Lecture 11 – Randomness, Pseudo Randomness, and Confidentiality

Andrew Miller / Michael Bailey
University of Illinois
ECE 422/CS 461

Randomness
and
Pseudorandomness

Review

Problem:

Integrity of message sent from Alice to Bob

Append bits to message that only Alice and Bob can make

Solution:

Message Authentication Code (MAC)

Practical solution:

Hash-based MAC (HMAC) – ***HMAC-SHA256***$_k$(M)

Where do these random keys **k** come from … ?

Careful: We're often sloppy about what is "random"

True Randomness

Output of a physical process that is inherently random

Scarce, and hard to get

Pseudorandom Function (PRF)

Sampled from a family of functions using a key

Pseudorandom generator (PRG)

Takes small seed that is really random

Generates a stream (arbitrarily long sequence) of numbers that are “as good as random”

Definition: **PRG** is secure if it's indistinguishable from a random stream of bits

Similar game to PRF definition:

1. We flip a coin secretly to get a bit **b**
2. If **b**=0, let ***s*** be a truly random stream
   If **b**=1, let ***s*** be $\boldsymbol{g}_{\mathrm{k}}$ for random secret **k**
3. Mallory can see as much of the output of ***s*** as he/she wants

1. Mallory guesses **b**,
   wins if guesses correctly

g is a secure PRG if no winning strategy for Mallory*

Here's a *simple PRG that works:*

For some random k and PRF *f*,

output: $f_k(0) \;||\; f_k(1) \;||\; f_k(2) \;||\; \ldots$

Theorem: If ***f*** is a secure PRF, and ***g*** is built from ***f*** by this construction, then ***g*** is a secure PRG.

Proof: Assume ***f*** is a secure PRF, we need to show that ***g*** is a secure PRG.

Proof by contradiction:

1. Assume ***g*** is *not* secure; so Mallory can win the PRG game
2. This gives Mallory a winning strategy for the PRF game:
   a. query the PRF with inputs 0, 1, 2, …
   b. apply the PRG-distinguishing algorithm
3. Therefore, Mallory can win PRF game; this is a contradiction
4. Therefore, g is secure

Where do we get true randomness?

Want "indistinguishable from random" which means: adversary can't guess it

Gather lots of details about the computer that the adversary will have trouble guessing [Examples?]

- Problem: Adversary can predict some of this
- Problem: How do you know when you have enough randomness?

Modern OSes typically collect randomness, give you API calls to get it

- e.g., Linux:

`/dev/random` a device that gives random bits, blocks until available

`/dev/urandom` gives output of a PRG, nonblocking, seeded from `/dev/random` *eventually*

Review: Message Integrity

Integrity of message sent over an untrusted channel

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

Idealized solution: Random function

Practical solution:

e.g. “Attack at dawn”, 628369867…

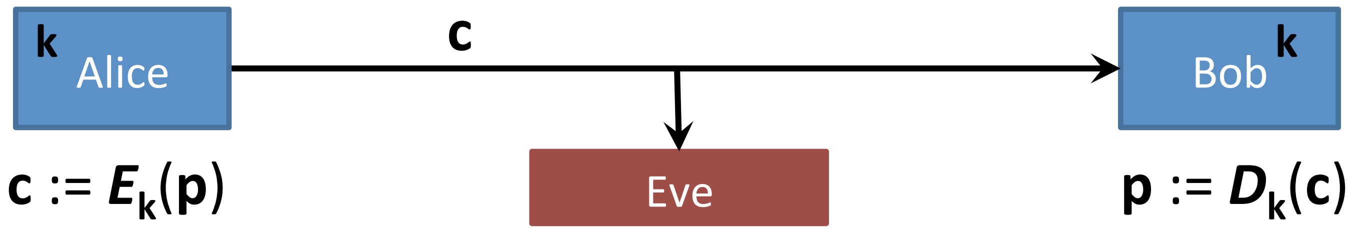
(Hash-based) MAC

$\boldsymbol{f}_{\mathbf{k}}$ is (we hope!) indistinguishable in practice from a random function, unless you know **k**

Confidentiality

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

Digression: **Classical Cryptography**

Caesar Cipher

First recorded use: Julius Caesar (100-44 BC)

Replaces each plaintext letter with one a fixed number of places down the alphabet

Encryption: $\mathbf{c_i} := (\mathbf{p_i} + \mathbf{k}) \bmod 26$

Decryption: $\mathbf{p_i} := (\mathbf{c_i} - \mathbf{k}) \bmod 26$

e.g. (**k**=3):

```
    Plain:   ABCDEFGHIJKLMNOPQRSTUVWXYZ
    +Shift:  33333333333333333333333333
=Cipher:     DEFGHIJKLMNOPQRSTUVWXYZABC

    Plain:   fox   go wolverines
    +Key:    333   33 3333333333
=Cipher:     ira   jr zroyhulqhv
```

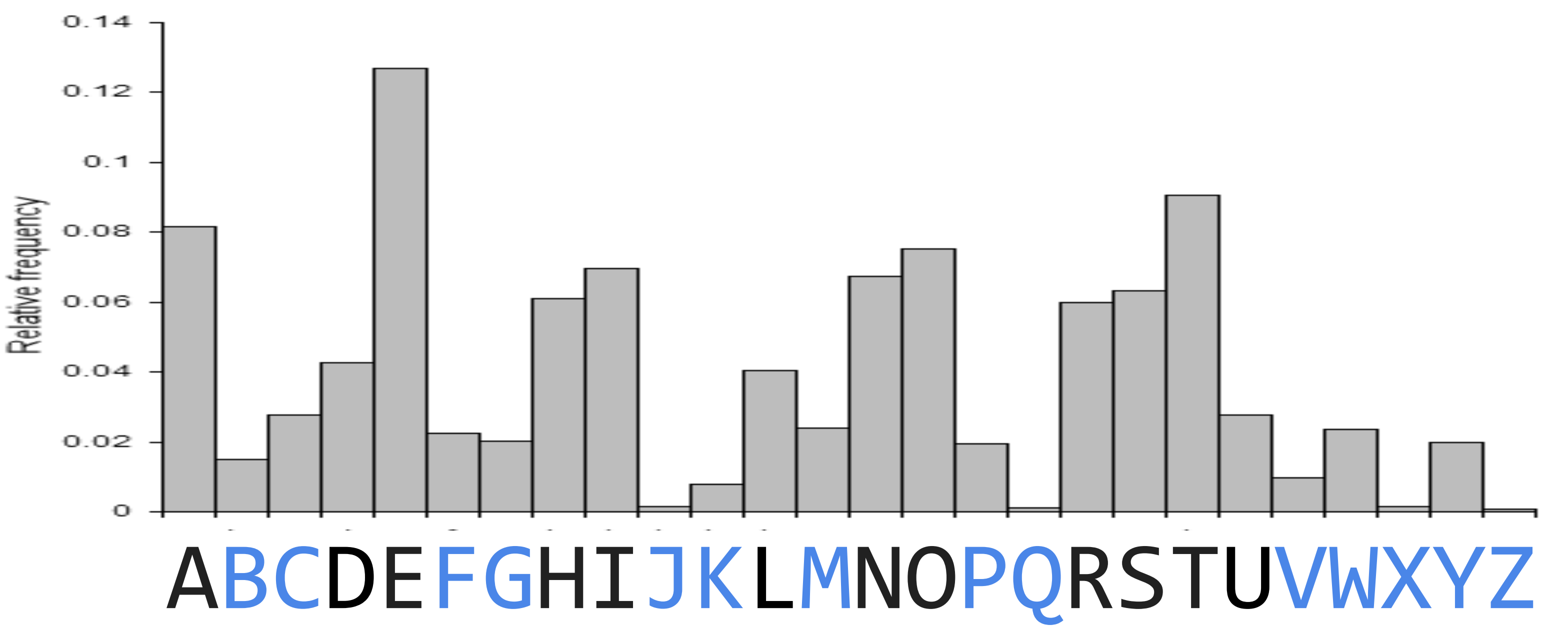
Cryptanalysis of the Caesar Cipher

Only 26 possible keys:

Try every possible **k** by “*brute force*”

Can a computer recognize the right one?

Use *frequency analysis*: English text has distinctive letter frequency distribution



Later advance: Vigènere Cipher

First described by Bellaso in 1553,
later misattributed to Vigenère

Called « le chiffre indéchiffrable »
(“the indecipherable cipher”)

Encrypts successive letters using a sequence of Caesar ciphers determined by the letters of a keyword

For an **n**-letter keyword **k**,

Encryption: $\mathbf{c}_\mathbf{i} := (\mathbf{p}_\mathbf{i} + \mathbf{k}_{\mathbf{i} \bmod \mathbf{n}}) \bmod 26$

Decryption: $\mathbf{p}_\mathbf{i} := (\mathbf{c}_\mathbf{i} - \mathbf{k}_{\mathbf{i} \bmod \mathbf{n}}) \bmod 26$

Example: **k**=ABC (i.e. $\mathbf{k}_0$=0, $\mathbf{k}_1$=1, $\mathbf{k}_2$=2)

```
Plain:    bbbbbb    amazon
+Key:     012012    012012
=Cipher:  bcdbcd    anczpp
```

Cryptanalysis of the Vigènere Cipher

Simple, if we know the keyword length, **n**:

1. Break ciphertext into **n** slices
2. Solve each slice as a Caesar cipher

How to find **n**? One way: Kasiski method

Published 1863 by Kasiski (earlier known to Babbage?)

Repeated strings in long plaintext
will sometimes, by coincidence,
be encrypted with same key letters

Plain: **CRYPTO**ISSHORTFOR**CRYPTO**GRAPHY
+Key: ABCDABCDABCDABCDABCDABCD
=Cipher: **CSASTP**KVSIQUTGQU**CSASTP**IUAQJB

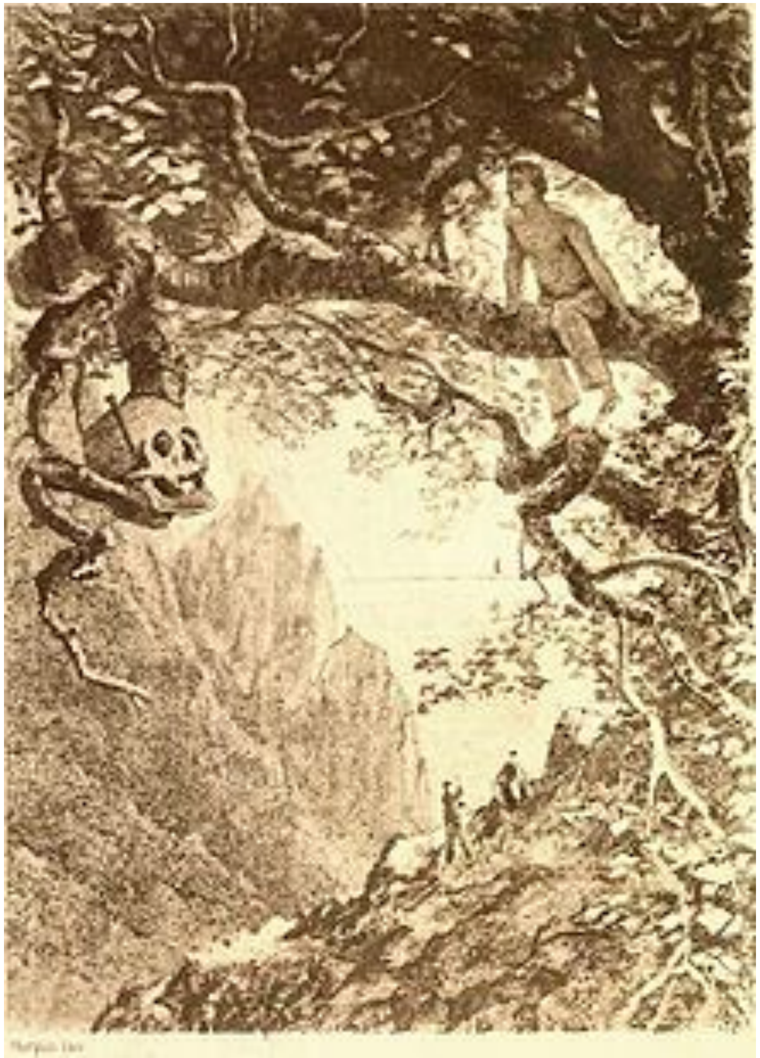
Distance between repeated strings in ciphertext is likely a multiple of key length e.g., distance 16 implies **n** is 16, 8, 4, 2, 1

[What if key is as long as the plaintext?]

Another example of “pre-modern” crypto:

“The Gold Bug”

By Edgar Allen Poe, 1843



Kerckhoff’s Principles

1st: The system must be practically, if not mathematically, indecipherable;

2nd: The system must not require secrecy and must not cause inconvenience should it fall into the hands of the enemy;

3rd: The key must be able to be used in communiques and retained without the help of written notes, and be changed or modified at the discretion of the correspondents;

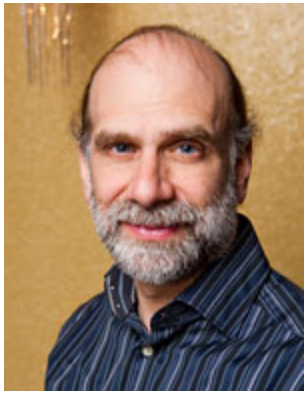
4th: The system must be compatible with telegraphic communication;

5th: The system must be portable, and remain functional without the help of multiple people;

6th: Finally, it’s necessary, given the circumstances in which the system will be applied, that it’s easy to use, is undemanding, not overly stressful, and doesn’t require the knowledge and observation of a long series of rules

“Schneier's law”

“Any fool can invent a cipher that he himself cannot break.”



https://www.schneier.com/crypto-gram/archives/1998/1015.html#cipherdesign

One-time Pad (OTP)

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

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

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

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 ($\mathbf{a}$ xor $\mathbf{k}_i$) and ($\mathbf{b}$ xor $\mathbf{k}_i$)

Adversary xors those to get (**a** xor **b**), which is useful to him [How?]

Provably secure [Why?]

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

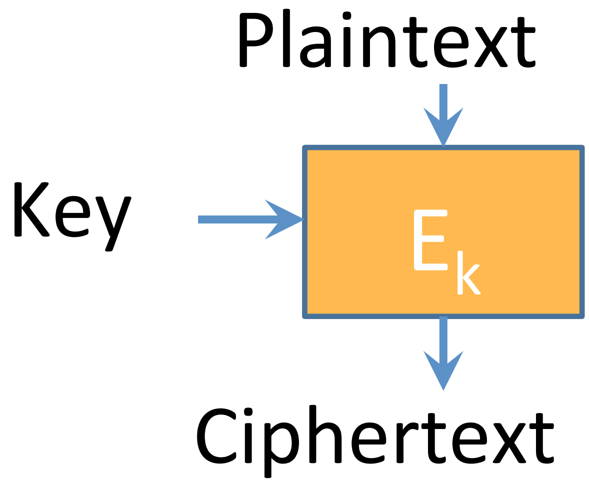
Works nicely, but: don't *ever* reuse the key, or the generator output bits

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 <u>not</u> a pseudorandom function [Why?]

What we want instead:

pseudorandom permutation (PRP)

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

distinct inputs yield distinct outputs (one-to-one)

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

Definition: a cipher is "Semantically Secure"

Similar game to PRF/PRG/PRP definition:

1. We flip a coin secretly to get a bit **b**, random secret **k**
2. Mallory chooses arbitrary $\mathbf{m_i}$ in **M**, gets to see $\boldsymbol{Enc_k}(\mathbf{m_i})$
3. Mallory chooses two messages $\mathbf{m'_0}$ and $\mathbf{m'_1}$ not in **M**
4. If **b**=0, let **c** be $\boldsymbol{Enc_k}(\mathbf{m'_0})$
   If **b**=1, let **c** be $\boldsymbol{Enc_k}(\mathbf{m'_1})$
5. Mallory can see **c**
6. Mallory guesses **b**, wins if guesses correctly

We can prove this follows from a PRP definition. **[Fun to try!]**

Also known as: IND-CPA "Chosen plaintext attack"

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

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

Four steps:

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

1. Non-linear step

   Run each byte through 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 constant invertible matrix

4. Key-addition step

   XOR each byte with corresponding byte of round subkey

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?]

Cipher modes of operation

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

$\mathbf{C_i} := \boldsymbol{E}_{\mathbf{k}}(\mathbf{P_i})$

[Disadvantages?]

Cipher modes of operation

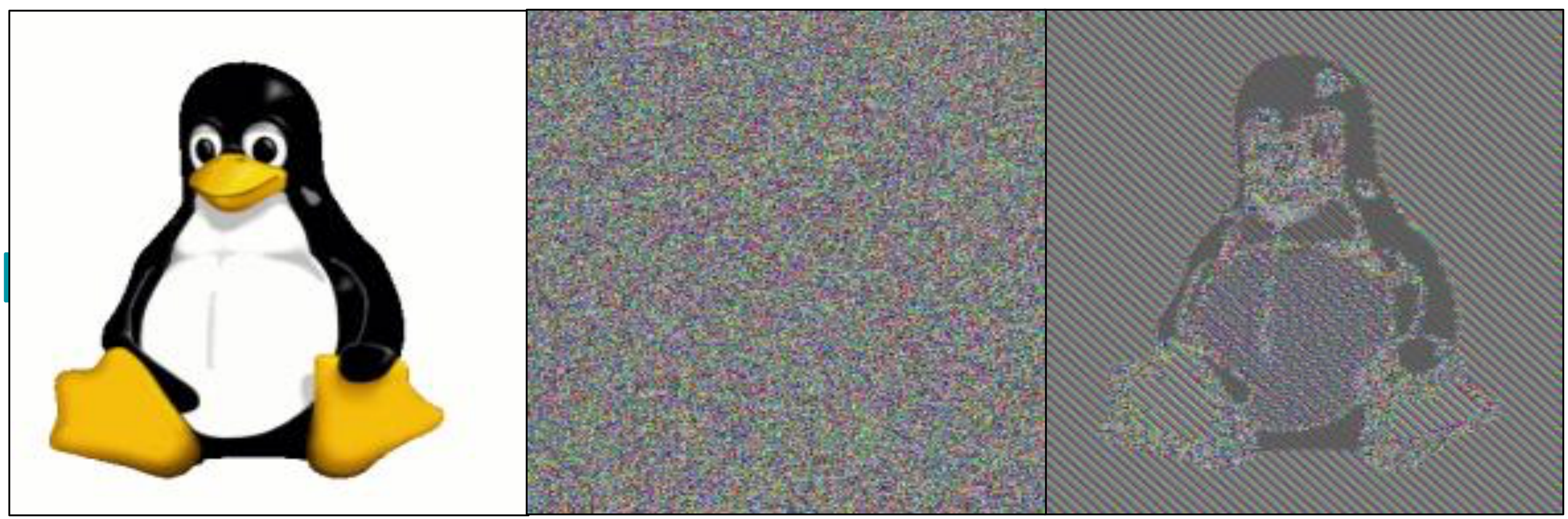
We know how to encrypt one block,

but what about multiblock messages?

Different methods, called “cipher modes”

Straightforward (but bad) approach:

ECB mode (**encrypted codebook**)



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. $\mathbf{C_i}$:= ($\mathbf{R_i}$ || $\boldsymbol{E}_k$($\mathbf{P_i}$ xor $\mathbf{R_i}$))

[Pros and cons?]

Real CBC

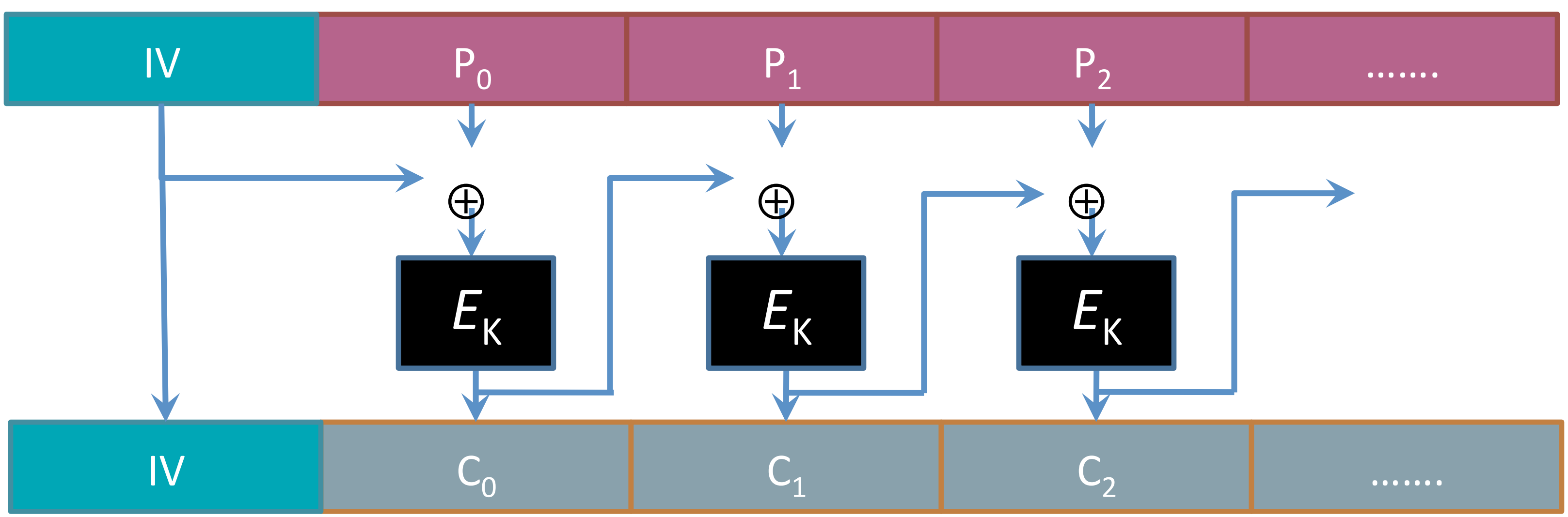
Replace $\mathbf{R_i}$ with $\mathbf{C_{i-1}}$

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



Using OpenSSL to do AES encryption from the command line

```
$ KEY=$(openssl rand -hex 16)
```

Generates a random string

```
$ openssl aes-256-cbc -in mymsg.txt -out mymsg.enc
-p -K ${KEY} -iv $(openssl rand -hex 16)
    key=8582D9E1A36DA4DB065394FB1F401DB3
    iv =DBB272FE6486C4D9B09DBE464E080468
```

Prints the key and IV

```
$ openssl aes-256-cbc -d -in mymsg.enc -out mymsg.txt
-K ${KEY} -iv <iv from above>
```

- By default, uses the standard padding described earlier
- Unfortunately, you have to handle prepending/extracting the IV on your own

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 $\boldsymbol{E}_{\mathbf{k}}$(message_id || **i**)

[Why do we need message_id?]

[Do we need a message_id for CBC mode?]

[Recover after errors? Decrypt in parallel?]

What is **NOT** covered by Semantic Security?

- "Malleability" attacks

Given just some ciphertexts, can the attacker create new ciphertexts that Bob decrypts the wrong value?

- Encryption does NOT IMPLY integrity!

Often you really want both ("authenticated encryption")

- Chosen Ciphertext attacks

The "semantic security" definition does not allow the adversary to see decryptions of (potentially garbage) ciphertexts chosen by the adversary

- Solution: Encrypt-then-MAC

Assumption we've been making so far:

Alice and Bob ***shared a secret key*** in advance

Amazing fact:

Alice and Bob can have a **public** conversation to derive a shared key!

So Far

Message Integrity

Randomness /Pseudorandomness

Confidentiality: Stream Ciphers, Block Ciphers

Wednesday…

Key Exchange, Key Management, Public Key Crypto