# Information Security (WBCS004-05)

**Fatih Turkmen**

Some slides are borrowed from Dr. Frank B. Brokken

## Today

- Topics (1st lecture) on Modern Cryptography:
  - More on Transposition ciphers
  - Perfect Secrecy
  - One-time Pad (Vernam cipher)
  - Symmetric Crypto
    - Stream ciphers (RC4)
    - Block ciphers (Feistel, DES, 3DES)
  - Asymmetric Crypto
    - Knapsack

# Double Transposition

A permutation matrix P is a matrix that has only one "1" in each row or column.

- Is a form of **diffusion**
  - Smears the text in matrix M containing the plaintext row/column-wise over another matrix C containing the ciphertext. How to implement:
    - use *permutation matrices* P. If P' denotes P's transposition then **P'P = I**.
    - fill a matrix **M** with the plaintext;
    - p**R**emultiply by permutation matrix $\mathbf{P_1}$ to permute **R**ows: $\mathbf{P_1 M}$;
    - p**O**stmultiply $\mathbf{P_1 M}$ by permutation matrix $\mathbf{P_2}$ to permute c**O**lumns: $\mathbf{C} = \mathbf{P_1 M P_2}$
  - Decrypting is easy:
    - $\mathbf{P_1' C P_2' = (P_1' P_1) M (P_2 P_2') = M}$

# Double Transposition

- Transposition Example:

permutes rows. E.g., row 1 to row 2

$$\begin{matrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{matrix} * \begin{matrix} u & s & i & n & g \\ a & t & r & a & n \\ s & p & o & s & i \\ t & i & o & n & c \\ i & p & h & e & r \end{matrix} = \begin{matrix} a & t & r & a & n \\ u & s & i & n & g \\ t & i & o & n & c \\ i & p & h & e & r \\ s & p & o & s & i \end{matrix}$$

$$\begin{matrix} a & t & r & a & n \\ u & s & i & n & g \\ t & i & o & n & c \\ i & p & h & e & r \\ s & p & o & s & i \end{matrix} * \begin{matrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{matrix} = \begin{matrix} n & a & a & t & r \\ g & u & n & s & i \\ c & t & n & i & o \\ r & i & e & p & h \\ i & s & s & p & o \end{matrix}$$

permutes cols. E.g., col 4 to col 3

- Key: 2 1 4 5 3,   5 1 4 2 3 $\rightarrow$ badec, eadbc

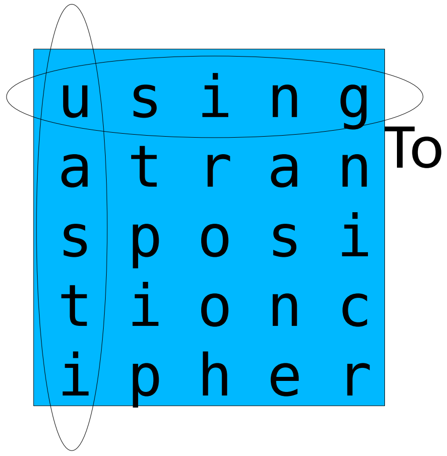
- A permutation matrix P is a matrix that has only one "1" in each row and column.
- When premultiplying a matrix (i.e., M) by P each row *i* will be moved to row *j*, if $P_{ij} = 1$ (i,j row and column indices for P)
- When postmultiplying a matrix (intermediate M) by P each column *j* will be moved to column *i*, assuming that $P_{ij} = 1$
- Multiplying P by its transpose results in the identity matrix:

$$P'P = PP' = I$$

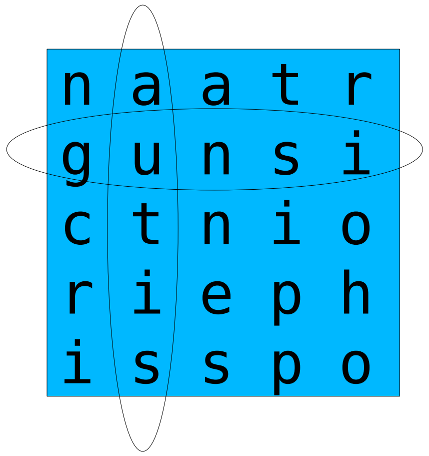
If I use words to represent these keys, what is the relation between the these two?

# Double Transposition

Fancy of online ciphers? https://www.boxentriq.com/code-breaking/double-transposition-cipher

- Transposition Example:
  - From: 

|   |   |   |   |   |
|---|---|---|---|---|
| u | s | i | n | g |
| a | t | r | a | n |
| s | p | o | s | i |
| t | i | o | n | c |
| i | p | h | e | r |

To: 

|   |   |   |   |   |
|---|---|---|---|---|
| n | a | a | t | r |
| g | u | n | s | i |
| c | t | n | i | o |
| r | i | e | p | h |
| i | s | s | p | o |

- *Breaking is tedious but doable*:
  each row cq. each column shows the characters of a row cq. column of the original matrix. Once you can guess (find something meaningful), then the rest quickly follows; e.g., multiple anagramming over multiple messages.

## Perfect Secrecy (i.e., Provably Secure)

Assume that the key is chosen at random, and used only once

When something is provably secure, it means “some statement” regarding it's security can be (has been) proven.

*“Given the ciphertext, any ‘plaintext’ of the same length can be generated by a suitable choice of ‘key’ and all possible plaintexts are equally likely. So the ciphertext provides no meaningful information at all about the plaintext.”*

# Perfect Secrecy more formally

- **M**: message space
- **K**: key space
- **C**: the set of all possible ciphertexts

Perfect Secrecy is defined with probability distributions over M, K and C

- For any key k ∈ K, **Pr[K=k]** denotes the probability that Gen's output is equal to k.
- **Pr[M=m]**: the message takes on the value m ∈ M
- **Pr[C=c]**: $E_k(m)$ results with c ∈ C,

c ⟵ $E_k(m)$: probabilistic, i.e. c is selected probabilistically from C

c **:=** $E_k(m)$: deterministic

An encryption system: <G,E,D>

- G: a probabilistic key generation algorithm
- $E_k(m)$: Encryption algorithm (key k and message m as input, and ciphertext c as output)
- $D_k(c)$: Decryption algorithm (key k and ciphertext c as input, and message m as output)

Conditional Probability in Encryption (Bayes' Theorem)

Pr[M=a | C=B]: What is the probability that the message a was encrypted given that we observe ciphertext B?

What is this?

$$Pr[M=a \vee C=B] = Pr[C=B|M=a¿.Pr[M=a]\frac{¿}{Pr[C=B]}$$

## Perfect Secrecy more formally (cont.)

Definition 1: An encryption scheme is perfectly secret with a message space M, if for every probability distribution for M, every message $m \in M$ and every ciphertext $c \in C$ (for which $Pr[C=c] > 0$)

$$Pr[M=m \mid C=c] = Pr[M=m]$$

**Meaning**: The **a posteriori probability** that some message $m \in M$ was sent, conditioned on the ciphertext that was observed, should be no different from **a priori probability** that m would be sent.

**Plain English**: An eavesdropper can observe the ciphertext. Observing the ciphertext should have no effect on the adversary's knowledge regarding the actual message that was sent.

# Perfect Secrecy more formally (cont.)

Two more but equivalent definitions:

- Definition 2 (**Ciphertext Distribution**): One that requires the distribution of the ciphertext does not depend on the plain text. That is, for every m,m' ∈ M and every c ∈ C:

$$\Pr[E_k(m) = c] = \Pr[E_k(m') = c]$$

- Definition 3 (**Perfect (adversarial) indistinguishability**): One, perhaps more interesting, that considers an experiment where, given a ciphertext, the adversary (eavesdropper, denoted as ) tries to guess which of the two possible messages was encrypted. Denoted as
  - outputs a pair of messages $m_0, m_1 \in M$
  - A key k is generated using Gen and a uniform bit $b \in \{0,1\}$ is chosen. $c \longleftarrow E_k(m_b)$ is computed and given to .
  - outputs a bit b'.
  - The result of the experiment is 1 if b' = b, otherwise 0. That is

$$\boxed{\text{We want } \Pr[=1] = 1/2}$$

## One-time Pad (OTP, a.k.a. Vernam cipher)

- Provably secure
- **Textbook Definition**: (Plain text and Key) Letters are mapped to bits (e.g., 'a' = 01000001) and bitwise XORed
- Assume the elements of m (message) and k (key) are integers modulo 26
  *To encrypt a message m sequence with the key k sequence*

$$E_k(m[i]) = c[i] = (m[i] + k[i]) \% 26$$

*To decrypt*

$$D_k(c[i]) = m[i] = (26 + c[i] - k[i]) \% 26$$

In binary form: Add the plaintext and key bits **modulo 2**, i.e., XOR

Just to stay always positive

- The key *k* is a random sequence with the size |m|
- *Any* plaintext can be reconstructed from a given *c*

# One-time Pad (cont.)

- In **OTP**: *xor* (⊕) for enc/dec
- for *xor* we know;
  - $a \oplus 0 = a$ (identity),
  - $a \oplus a = 0$ (inverse),
  - $a \oplus (b \oplus c) = (a \oplus b) \oplus c = (a \oplus c) \oplus b$ (associativity)

*Xor* truth table:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | rhsrhs | | | |
| | | ⊕ | ⊕ | 1 | 1 | 0 | 0 | |
| lhs | lhs | 1 | 1 | v | 10 | v | 21 | |
| | | 0 | 0 | v | 31 | v | 40 | |

❗ What are the values of v1,v2,v3,v4?

# One-time Pad (cont.)

- In OTP; how does Enc/Dec really work?

```
(Enc) m[i] ⊕ k[i] = c[i]
(Dec) c[i] ⊕ k[i] = (m[i] ⊕  k[i]) ⊕ k[i]
                   =  m[i] ⊕ (k[i] ⊕ k[i])
                   =  m[i] ⊕    0    = m[i]
```

- Recall: Enc/dec in **simple substitution ciphers** using addition/subtraction (very similar!):

```
     (m[i] + shift) % 26 = c[i]
(26 + c[i] - shift) % 26 = m[i]
```

# One-time Pad (cont.)

Everything is modulo the chosen alphabet length!

- *One-time pad (dropping the letter index i):*

$m_1 + k_1 = c$ so: $m_1 = c - k_1$

encryption

$m_2 + k_2 = c$ so: $m_2 = c - k_2$

(encrypt)
(decrypt)

actually used

other possibility/ies for achieving c?

- Same cipher text, only the keys differ. *c* does not reveal any info whether it is derived from $m_1$ or $m_2$.

$m_1 + k_1 - m_2 = c - m_2 = k_2$

- How can we obtain such key ($k_2$)?:

❗

Note: *any* key is possible; there is *no* reason why *key-1* should be the `correct key', rather than *key-2*

# One-time Pad (cont.)

| a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |

- *Example*
  - Real text[1] and used key:

    Plain text, $m_1$: kill the president

    Key: qwertyuiiopasfga

❗ What are the first two letters of the ciphertext?

*aepcmfyxzshivjtt*

$(m_1 + k_1 = c)$

- *Let's see how it is perfectly secure (Definition 1)*
  - Choose an alternate plaintext, $m_2$ : hello world crypto
  - First: *decrypt* cipher text with the alternate plain text

    Decrypt: AEPCMFYXZSHIVJTT with `key' hello world crypto

    plaintext/ciphertext: taeryjkgopfrxuaf ($k_2$: the alternative key)

$(k_2 = c - m_2)$

[1]If you run this example remember to omit the blanks

# One-time Pad (cont.)

- Then: *apply* the alternate key to the ciphertext:

Decrypt: AEPCMFYXZSHIVJTT with taeryjkgopfrxuaf

Results in: helloworldcrypto

What did we do?

$m_1$(kill the president) + $k_1$(qwertyuiiopasfga) = $m_2$(hello world crypto) + $k_2$(taeryjkgopfrxuaf)

Pr[M="kill the president" | C="*aepcmfyxzshivjtt*"] = Pr[M="kill the president"]

# One-time Pad (cont.)

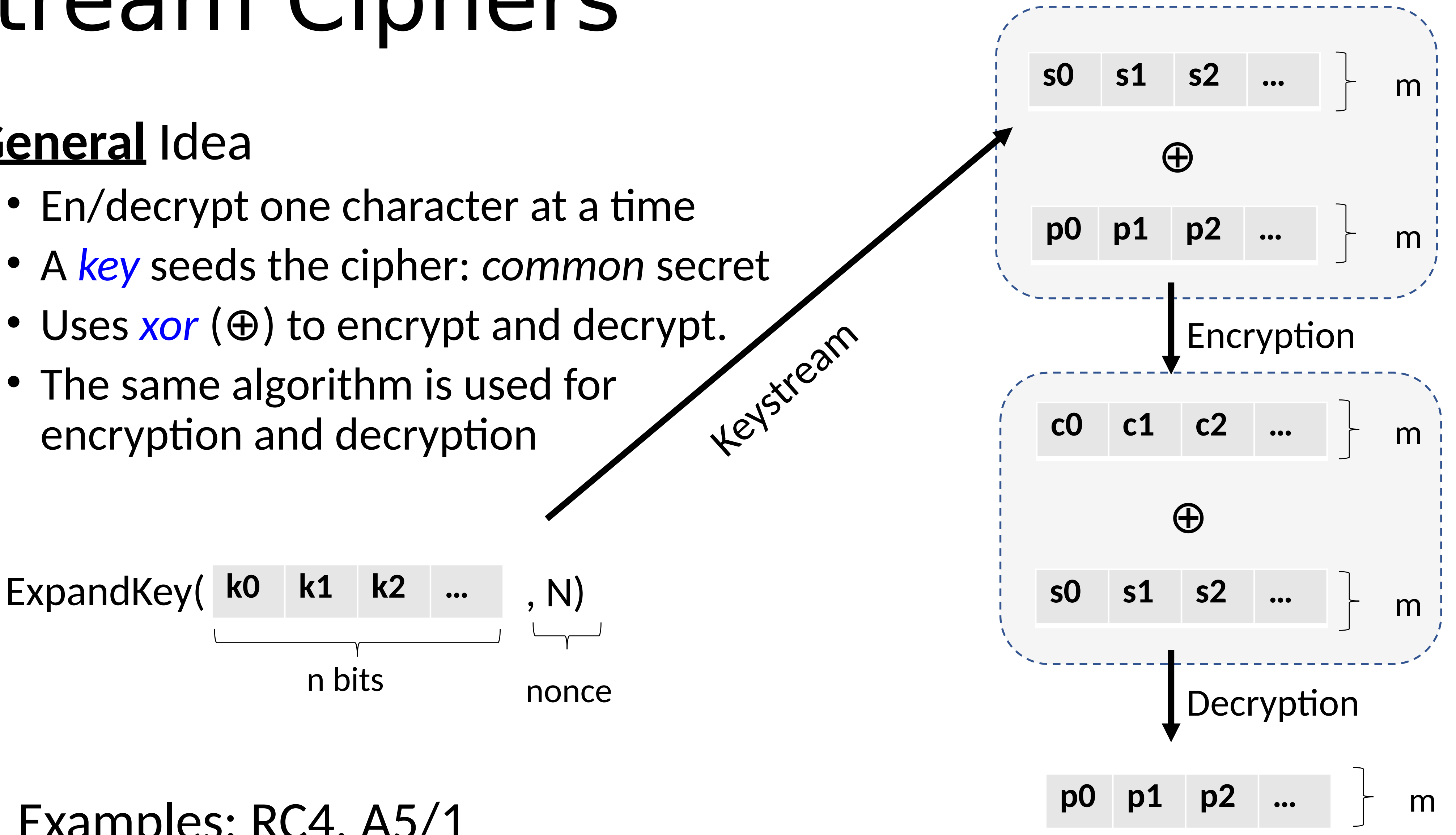
- *One-time pad - disadvantages*
  - Key size, distribution and usability (*in depth* vulnerability)
  - If padding is used then the “pad” information needs be securely communicated!
  - Can be used once! ▭ Key needs to be changed after every encryption

  $m1 + k = c1$; $m2 + k = c2$ $c1 - c2 = m1 - m2$ *(the key vanishes)*

  (note: m1, m2: plain *texts*, not characters, *k*: the used key)

## Stream Ciphers

- **General** Idea
  - En/decrypt one character at a time
  - A *key* seeds the cipher: *common* secret
  - Uses *xor* (⊕) to encrypt and decrypt.
  - The same algorithm is used for encryption and decryption



- Examples: RC4, A5/1

## Stream Ciphers

Stamp’s book mentions:

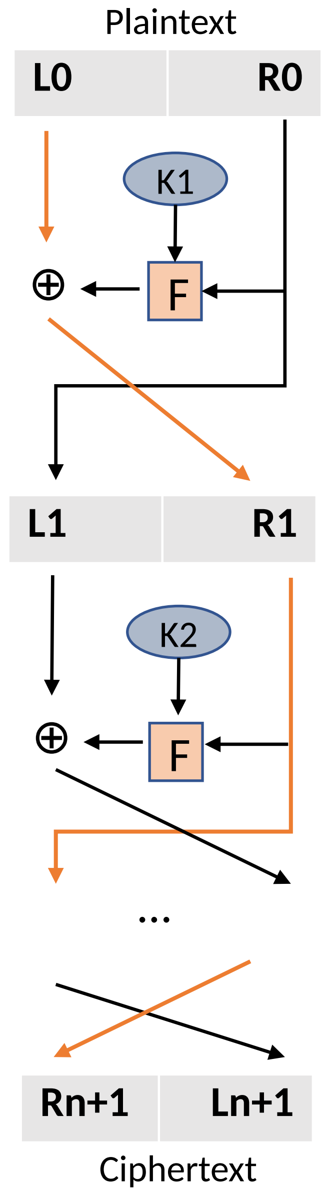
- RC4:
  - Used in SSL communication. If used correctly: fine
  - Byte by byte encryption
  - Tailored to software
- A5/1:
  - Tailored to hardware
  - Used in GSM encryption. Cracked. But how bad is that?
  - cf: *http://www.schneier.com/blog/archives/2008/02/cryptanalysis_o_1.html*

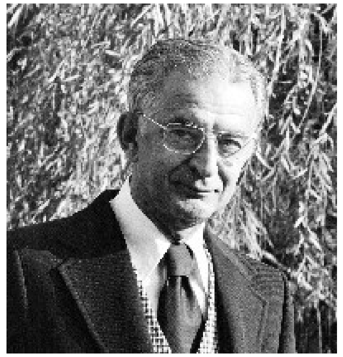
There are many others such as Trivium (EU Project), E0 (bluetooth)… See the Wikipedia for a list.

## Block Ciphers

- Block ciphers:
  - Key Motivations: **Efficiency** and **Security**
  - Widely used
  - Encryption not *byte-wise* but *block-wise* (e.g. 8 byte blocks)
  - Plaintext is mangled over several `rounds'
  - To decrypt: the rounds are played back.
  - It is *difficult* to develop *secure **and** efficient* algorithms.
  - Block ciphers are often designed as Feistel ciphers

# Feistel





- Feistel ciphers:
  - Horst Feistel (1915-1990)
  - Achieve **invertibility** from **non-invertible** components (i.e., round function)
  - **Encryption** procedure:
    - Split blocks in left/right halves

$$M = (L, R)$$

    - The next round's *left* half is the previous *right* half:

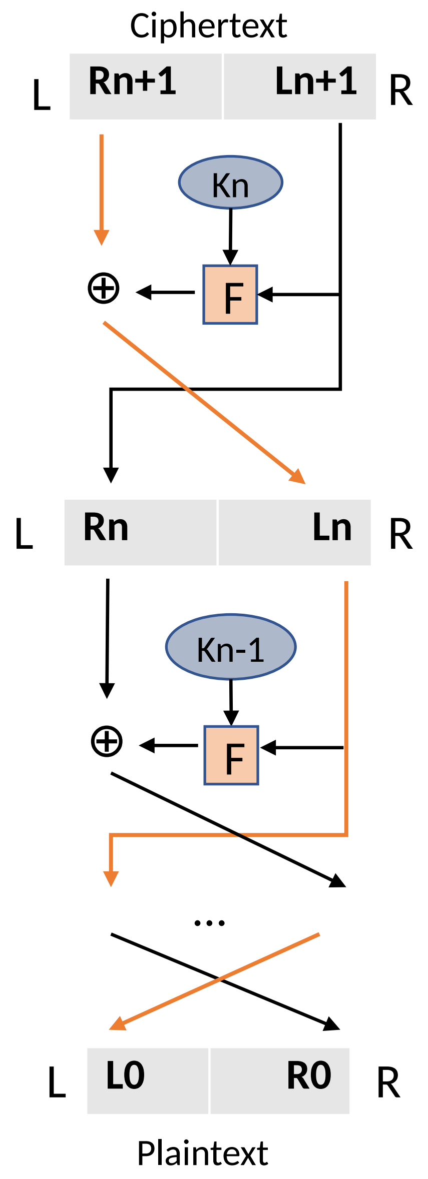
$$L_i = R_{i-1}$$

    - The next round's *right* half: previous *left* *xor*-ed with the result of a round (*key schedule*) function **F**:

$$\boxed{R_i = L_{i-1} \oplus F(R_{i-1}, \mathbf{K_i})}$$

Note the final swap!!

# Feistel (cont.)



- ***Decrypting*** Feistel ciphers:
  - <u>**Invert**</u> the process until M is reached:
    - Split the ciphertext in halves: $C = (L_i, R_i)$
    - Compute ***previous* right** from *current* left.

    That is: since $L_i = R_{i-1}$ (encryption) we get $R_{i-1} = L_i$

    - Compute **previous left**: from encryption

    That is: since we have $R_i = L_{i-1} \oplus F(R_{i-1}, K_{i-1})$ [Encryption]

    we get: $L_{i-1} = R_i \oplus F(R_{i-1}, K_i)$

KAERB A DEEN I || I NEED A BREAK!

# DES

- *Data Encryption Standard*
- *Feistel cipher* (with 16 rounds)
- Key/block length: **64 bits** (key used to be 128  in Lucifer)
- Actual # of bits used for the key: 56 (8 bits are discarded)
- Each round uses 48 bit subset of the 56 bit key
- Think about brute force attacks….
  - Lucifer had exhaustive key search space of $2^{127}$
  - DES has exhaustive key search space of  $2^{55}$

A thought experiment

Assume that:
- The effective key length of DES is 48 bits
- We can test $10^8$ keys per second
- $2^k$ is approximately $10^{k/3}$
How many (roughly) **months** does it take to find a DES key?

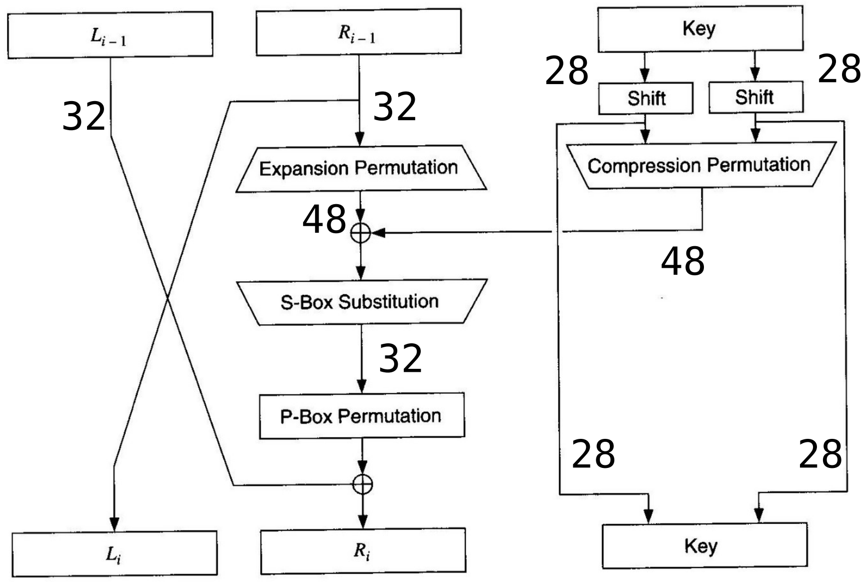
# DES

- One DES round:
  - Expansion:

    32 bits to 48.
  - S-boxes (8 of them):

    map 6 bits down to 4

    *provides non-linearity*
  - Key schedule:
    - initially permute the **master key**, and split it to two 28 bits
    - shifts + compression
    - next key: concatenates shifted (rotated) halves
    - key schedule is fixed and public, only master key is secret



**(Feistel) Round function in DES**

$F(R_{i-1}, K_i) = \text{P-Box}(\text{S-Boxes}(\text{Expand}(R_{i-1}) \oplus K_i))$

**The main vulnerability is the key size!**

# Triples DES (3DES)

- 3DES:
  - DES disadvantage: small key.
  - Solution: use a 112 bit key, split in halves:
    - `2DES' is insufficient: Using a table of $2^{56}$ keys and a known *plaintext* 2DES can be broken.
    - 3DES is defined as:

```
C = E( D( E(P,K₁), K₂), K₁)
```

$\longrightarrow$ Why EDE instead EEE?

*Backward compatibility!*

- ***Nice:*** for $K_1$ == $K_2$ this reduces to DES:

```
C = E( D( E(P,K), K), K)
```

- so:

```
C = E(       P        , K)
```

## Block Cipher Modes

- How should multiple blocks be encrypted with a block cipher?
- Highlights:
  - Padding
  - Avoid: Electronic Code Book (ECB) mode
  - Acceptable modes
    - CBC (cipher block chaining mode)
    - CTR (counter mode)
  - Initialization vectors
  - Information Leakage / Collisions
- See also: http://csrc.nist.gov/groups/ST/toolkit/BCM/modes_development.html

## Block Cipher Modes

- Padding
  - Padding is almost (CBC) always (ECB) required, always *at least* 1 byte
  - Well-known methods (block size *b*):
    - *Marker:* add byte *128* and a remaining number of required *0*-bytes (*0..b-1*)
    - Determine how many padding bytes are needed (*n*) and *add* that many *n-value* bytes.

## Block Cipher Modes

- Avoid: *Electronic Code Book* (ECB)

```
CodeBlock = E(PlainText, Key)  (for each block)
```

  - Same plain text results in identical code blocks
    - Think about padding, headers, footers
    - Highly susceptible to
      - Known plaintext attack
  - Plain text often has predictable characteristics
  - What is the key?
    - With ECB the key is always the same, opportunities for known plaintext attacks

# Block Cipher Modes

- OK mode: Cipher Block Chaining (CBC)
  - The plain text block is first xor-ed with the previous cipher text
  - Advantage: Identical plain text blocks no longer result in identical cipher blocks
  - Method:

```
CodeBlock = E(PlainText xor Previous Code Block, Key)
```

  - **Still to solve**: what is the '*previous encrypted block*' for the very *first* plain text block?
    - Use Initialization Vectors (IVs)

## Block Cipher Modes

- OK mode: Counter Mode (CTR)
  - A block cipher mode used as a *stream cipher*.
  - The algorithm generates a *stream of keys*
  - Method:

concatenation

```
Key2Use = E(nonce || I, Key)          (for i = 1, ... k)

CipherText = PlainText xor Key2Use
```

# Block Cipher Modes

- OK mode: Counter Mode (CTR)
  - A block cipher mode used as a *stream cipher*.
  - The algorithm generates a *stream of keys*
  - Method:

concatenation

```
Key2Use = E(nonce || I, Key)          (for i = 1, ... k)

CipherText = PlainText xor Key2Use
```

  - 128 bit blocks, e.g.

128 bits

- *Nonce*: e.g., 48 bit nonce (i.e., IV) plus 16 bit encoded "i": must be *unique* or *data will leak* (cf. ECB)
- 64 bit counter

## Block Cipher Modes

- Initialization vectors (IVs)
  - Are used to initialize block ciphers:
    - 'Code block' for the 1st plain text block (*CBC*)
    - Should not be fixed (or data will leak with different messages)
    - *Counter IV*s: small differences and (almost) identical plain text blocks may cause data to leak
    - *Random IV*: fine, but IV is sent as 1st block, enlarging the message by 1 block
    - *Nonce*: used to *compute* the IV/Key (*in case of CTR*), is much smaller

## Block Cipher Modes

- Initialization vectors (IVs)
  - No need to keep secret.
  - Consider:

```
CodeBlock = E(Key, PlainText xor Previous Code Block)
```

    - Plaintext is (i.e., remains) unknown;
    - *Plaintext xor IV* enters the key schedule, where it is modified:
      - the plaintext cannot be reconstructed from the CodeBlock alone.
  - Analogous considerations apply to CTR.

# Block Cipher Modes

- Information Leakage
  - ECB: always leaks (of course: same keys)
  - With CBC, when C blocks are equal, then:

```
              Ci == Cj
E(K, Pi ⊕ Ci-1) == E(K, Pj ⊕ Cj-1)
      Pi ⊕ Ci-1 == Pj ⊕ Cj-1
        Pi ⊕ Pj == Ci-1 ⊕ Cj-1
```

    - *previous* C-blocks reveal info about P blocks (cf. one-time pad)
    - There is also cut-and-paste attack (see the book)
  - CTR does not have this property (which is nice)

# Enter: Asymmetric Cryptography

Knapsack Cryptosystem

- Does not use a shared key
- First example of *public key cryptography*
- Proposed by Merkle and Hellman
- Based on a paper by Diffie and Hellman
- Uses a *superincreasing knapsack*

# Knapsack Cryptosystem

i= {**0**…n}

- Knapsack:
  - The **general** knapsack problem:
    - Given a set of n weights, $w_0,..w_{n-1}$, find a series of (integral) values $a_i$ ($\in \{0,1\}$) such that a given sum **S** is achieved:

    S =

  - Special case: *superincreasing knapsack* (SK):
    - Order the weights from least to greatest
    - **But**: Each next $w_i$ value exceeds the sum of the previous $w_j$ values: $\forall w_i$: $w_i > \sum w_j$ $(0 \leq j < i)$
    - The problem is now easily solved: **start with the largest weight, repeatedly subtract values until 0.**

# Knapsack Cryptosystem

- Example SK:

```
  2, 3, 7, 14, 30, 57, 120, 251
```

$(\sum w_j$: 2 5 12 26 56 113 233)

- For S = 171, the $a_i$ are:

```
  0, 0, 1,  1,  1,  0,   1,   0
```

How do we solve this?

171-120=51 → New target sum

51-30=21 → New target sum

21-14=7

7-7=0

# Asymmetric Crypto

- In public key crypto, you need:
  - a *private key*, kept secret, allowing us to *decrypt* information;
  - a *public key*, made public, allowing others to send us *encrypted* information
- In practice, the *private key* cannot be obtained from the public key.

# Constructing a Knapsack Cryptosystem

- Given the SK:

```
   2, 3, 7, 14, 30, 57, 120, 251   (k values)
(∑wj: 2   5   12   26   56   113   233)
```

- Generating other parts of the private key:
  - Choose *n* (called modulus)
    - $n > \sum w_i$ $(0 \leq i < k)$, e.g. $n = 491$ $(> 251 + 233)$
  - Choose *m* such that *m rp n* (i.e., m and n are relatively prime)
    - m: a multiplication constant. E.g., m = 41
  - Private key: the SK and the values m and n.

# Constructing a Knapsack Cryptosystem

- Given the SK:

```
  2, 3, 7, 14, 30, 57, 120, 251
```
($\sum w_j$: 2 5 12 26 56 113 233)

- The *public key* is the *general knapsack*:
  - Computed from the SK:

    Public key: $\mathbf{k_i = m * w_i \% n}$, producing:

    Weights from SK

```
82,123,287,83,248,373,10,471
```

  - Private key: *m*, *n* and the superincreasing knapsack
    - n > $\sum w_i$, e.g. 491
    - m rp n: e.g., 41

# Constructing a Knapsack Cryptosystem

- Knapsack: *encrypting* a message
  - Assume that the message is ”p” in binary **00001110**$_b$
  - *Use the bit-pattern as weight-vector* and *compute c =* **a’w** using the published *public* key

```
    82,123,287,83,248,373,10,471
idx: 0, ...                    7
  a: 0,  0,  0, 0,  1,  1, 1,  0
```

c = 248 + 373 + 10 = 631

What do we use here?
**Hint**: This asymmetric/public key crypto

# Constructing a Knapsack Cryptosystem

- Knapsack: *decrypt* a message (the receiver has *m* and *n*):
  - Compute (once) $\mathbf{m^{-1} \% n} = 41^{-1} \% 491 = \mathbf{12}$ (modulo inverse)

  - Intuition: the public key has values: $\mathbf{k_i = m * w_i} \% n$,
    so: $m^{-1} * k_i = m^{-1} * m * w_i \% n = w_i \% n = w_i$

# Constructing a Knapsack Cryptosystem

- Knapsack: *decrypt* a message:

$$m^{-1} * k_i = \boxed{m^{-1} * m * w_i \% n = w_i \% n = w_i}$$

- Encrypted values are handled likewise :

$$c = \sum k_i = \sum m * w_i = m \sum w_i$$

so: $$\mathbf{m^{-1} * c} = m^{-1} \sum m * w_i = m^{-1} * m \sum w_i = \sum \mathbf{w_i} (\% n)$$

- Example:
  - c = 631, 631 * 12 (% 491) = 207 $(= \sum \mathbf{w_i} (\% n))$

# Constructing a Knapsack Cryptosystem

- Knapsack: *decrypting* a message:
  - We just computed:
    - $C = 631, \quad \overset{m^{-1}}{631 * 12} \mathbin{\%} \overset{n}{491} = 207 = \sum a_i \mathbin{\%} n$
  - Next solve S=207 in the superincreasing knapsack:
    - 207 = 30 + 57 + 120

*idx:0* `2, 3, 7, 14, 30, 57, 120, 251` *idx:7*

  - Interpret the values as 1-weights in the **a** vector and thus in binary: $00001110_b$

## Take Home

- A *huge* knapsack… (just for fun)
  - public key:

```
328D7C79E4FDBE27F1D46591602C408BCF5BFB7DB39A515518A1
819639474E581D95DD469AA423F9B9FED9EB296BD733436A96FC
B9C44FFF07D92879ADD74236FCC71DFF4E151856A6ACC330E8BA
B5740651905C54E59176EA9ACD5B4529101786466EA26FA8A080
31B30E1AE9EDDD7476DE05E8887BE78968862B91C96853BE996A
2D0EEA17CB85D0543344578251358019E144313A21F30008512C
2AF23628723DBC79A2FB5F1E8E374A0DB73EC62E29B75476F924
3F08D48DC39DF0BCB017FD760F8BD80089FFF3964011C40675A2
```

  - private key:

```
Multiplier: 0D19CC01EF6B6082EDF119437B
modulo: E1C0B242BB64C30487F4A19C1039C25AE4F53DD01A4D467A0D0A
F9CFEFE5E4F15F3C0C6151A0654C4281E61CBCB6C94EA851C9
01A3049CAAD6CBBB0B503B249D9A30E86E2351C294EE8DA620EE
039A82CA6C0BBF5831A0A1329DCD63B509C7ED034C913B5A8322
07168E4FD780F9A05E05B189B3C80C9D0E0E5B415D601C4F119A
0E41DE9C50EC4B29A65273E4D99F74E0D8AB2E13D3B6295D3CCC
1C1D9C0592BCC333AC4489C95DB67C0D89F375B5CB5F55B48BE0
385EDBA2475815EF3EC3AD9FCD57454DBC610F906480594178C8
70ED695B21FC6797CF7AA000FF9D0D698127650AE2441B76A218
```

## What did we learn?

- Some more on Transposition Cipher
- Perfect Secrecy and One-time Pad
- Symmetric Cryptography
  - Stream Ciphers
  - Block Ciphers: Feistel Cipher, DES
- Introduction to Asymmetric Cryptography
  - Knapsack Problem

*That's All, Folks,*

*for today.*