Stream and block ciphers

Categorizing ciphers

- by elemental unit
  - stream, a single unit (bit, byte, letter) at a time
  - block, a group of units at a time
- by algorithm
  - symmetric or secret-key (1 key)
  - asymmetric or public-key (2 keys)
- by transformation
  - substitution
  - permutation
Let’s talk about these

- by elemental unit
  - stream ciphers
  - block ciphers
- by algorithm
  - symmetric or secret-key (1 key)
  - asymmetric or public-key (2 keys)
- by transformation
  - substitution
  - permutation

Stream elements

- might be a stream of
  - numbers
  - letters
  - bits
  - bytes
Data streams

Stream ciphers perform progressive element matchup between streams

message(i) matched up with key(i)
for some operation

e.g., E with B, O with S, I with W

Number element encipherment

- Add message element and key element
- Transform as below

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8
```

Example:

```
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>2</td>
<td>6</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>
```

message stream
key stream
resulting ciphertext

© David Morgan 2011-2015
This is modulo-10 arithmetic

- no-carry addition
  - $9+8=17$ throw away the 1, answer is 7
  - $6+7$ is 3
  - $6+4$ is 0
- no-borrow subtraction
  - $8-9$ becomes $18-9$, answer is 9
  - $1-7$ is 4
  - $6-8$ is 8

Number element decipherment

Previous encipherment (earlier slide):

| 6 | 6 | 2 | 8 | 4 | 7 | 1 | 0 | 6 | 9 | 2 | 3 |
| 1 | 7 | 0 | 8 | 4 | 2 | 3 | 7 | 6 | 2 | 7 |

| 7 | 3 | 2 | 6 | 8 | 1 | 3 | 3 | 3 | 5 | 4 | 0 |

Corresponding decipherment:

| 7 | 3 | 2 | 6 | 8 | 1 | 3 | 3 | 3 | 5 | 4 | 0 |
| 1 | 7 | 0 | 8 | 4 | 2 | 3 | 7 | 6 | 2 | 7 |

| 6 | 6 | 2 | 8 | 4 | 7 | 1 | 0 | 6 | 9 | 2 | 3 |

message stream
key stream
resulting ciphertext

ciphertext stream
key stream
resulting recovered message
Modular arithmetic

- clocks use modulo-12
- streams of
  - numbers use modulo-10
  - letters use modulo-26
  - bits use modulo-2
  - bytes use modulo-256

Letter element encipherment

- assign 0 to A, 1 to B, … , 25 to Z
- use modulo-26 arithmetic

Example:

```
COMPUTERIZED
7 7 7 7 7 7 7 7 7 7 7
```

- message stream

```
JVTWBALYPGLK
```

- resulting ciphertext

C becomes J because C (or 2) + 7 = 9 (or J)
U becomes B because U (or 20) + 7 = 27-26 for mod26 = 1 (or B)
Z becomes G because Z (or 25) + 7 = 32-26 for mod26 = 6 (or G)
**Letter encipherment, different key**

- assign 0 to A, 1 to B, …, 25 to Z
- use modulo-26 arithmetic (adding)

Example:

```
<table>
<thead>
<tr>
<th>Message Stream</th>
<th>Key Stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPUTERIZED</td>
<td>4 9 3 18 22 8 11 10 6 15 25 14</td>
</tr>
</tbody>
</table>

C becomes G because C (or 2) + 4 = 6 (or G)
U becomes Q because U (or 20) + 22 = 42-26 for mod26 = 16 (or Q)
Z becomes O because Z (or 25) + 15 = 40-26 for mod26 = 14 (or O)
```

---

**Letter decipherment**

- assign 0 to A, 1 to B, …, 25 to Z
- use modulo-26 arithmetic (subtracting)

Example:

```
<table>
<thead>
<tr>
<th>Ciphertext Stream</th>
<th>Key Stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPBQBOODR</td>
<td>4 9 3 18 22 8 11 10 6 15 25 14</td>
</tr>
</tbody>
</table>

G becomes C because G (or 6) - 4 = 2 (or C)
Q becomes U because Q (or 16 +26 for mod26) = 42 - 22 = 20 (or U)
O becomes Z because O (or 14 +26 for mod26) = 40 - 15 = 25 (or Z)
```
With aids

A Vigenère cipher is a polyalphabetic substitution cipher, which uses a secret key (a keyword) to encrypt and decrypt messages. The key is used to select different Caesar ciphers for different characters in the message. This method is used for hiding the message content from potential attackers.

Example:

**Message stream (nonrandom text)**

```
COMPUTERIZED
```

**If these are random…**

```
GXPHQBPOODR
```

**… so is this.**

Random = follows no pattern, reflects no rhyme nor reason, contains no cause ("nondeterministic")
therefore cannot be reproduced or predicted = unbreakable

© David Morgan 2011-2015
Unbreakable??

But if attacker starts with the ciphertext and comes upon the key (e.g. by brute force, successive attempts) doesn’t he learn the plaintext?

What about this key instead?

different key

same ciphertext
It also “works”

- attacker starts with the ciphertext, comes upon this key and gets this plaintext

plaintext = ciphertext - key

given any plaintext, you can construct a key that produces it from this ciphertext (I just did) (key = ciphertext – plaintext)

no way to verify if a plaintext is the right one

Bit element encipherment

- elements are 0 and 1
- use modulo-2 arithmetic

Example:

```
Message stream           Key stream           Resulting ciphertext
1 0 0 1 1 0 1 1 1 1 0   1 1 1 0 0 1 1 1 1 1 0
0 1 1 0 1 0 1 1 0 0 1 0
```

© David Morgan 2011-2015
**XOR - frequent appearances**

The XOR operator is extremely common as a component in more complex ciphers. By itself, using a constant repeating key, a simple XOR cipher can trivially be broken using frequency analysis. If the content of any message can be guessed or otherwise known, then the key can be revealed. Its primary merit is that it is simple to implement, and that the XOR operation is computationally inexpensive. A simple repeating XOR cipher is therefore sometimes used for hiding information in cases where no particular security is required.

If the key is random and as long as the message (so it never repeats), the XOR cipher is more secure. When the keystream is generated by a virtual random number generator, the result is a stream cipher. With a key that is truly random, the result is a one-time pad, which is unbreakable even in theory.

In any of these ciphers, the XOR operator is vulnerable to a known-plaintext attack, since pl(1) xor ct = key.

http://en.wikipedia.org/wiki/XOR_cipher

---

**Binary XOR operation**

- XORing a bit with 1 *inverts it*
- XORing a bit with 0 *leaves it alone*

**XORing with 1:**
- 1 XOR 1 is 0
- 0 XOR 1 is 1

**XORing with 0:**
- 1 XOR 0 is 1
- 0 XOR 0 is 0
XOR is mod2 addition

XORing with 1:
- 1 XOR 1 is 0
- 0 XOR 1 is 1

XORing with 0:
- 1 XOR 0 is 1
- 0 XOR 0 is 0

adding 1 mod2:
- 1 + 1 = 10 → 0
- 0 + 1 = 1

adding 0 mod2:
- 1 + 0 = 1
- 0 + 0 = 0

XOR twice with same bit leaves input as is

- by inverting twice (if XORing with 1)
  - changes it, changes it back, or
- by inverting never (if XORing with 0)

XORing twice with 1:
- 1 XOR 1 is 0
- 0 XOR 1 is 1

XORing twice with 0:
- 1 XOR 0 is 1
- 0 XOR 0 is 0

or: (A XOR B) XOR B = A
**double XOR = alteration & restoration**

<table>
<thead>
<tr>
<th>input:</th>
<th>11000000 10101000 00000100 00000001</th>
</tr>
</thead>
<tbody>
<tr>
<td>XOR with:</td>
<td>10111110 01001010 10111001 00001101</td>
</tr>
<tr>
<td>result:</td>
<td>01111110 11100010 10111101 00001100</td>
</tr>
<tr>
<td>above result:</td>
<td>01111110 11100010 10111101 00001100</td>
</tr>
<tr>
<td>again with:</td>
<td>10111110 01001010 10111001 00001101</td>
</tr>
<tr>
<td>above input:</td>
<td>11000000 10101000 00000100 00000001</td>
</tr>
</tbody>
</table>

**XOR becomes a symmetric stream cipher**

<table>
<thead>
<tr>
<th>plaintext:</th>
<th>11000000 10101000 00000100 00000001</th>
</tr>
</thead>
<tbody>
<tr>
<td>key:</td>
<td>10111110 01001010 10111001 00001101</td>
</tr>
<tr>
<td>ciphertext:</td>
<td>01111110 11100010 10111101 00001100</td>
</tr>
<tr>
<td>ciphertext:</td>
<td>01111110 11100010 10111101 00001100</td>
</tr>
<tr>
<td>same key:</td>
<td>10111110 01001010 10111001 00001101</td>
</tr>
<tr>
<td>plaintext:</td>
<td>11000000 10101000 00000100 00000001</td>
</tr>
</tbody>
</table>

© David Morgan 2011-2015
XOR operation

- XORing key with plaintext yields ciphertext
  (that’s called encryption)
- XORing key with ciphertext yields plaintext
  (that’s called decryption)

  and also
- XORing plaintext and ciphertext yields key

If key is random, so is ciphertext

| plaintextA:  | 11000000 10101000 00000100 00000001 |
| keyA:       | 10111110 01001010 10111001 00001101 |
| ciphertext: | 01111110 11100010 10111101 00001100 |

| plaintextB: | 01010110 11101010 00100001 01101001 |
| keyB:       | 00101000 00001000 10011100 01100101 |
| ciphertext: | 01111110 11100010 10111101 00001100 |

The (single) ciphertext shown is representative of both plaintexts, given the corresponding key. A key can be constructed to convert any plaintext to this same ciphertext. Attacker must ask which key was actually used, to arrive at the actual plaintext. If key is produced randomly, he has no basis to choose any particular key therefore none to choose the actual one.
For unbreakability

- keystream must be as long as the plaintext
- keystream elements must be random
- same keystream must never be re-used
  - possession of 2 ciphertexts from same keystream facilitates recovering it
- same keystream must be shared by encryptor and decryptor

One-time pad

- this technique is called “one-time pad”
  (sometimes one-time tape or one-time key)
  - random keystreams were written on paper pads
  - each sheet to be used, torn off, and destroyed
  - paper tapes were used later
- it is the only unbreakable cipher
- unless misused
  - Soviet codes broken due to pad/keystream re-use (Venona project)

**XOR based one-time pad**

- XOR needs a random stream producer
- rc4 is (nearly) that

---

**rc4 – a stream cipher**

rc4 serves as a keystream machine, an endless font of utility data

"RC4 generates a pseudorandom stream of bits (a keystream). As with any stream cipher, these can be used for encryption by combining it with the plaintext"
How to achieve keystream sharing

- physically secure hand delivery

- rc4 keystream reproducible on demand with a given key
  - don’t share the keystream, share the key that produces it
  - shifts (and reduces) the keystream distribution problem to a key distribution problem

Some stream ciphers

- rc4
- HC-256
- SNOW 3G
Block ciphers

- treats a block of elements, not an individual element, at a time
  - 64-bit block instead of just 1 bit e.g.
- 64-bits in, 64-bits out under influence of a key
  - for same key, same output for given input every time
- change any input bit, output altogether different
- output is a “block average” destroys frequency stats of input

Some block ciphers

- DES
- AES
- Blowfish
- rc2, rc5, rc6
- CAST
- IDEA
- serpent
Encryption modes

- used with any block cipher
- so identical blocks don’t produce identical output
- by feeding extra, varying data into each block’s processing
- sometimes called “modes of operation”

Electronic codebook mode

- compares to a traditional physical lookup codebook
  - identical input blocks produce same output block
- no inter-block dependency
- amounts to monoalphabetic substitution
- some input patterns may leak into output
  - esp for known, repeated input structures (eg. packet headers)
An ECB weakness

- The plaintext of a file encrypted as 16 blocks.

- for same key, same output with a given input
- cipher attack: substitute 12th encrypted block for 4th
  - Leslie gets same bonus as Kim

Counter mode

- maintains an incrementing counter
- block algorithm applied to counter blocks
- block algorithm *not* applied to plaintext blocks
- XOR of encrypted counter blocks applied to plaintext blocks
- identical input blocks give different output blocks
Cipher block chaining Mode

- use CBC mode to defeat the above attack
- each ciphertext block now depends on all its predecessors, not only its input block
- any change propagates through rest of document
- initialization vector for first block