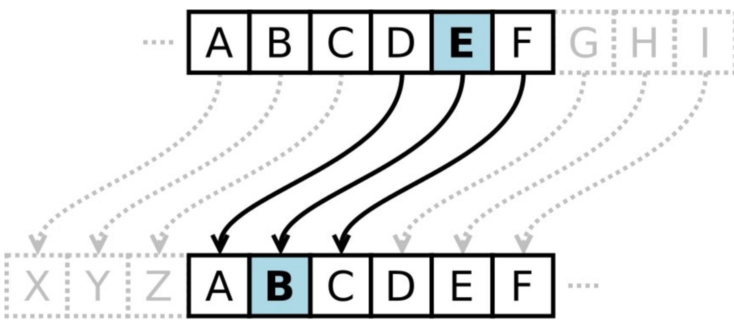
(Symmetric) Encryption algorithms

- $\text{Gen}(1^n)$: generate a **key** of length n
- $\text{Enc}(m; k)$: encrypt **message** under key k
- $\text{Dec}(m; k)$: decrypt **ciphertext** c with key k



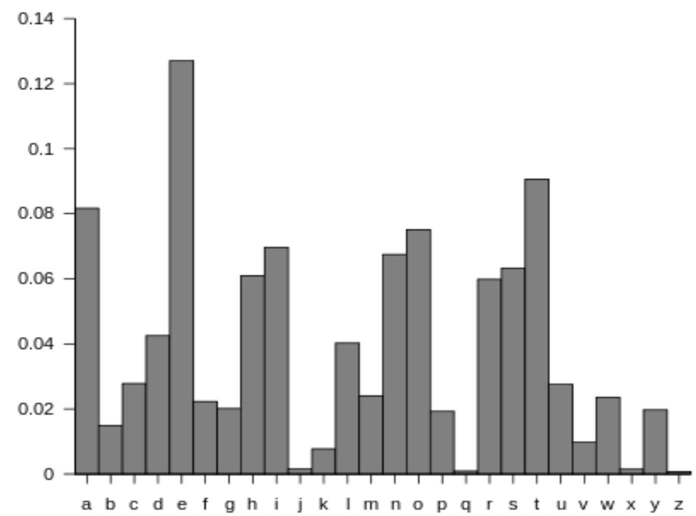
$(\text{Gen}, \text{Enc}, \text{Dec})$ is a symmetric-key **encryption scheme** aka **cryptosystem**

Classical Crypto: Substitution Ciphers



WKLV LV QRW VR VHFXUH

THIS IS NOT SO SECURE



Classical Crypto: Vigenere Cipher

```
THIS IS NOT SO SECURE
KEYK EY KEY KE YKEYKE
EMHD NR YTS DT RPHTCJ
```

Defining Security

- A crypto system is **secure** if

$$\forall \text{PPT } A, \exists \delta \in O(\frac{1}{2^n})\ s.t\ \forall n, \forall m, m' s.t. |m| = |m'| = n,$$

$$\Pr\left[A\left(Enc(m;k)\right) = m\right] \leq \Pr\left[A\left(Enc(m';k)\right) = m\right] + \delta(n)$$

One-Time Pad

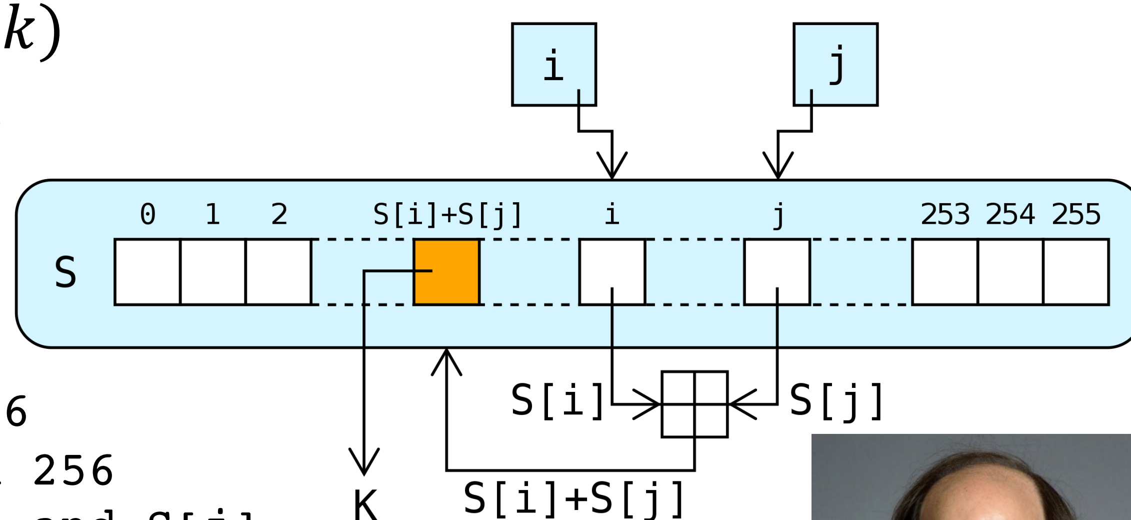
- $\mathrm{Gen}(1^n) :=$ generate a random bitstring of length n
- $\mathrm{Enc}(m; k) := m \oplus k$
- $\mathrm{Dec}(c; k) := c \oplus k$

plaintext	`THIS IS SECURE`
plaintext	01010100010010000100100101010011 …
key	01101010100101010100101000010110 …
ciphertext	00111110110111010000001101000101 …

- $\forall m, m' \text{ s.t. } |m| = |m'|,\ \Pr[m \mid c] = \left(\frac{1}{2}\right)^{\mathrm{len}(m)} = \Pr[m' | = c]$

Stream Ciphers: RC4

- $\mathrm{Gen}(1^n) \coloneqq$ generate a random bitstring of length n $\approx \mathbf{128}$
  use that to initialize permutation S of the 256 possible bytes
- $\mathrm{Enc}(m; k) \coloneqq m \oplus r(k)$
- $\mathrm{Dec}(c; k) \coloneqq c \oplus r(k)$

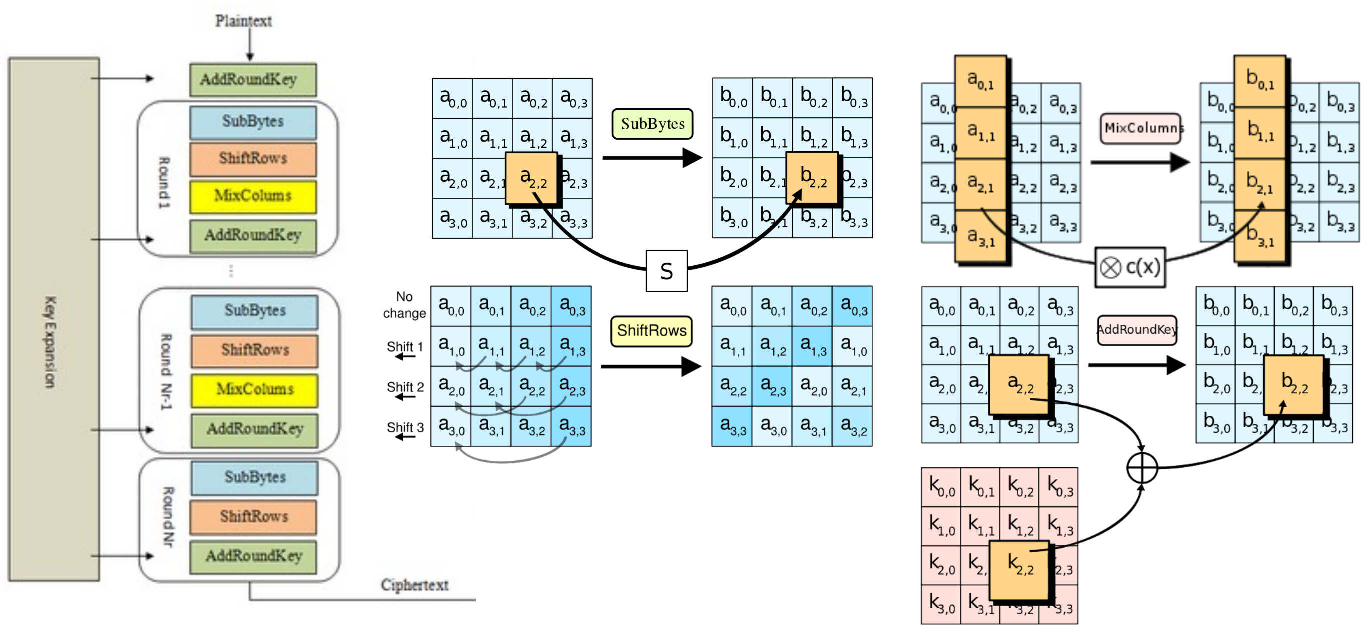



```
i := 0
j := 0
while True:
    i := (i + 1) mod 256
    j := (j + S[i]) mod 256
    swap values of S[i] and S[j]
    r := S[(S[i] + S[j]) mod 256]
output r
```

- Modern Alternative: ChaCha20

Block Ciphers: AES

- Encryption schemes that operate on fixed-size messages called **blocks**
- Advanced Encryption Standard (AES) result of 2001 NIST competition
- Currently no known practical attacks, approved by NSA for top-secret
- $\mathrm{Gen}(1^n) :=$ generate a random bitstring of length n



AES: Pre-processing

```
I have this thin
```

TODO: Generate ASCII encoding of each of these bytes

AES: Step 0 (Expand key)

a3d39ac91855c571b1ebe3894d5c4f47d7b8f762493f052d97f7ce8aeaf4c438

- AES key: random 256-bits
- Expand key to 240 bytes (1920 bits)

```
void expand_key(unsigned char *in) {
    unsigned char t[4];
    unsigned char c = 32;
    unsigned char i = 1;
    unsigned char a;
    while(c < 240) {
        for(a = 0; a < 4; a++) /* Copy the temporary variable over */
            t[a] = in[a + c - 4];
        if(c % 32 == 0) {/* Every eight sets, do a complex calculation */
            schedule_core(t,i);
            i++; }
        if(c % 32 == 16) {
            for(a = 0; a < 4; a++)
                t[a] = sbox(t[a]); }
        for(a = 0; a < 4; a++) {
            in[c] = in[c - 32] ^ t[a];
            c++;
        }
```

AES: Step 0 (Add round key)

- XOR 128 bits of message with first 128 bits of expanded key

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a3	d3	9a	c9	18	55	c5	71	b1	eb	e3	89	4d	5c	4f	47

AES: Step 1 (Substitute Bytes)

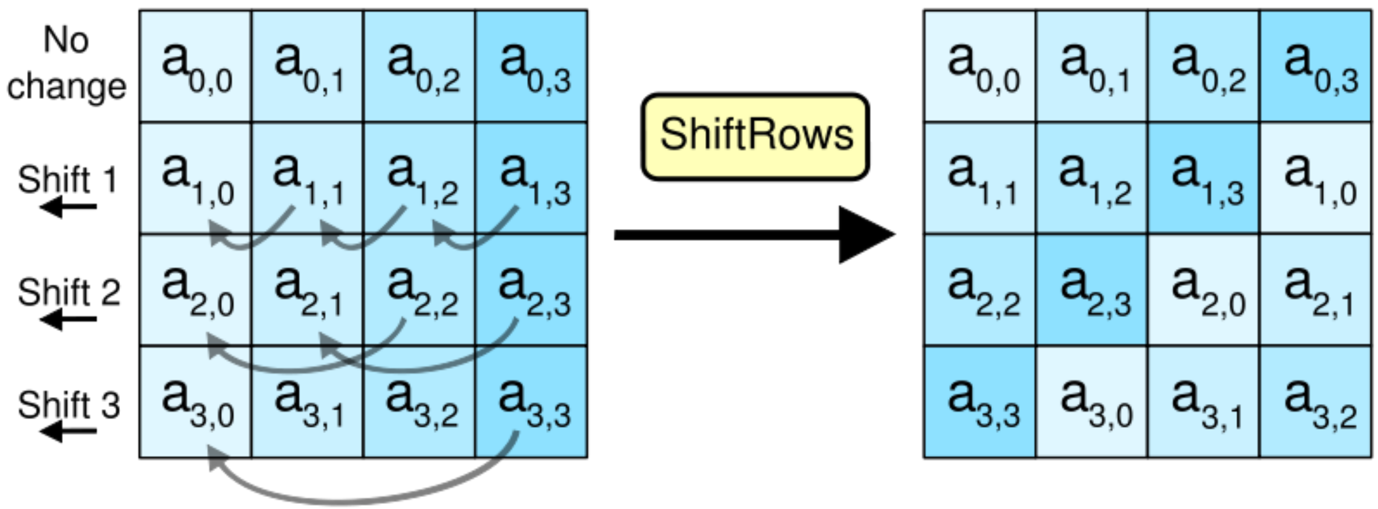
AES S-box

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

For example, 0x9a substitutes to 0xb8

AES: Step 2 (Shift rows)

- First row unchanged
- Second row shifts left by 1
- Third row shifts left by 2
- Fourth row shifts left by 3



AES: Step 3 (Mix Columns)

- Each 4-element column is mixed

```
void mix_columns(unsigned char *r) { /* input is array of 4 bytes = 1 column */
    unsigned char a[4];
    unsigned char b[4];
    for (unsigned char c = 0; c < 4; c++) {
        a[c] = r[c]; /* copy of input */
        b[c] = r[c] << 1;
        unsigned char h = r[c] >> 7; /* logical right shift, h is 0x01 or 0x00 */
        b[c] = b[c] ^ (h * 0x1B); /* Rijndael's Galois field */
    }
    r[0] = b[0] ^ a[3] ^ a[2] ^ b[1] ^ a[1]; /* 2 * a0 + a3 + a2 + 3 * a1 */
    r[1] = b[1] ^ a[0] ^ a[3] ^ b[2] ^ a[2]; /* 2 * a1 + a0 + a3 + 3 * a2 */
    r[2] = b[2] ^ a[1] ^ a[0] ^ b[3] ^ a[3]; /* 2 * a2 + a1 + a0 + 3 * a3 */
    r[3] = b[3] ^ a[2] ^ a[1] ^ b[0] ^ a[0]; /* 2 * a3 + a2 + a1 + 3 * a0 */
}
```

AES: Step 4 (Add round key)

- XOR 128 bits of message with next 128 bits of expanded key

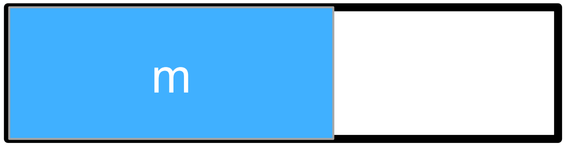
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
d7	b8	f7	62	49	3f	05	2d	97	f7	ce	8a	ea	f4	c4	38

AES: Repeat Rounds

- Repeat Steps 1-4 14 total times
- Except skip Mix columns in last round

Padding

What if the message length isn't *exactly* a multiple of block length? End up with final block that isn't full:

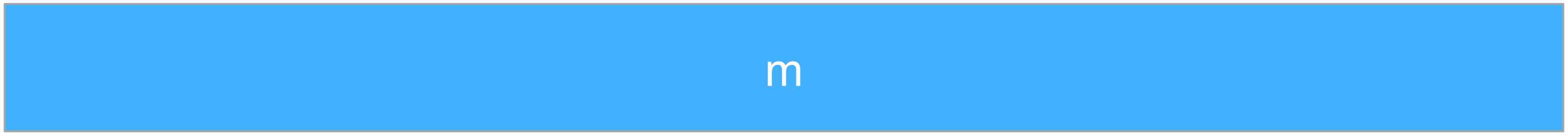


Non-solution: pad out final block with 0's (not reversible)

Solution: Let B be the number of bytes that need to be added to final plaintext block to reach block length. Pad with B copies of the byte representing B. Called *PKCS #5 or #7 padding.*

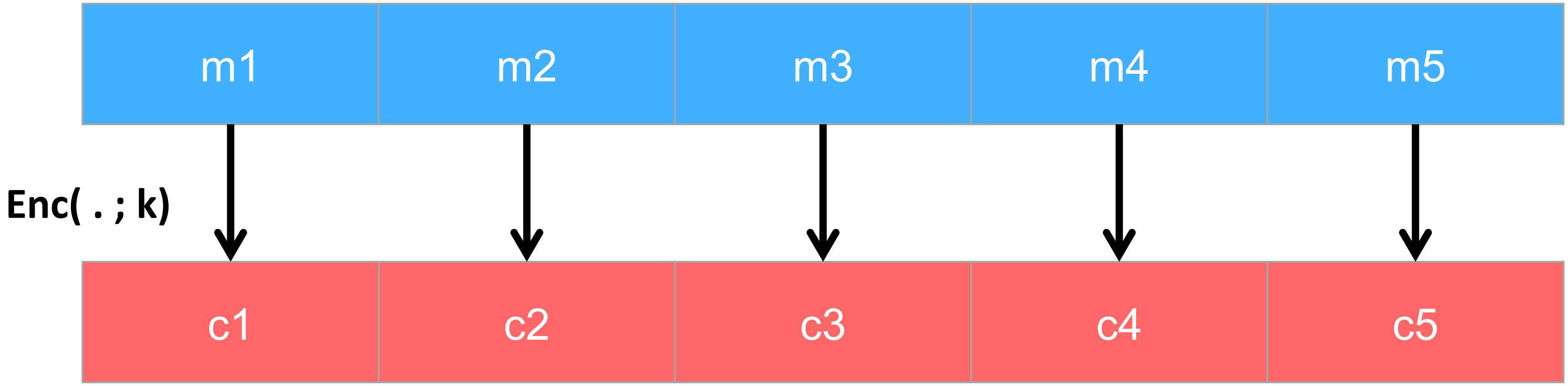
The obvious idea...

- Divide long message into short chunks, each the size of a block
- Encrypt each block with the block cipher



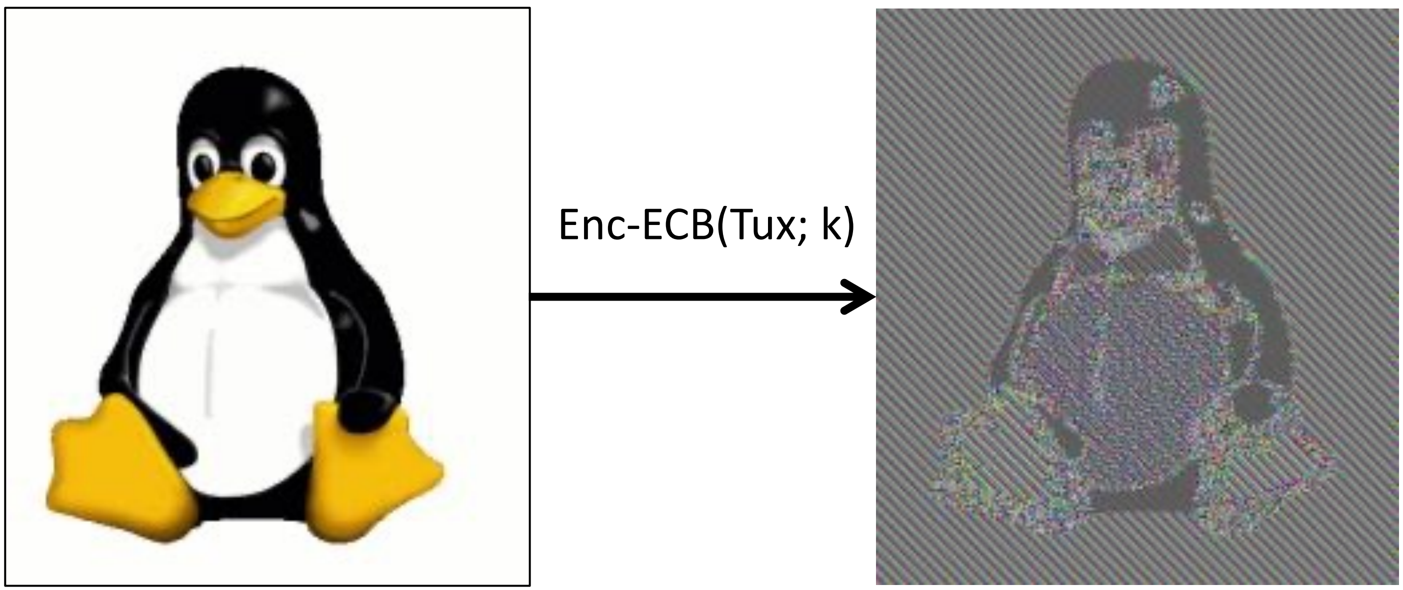
The obvious idea...

- Divide long message into short chunks, each the size of a block
- Encrypt each block with the block cipher



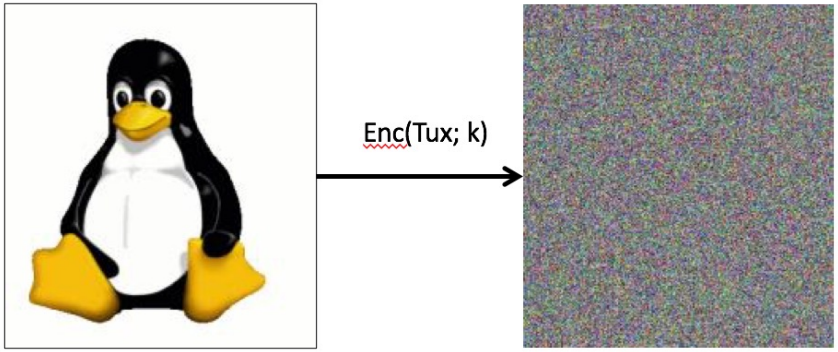
- Called *electronic code book* (ECB) mode

...is a bad idea



Better modes

- Cipher Block Chaining (CBC) mode
  - idea: XOR previous ciphertext block into current plaintext block
- Counter (CTR) mode
  - idea: derive one-time pad from increasing counter
- With both:
  - every ciphertext block depends in some way upon previous plaintext or ciphertext blocks
  - so even if plaintext blocks repeat, ciphertext blocks don't
  - so *intra-message* repetition doesn't disclose information

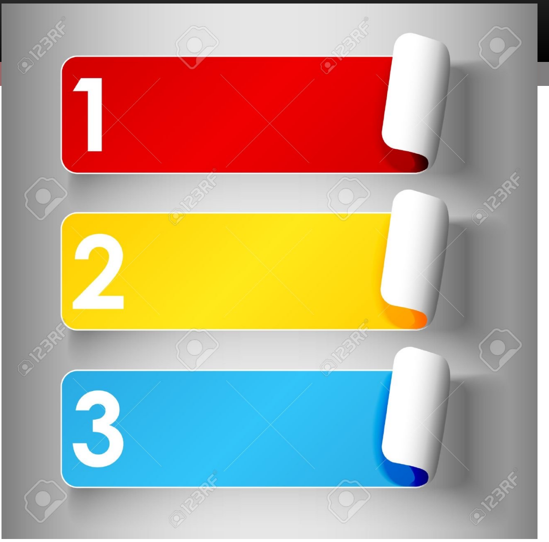


One more problem…

- Problem: block ciphers are *deterministic*: inter-message repetition is visible to attacker
- Both CBC and CTR modes require an additional parameter: a *nonce*
  - Enc(m; nonce; k)
  - Dec(c; nonce; k)
  - CBC calls the nonce an *initialization vector* (IV)
- Different nonces make each encryption different than others
  - Hence inter-message repetition doesn't disclose information

Nonces

A nonce is a number used once



Must be

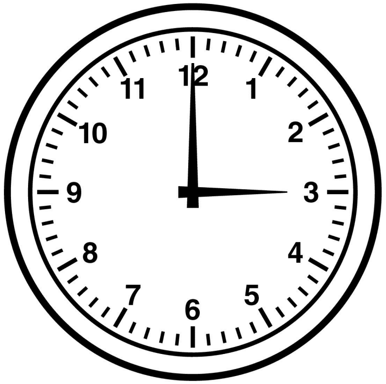
- **unique:** never used before in lifetime of system

and/or (depending on intended usage)

- **unpredictable:** attacker can't guess next nonce given all previous nonces in lifetime of system

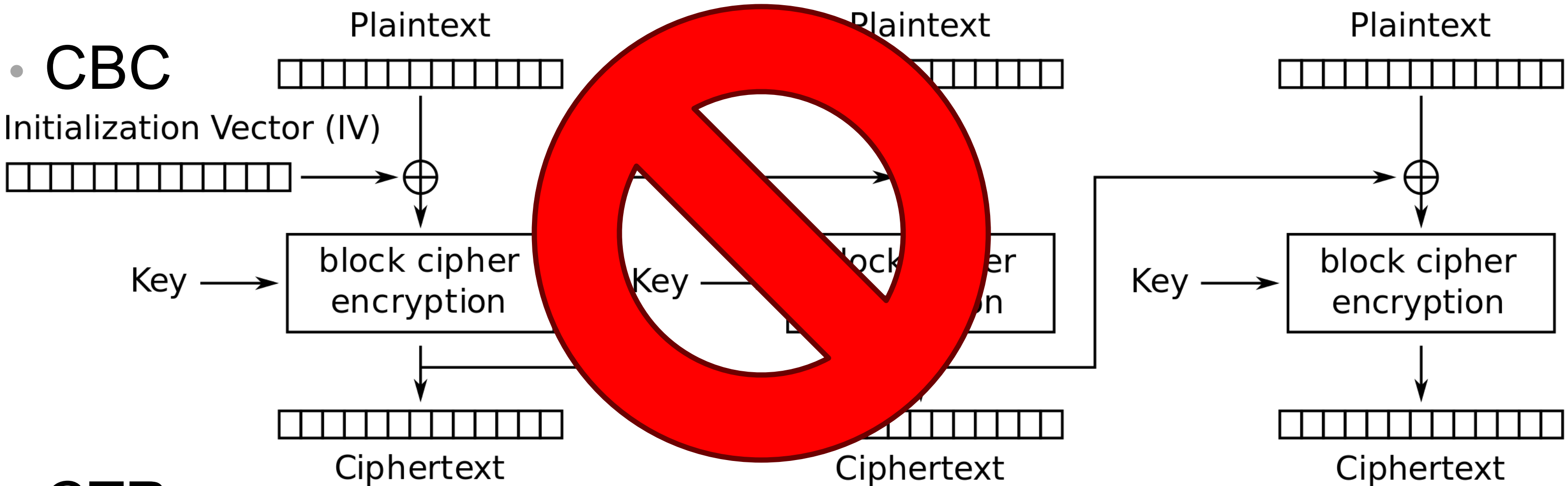
Nonce sources

- **counter**
  - requires state
  - easy to implement
  - can overflow
  - highly predictable
- **clock:** just a counter
- **random number generator**
  - might not be unique, unless drawn from large space
  - might or might not be unpredictable
  - generating randomness:
    - standard library generators often are not cryptographically strong, i.e., unpredictable by attackers
    - cryptographically strong randomness is a black art



How these modes work

- CBC



- CTR