Confidentiality

Review

Problem:

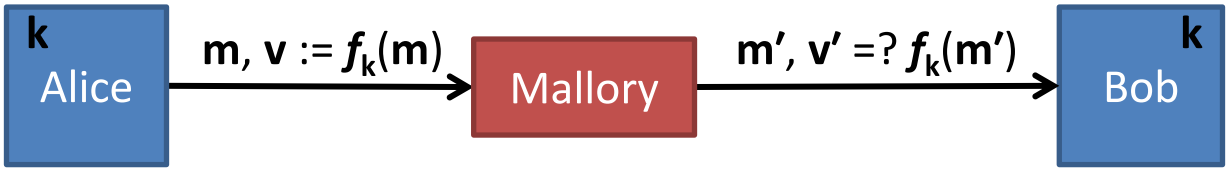
Integrity of message from Alice to Bob over an untrusted channel

Alice must append bits to message that only Alice (or Bob) can make

Solution:

Random function

Practical solution:



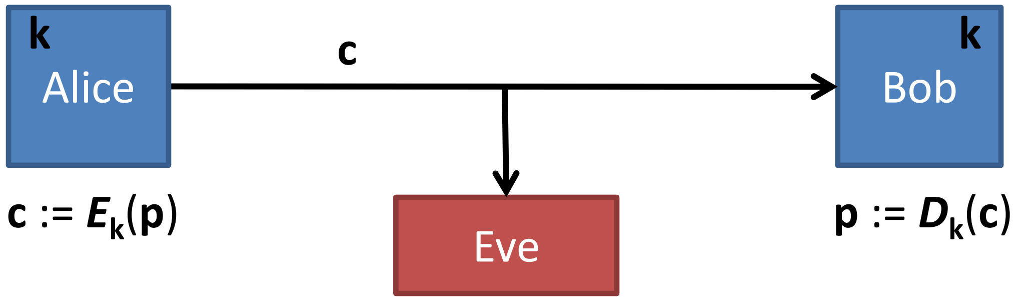
e.g. "Attack at dawn", 628369867…

(Hash-based) MAC

f_k is (we hope!) indistinguishable in practice from a random function, unless you know **k**

Review: Confidentiality

Goal: Keep contents of message **p** secret from an *eavesdropper*



Terminology

p	plaintext
c	ciphertext
k	secret key
E	encryption function
D	decryption function

Review: One-time Pad (OTP)

Alice and Bob jointly generate a secret,
very long, string of random bits
(the *one-time pad*, **k**)

To encrypt: $\mathbf{c}_i = \mathbf{p}_i$ xor $\mathbf{k}_i$
To decrypt: $\mathbf{p}_i = \mathbf{c}_i$ xor $\mathbf{k}_i$

a	**b**	**a** xor **b**
0	0	0
0	1	1
1	0	1
1	1	0

a xor **b** xor **b** = **a**
a xor **b** xor **a** = **b**

"one-time" means you should never reuse any part of the pad.
If you do:

Let $\mathbf{k}_i$ be pad bit
Adversary learns (**a** xor $\mathbf{k}_i$) and (**b** xor $\mathbf{k}_i$)
Adversary xors those to get (**a** xor **b**),
which is useful to him [How?]

Provably secure [Why?] One-time padding, to avoid brute force

Usually impractical [Why? Exceptions?]

Obvious idea: Use a **pseudorandom generator** instead of a truly random pad

(Recall: Secure **PRG** inputs a seed **k**, outputs a stream that is practically indistinguishable from true randomness unless you know **k**)

Called a **stream cipher**:

1. Start with shared secret key **k**
2. Alice & Bob each use **k** to seed the PRG
3. To encrypt, Alice XORs next bit of her generator's output with next bit of plaintext
4. To decrypt, Bob XORs next bit of his generator's output with next bit of ciphertext

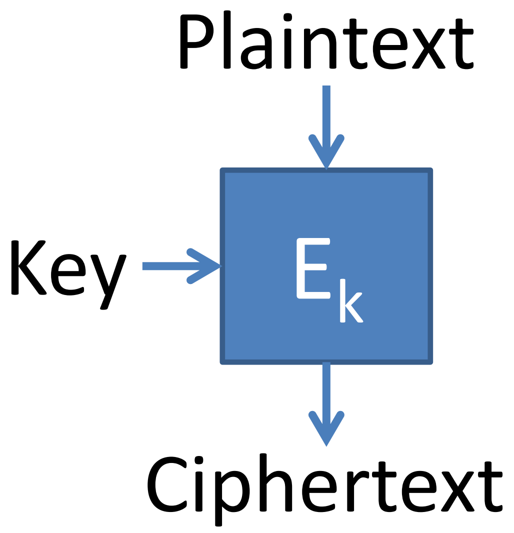
Works nicely, but: don't ever re-use the key, or the generator output bits!

Another approach: **Block Ciphers**

Functions that encrypts fixed-size blocks with a reusable key.

Inverse function decrypts when used with same key.

The most commonly used approach to encrypting for confidentiality.



A block cipher is not a pseudorandom function [Why?]

There's no inverse for pseudorandom function.
There's a inverse function of block cipher.

What we want instead:
pseudorandom permutation (PRP)

function from **n**-bit input to **n**-bit output

distinct inputs yield distinct outputs

Defined similarly to **PRF**:

practically indistinguishable from a
random permutation without secret **k**

Basic challenge: Design a hairy function
that is invertible, but only if you have the key

Minimal properties of a good block cipher:

Highly nonlinear (“confusion”)

Mixes input bits together (“diffusion”)

Depends on the key

Today's most common block cipher:

AES (**Advanced Encryption Standard**)

- Designed by NIST competition, long public comment/discussion period
- Widely believed to be secure, but we don't know how to prove it
- Variable **key size** and **block size**
- We'll use 128-bit key, 128-bit block (are also 192-bit and 256-bit versions)
- Ten **rounds**: Split **k** into ten **subkeys**, performs set of operations ten times, each with diff. subkey

Each AES round

128-bits in, 128-bit sub-key,
128-bits out

Four steps:
picture as operations on a
4x4 grid of 8-bit values

$S_{0,0}$	$S_{0,1}$	$S_{0,2}$	$S_{0,3}$
$S_{1,0}$	$S_{1,1}$	$S_{1,2}$	$S_{1,3}$
$S_{2,0}$	$S_{2,1}$	$S_{2,2}$	$S_{2,3}$
$S_{3,0}$	$S_{3,1}$	$S_{3,2}$	$S_{3,3}$

sub boxes

1. Non-linear step

   Run each byte thru a non-linear
   function (lookup table)

2. Shift step

   Circular-shift each row: **i**th row shifted by **i** (0-3)

3. Linear-mix step

   Treat each column as a 4-vector;
   multiply by a constant invertible matrix

4. Key-addition step

   XOR each byte with corresponding
   byte of round subkey

all steps are invertible

To decrypt, just undo the steps,
in reverse order

Remaining problem:
How to encrypt longer messages?

Padding

Can only encrypt in units of cipher blocksize, but message might not be multiples of blocksize

Solution: Add padding to end of message

Must be able to recognize and remove padding afterward

Common approach:
Add **n** bytes that have value **n**

[Caution: What if message ends at a block boundary?]

just add another 16 blocks to the message

… a5 6c 42 3f 42	10 10 10 … 10
message block	padding block

Cipher modes

We know how to encrypt one block,
but what about multiblock messages?

Different methods, called "cipher modes"

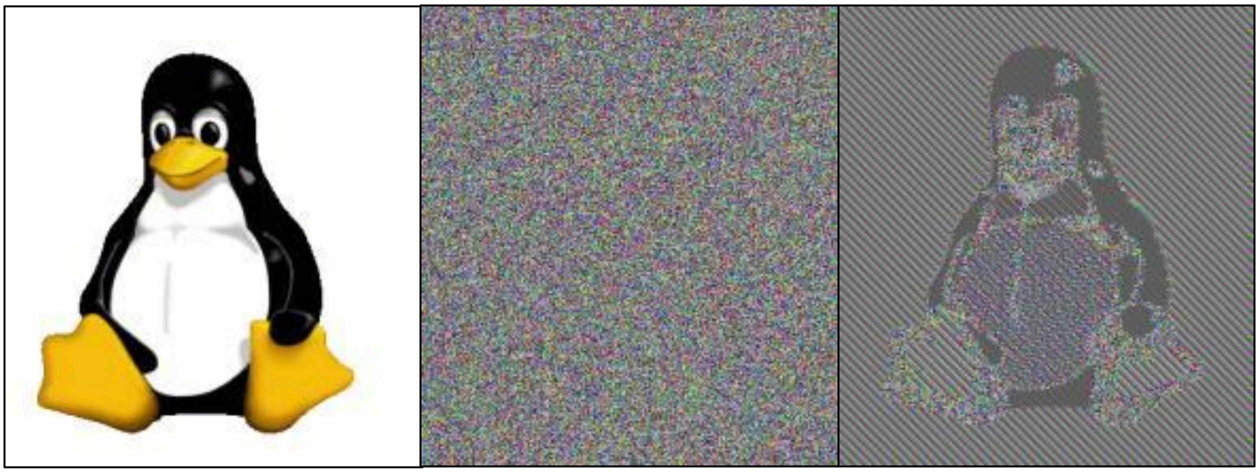
Straightforward (but bad) approach:

ECB mode (encrypted codebook)

Just encrypt each block independently

$\mathbf{C_i := \mathit{E}_k(P_i)}$

[Disadvantages?] frequency analysis still vulnerable



Plaintext Pseudorandom ECB mode

Better (and common):

CBC mode (cipher-block chaining)

Lame-CBC (for illustration only)

For each block $\mathbf{P}_i$:

1. Generate random block $\mathbf{R}_i$
2. $\mathbf{C}_i := (\mathbf{R}_i \,||\, E_k(\mathbf{P}_i \text{ xor } \mathbf{R}_i))$

[Pros and cons?]

Pros:
Is able to introduce some degree of randomness
in the resulting cipher text.

Cons:
doubled length as we need Ri has the same length
as the Pi resulting a double in size.

Real CBC

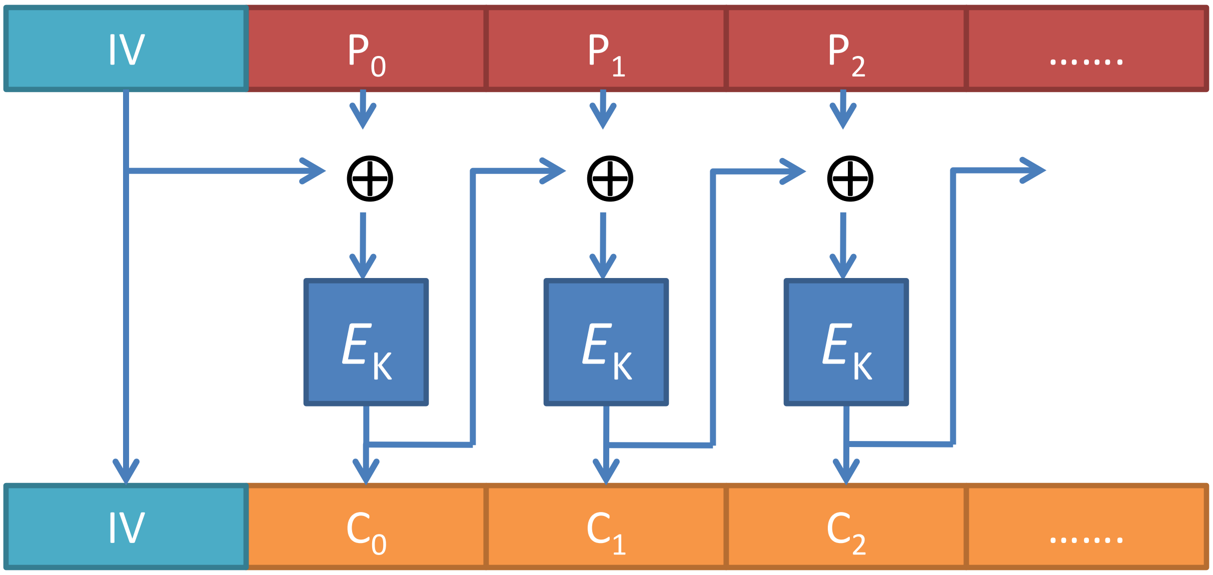
Replace $\mathbf{R}_i$ with $\mathbf{C}_{i-1}$

No need to send separately

Must still add one random $\mathbf{R}_{-1}$ to start, called "**initialization vector**" ("**IV**")

[Is CBC space-efficient?]

Illustration: CBC Encryption



[Decryption?]

Decode from C0 -> D Xor with IV => M0;
C1 -> D Xor with C0…

Other modes

OFB, CFB, etc. – used less often

Counter mode

Essentially uses block cipher as a pseudorandom generator

XOR **i**th block of message with $\boldsymbol{E}_k$(message_id || **i**)

[Why do we need message_id?]

Decode:
ID II 0 -> D Xor Co -> P0

the message and the tag are then encrypted using counter mode. One key insight is that the same encryption key can be used for both, provided that the counter values used in the encryption do not collide with the (pre-)initialization vector used in the authentication.

one way to change the cipher text to avoid collision. one way to do this is change k to k’. or we can just change the message_id to create some level of randomness to the encryption.

Building a secure channel

What if you want confidentiality and integrity at the same time?

- *Encrypt, then add integrity*, not the other way around (reasons are subtle)
- Use separate keys for confidentiality and integrity
- Need two shared keys, but only have one? That's what PRGs are for!
- If there's a reverse (Bob to Alice) channel, use separate keys for that

encryption first then integrity
to avoid attacker compromise
the communication before encryption
then encryption is useless.

Assumption we've been making so far:
Alice and Bob shared a secret key in advance

Amazing fact:
Alice and Bob can have a <u>public</u> conversation to derive a shared key!

So Far

The Security Mindset

Randomness and Pseudorandomness

Message Integrity

Confidentiality

Next week…

The single greatest advance
in the history of cryptography:
Public-key crypto