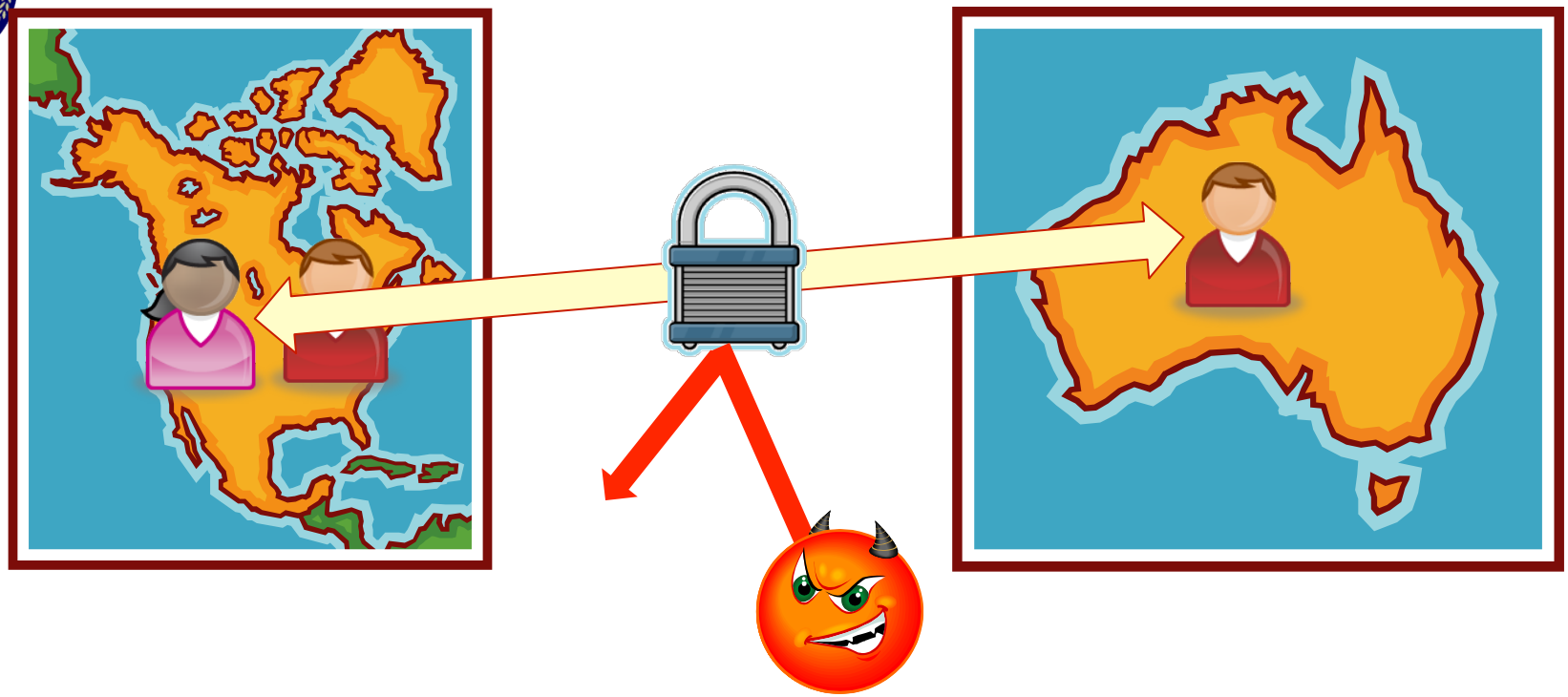
Applied Cryptography and Network Security

Adam J. Lee
adamlee@cs.pitt.edu
6111 Sennott Square

Lecture #4: Symmetric Key Cryptography
January 16, 2014

University of Pittsburgh

A Motivating Scenario



How can Alice and Bob communicate over an untrustworthy channel?

Need to ensure that:

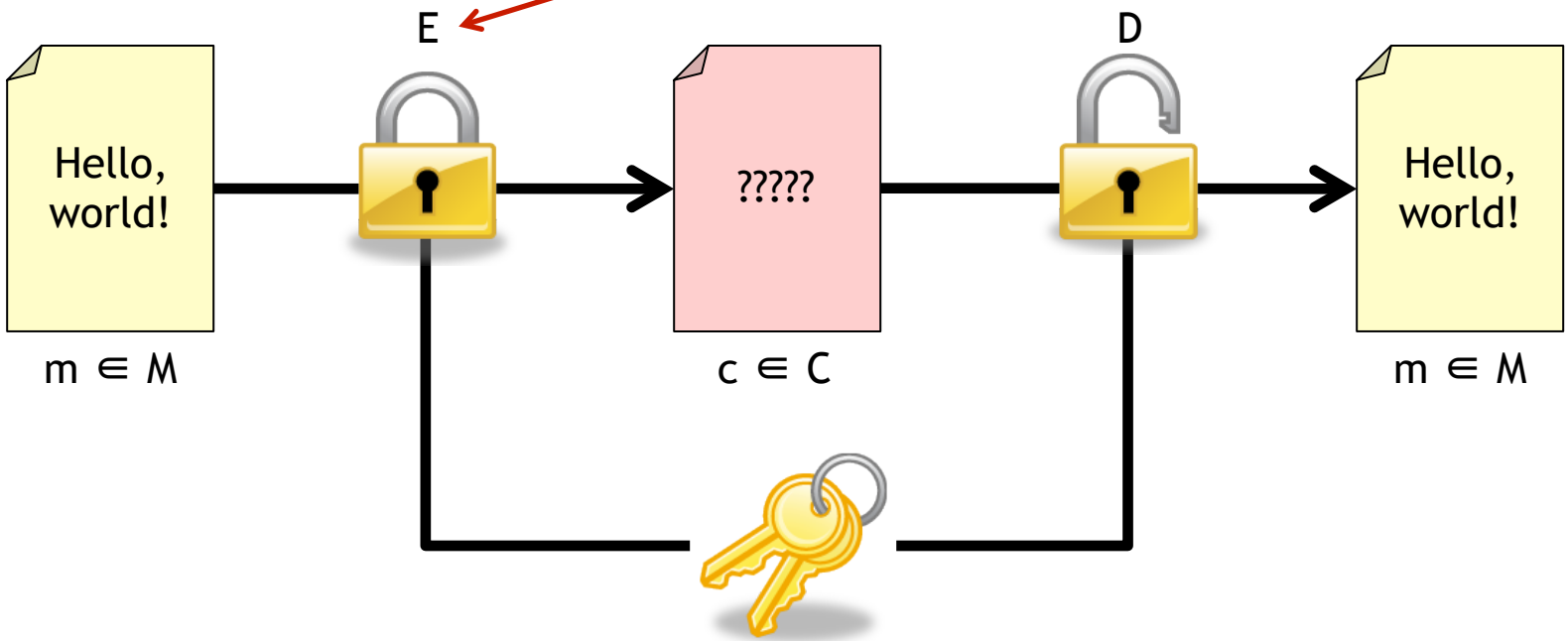
1. Their conversations remain secret (confidentiality)
2. Modifications to any data sent can be detected (integrity)

Recall our cryptographic model…

Formally, a cryptosystem can be represented as the 5-tuple (E, D, M, C, K)

- M is a message space
- K is a key space
- E : M × K → C is an encryption function
- C is a ciphertext space
- D : C × K → M is a decryption function

Today's focus is on symmetric key encryption



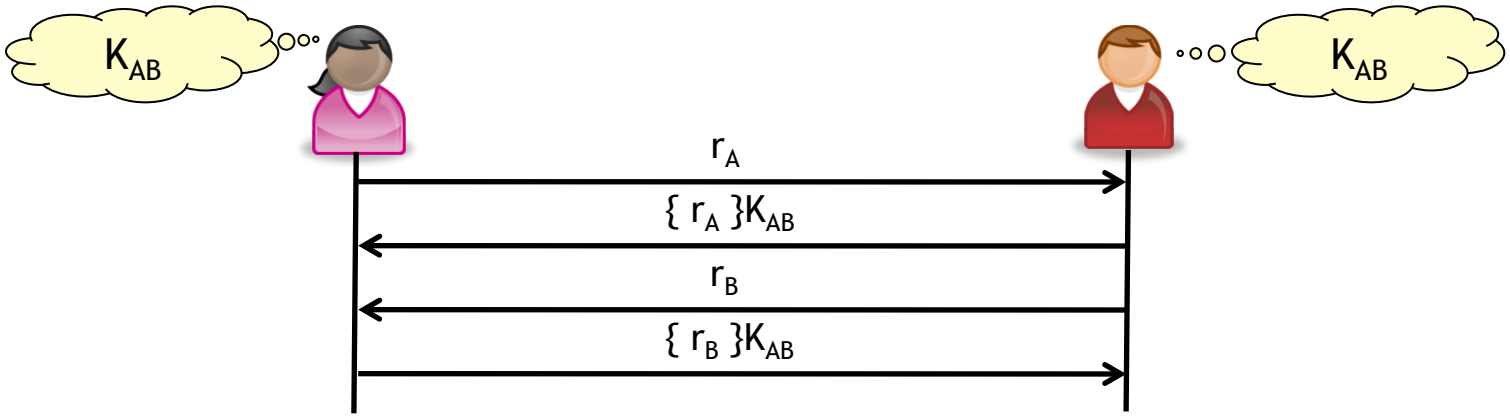
Why study symmetric key cryptography?

Rather obvious good uses of symmetric key cryptography include:

- Transmitting data over insecure channels
  - SSL, SSH, etc.
- Securely storing sensitive data in untrusted places
  - Malicious administrators
  - Cloud computing
  - …
- Integrity verification and tamper resistance

We'll go over these types of protocols in gory detail next month…

Authentication is (perhaps) a less obvious use of symmetric key crypto



The classical algorithms that we studied last time are examples of symmetric key ciphers

Unfortunately, most of these ciphers offer essentially no protection in modern times

The obvious exception is the one time pad which offers perfect security from an information theory perspective

- Namely, a single ciphertext of length *n* can decrypt to any message of length up to *n*.
- More formally, H(m) = H(m | c)

However, the large amount of key material required by the one time pad is a hindrance to its use for many practical purposes

- To transmit a message of length *n*, you need a key of length *n*
- If you have a secure channel to transmit *n* bits of key, why not use it to transmit *n* bits of message instead?

In modern cryptography, algorithms use a fixed-length key to encipher variable length data

In an ideal world, we would like to have the perfect security guarantees of the one time pad, without the hassle of requiring our key length to equal our message length

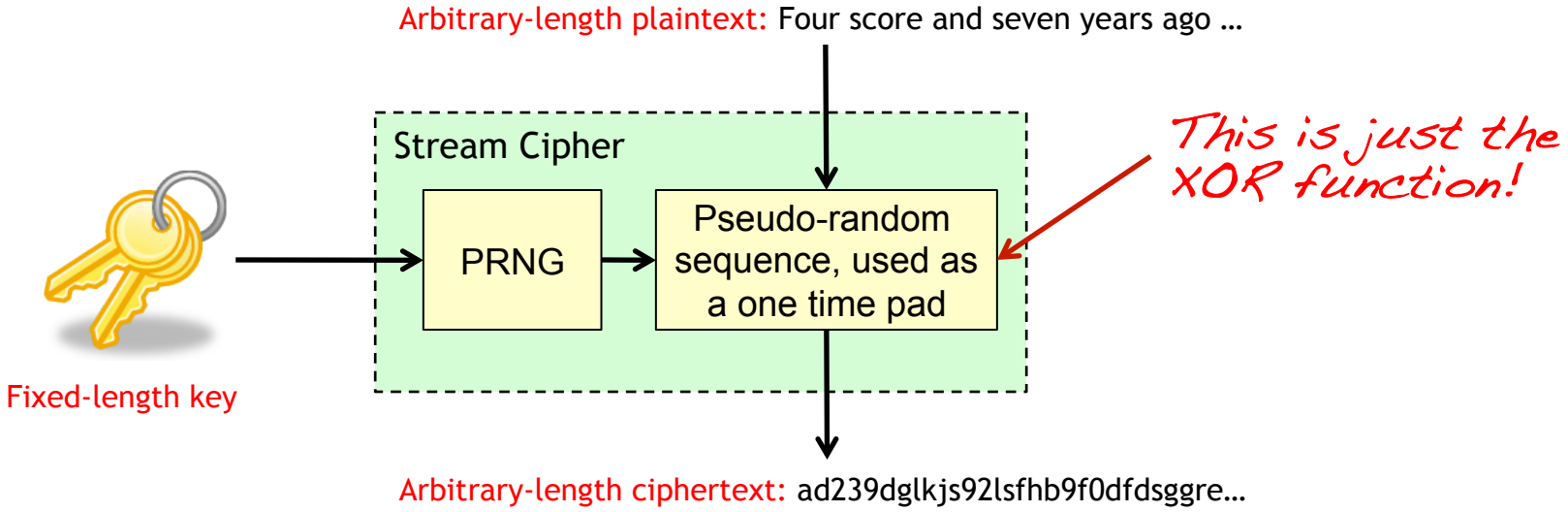
This is clearly very difficult!

However, modern cryptographers have developed many algorithms that give good security using very small keys

Today, we'll study two classes of symmetric key algorithms

- Stream ciphers
  - RC4, SEAL, etc.
- Block ciphers
  - DES, TDES, AES, Blowfish, etc.

Stream ciphers work like a one time pad



The secrecy of a stream cipher rests entirely on PRNG "randomness"

Often, we see stream ciphers used in communications hardware

- Single bit transmission error effects only single bit of plaintext
- Low transmission delays
  - Key stream can (sometimes) be pre-generated and buffered
  - Encryption is just an XOR
  - No buffering of data to be transmitted

Two types of stream ciphers

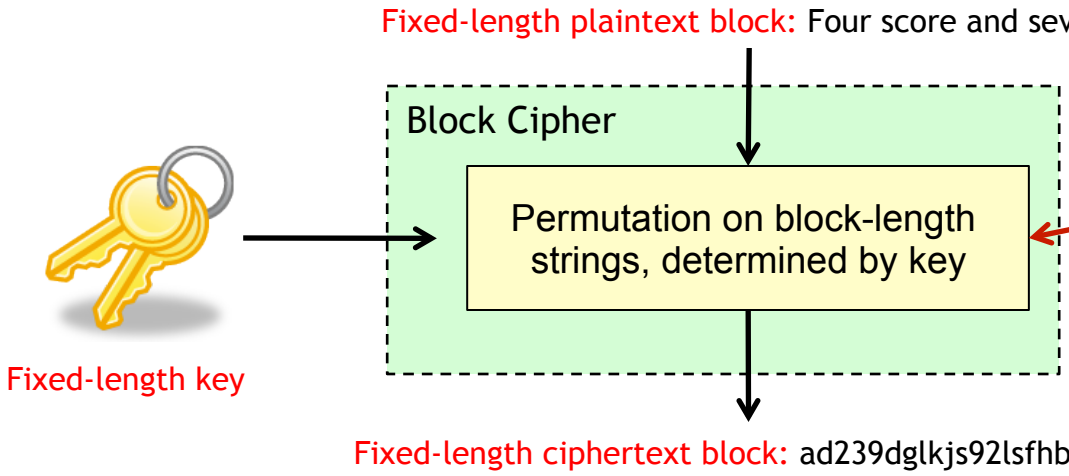
In a synchronous stream cipher, the key stream is generated independently of the ciphertext

- ***Advantages***
  - Do not propagate transmission errors
  - Prevent insertion attacks
  - Key stream can be pre-generated
- ***Disadvantage:*** May need to change keys often if periodicity of PRNG is low

In a self-synchronizing stream cipher, the key stream is a function of some number of ciphertext bits

- ***Advantages***
  - Decryption key stream automatically synchronized with encryption key stream after receiving *n* ciphertext bits
  - Less frequent key changes, since key stream is a function of key and ciphertext
- ***Disadvantage:*** Vulnerable to replay attack

Block ciphers are more commonly used than stream ciphers



Designing good block ciphers is as much of a black art as it is a science…

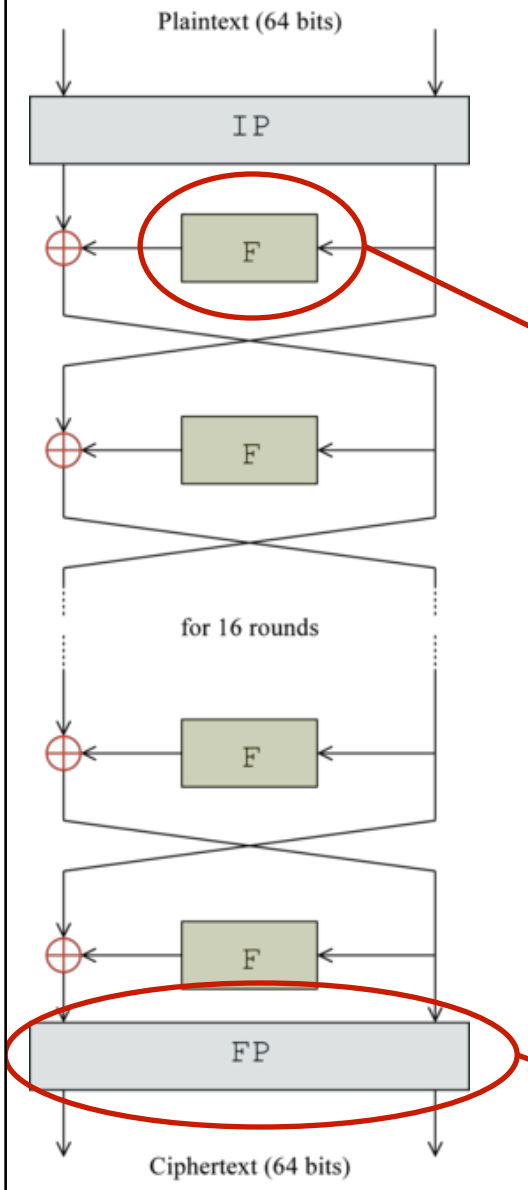
Block ciphers operate on fixed-length blocks of plaintext

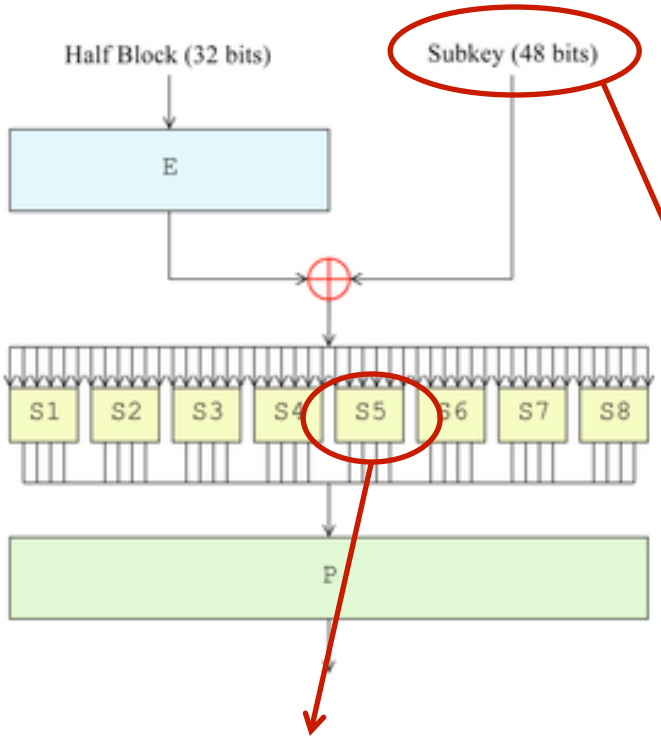
- Typical block lengths: 40, 56, 64, 80, 128, 192 and 256 bits

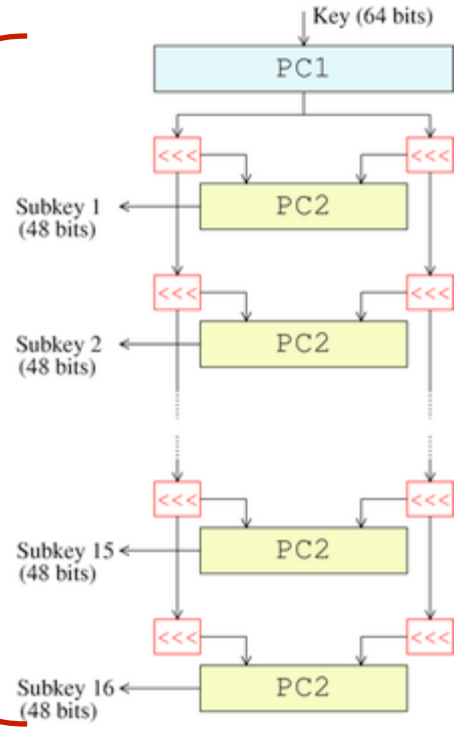
Often, block ciphers apply several rounds of a simpler function

- Most block ciphers can be categorized as Feistel networks
  - Bit shuffling, non-linear substitution, and linear mixing (XOR)
  - Confusion and diffusion *a la* Claude Shannon
- ***Example:*** DES is a Feistel network that uses 16 rounds

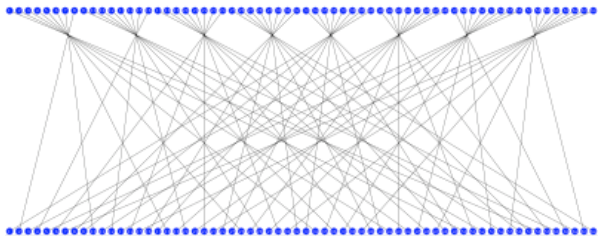
Example: DES





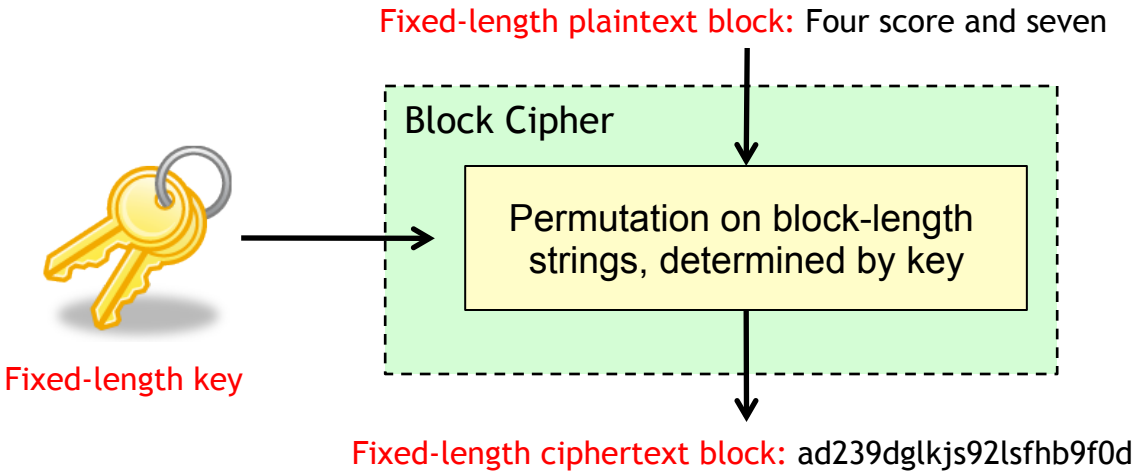


S_5		Middle 4 bits of input															
		0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Outer bits	00	0010	1100	0100	0001	0111	1010	1011	0110	1000	0101	0011	1111	1101	0000	1110	1001
	01	1110	1011	0010	1100	0100	0111	1101	0001	0101	0000	1111	1010	0011	1001	1000	0110
	10	0100	0010	0001	1011	1010	1101	0111	1000	1111	1001	1100	0101	0110	0011	0000	1110
	11	1011	1000	1100	0111	0001	1110	0010	1101	0110	1111	0000	1001	1010	0100	0101	0011



Images from Wikipedia

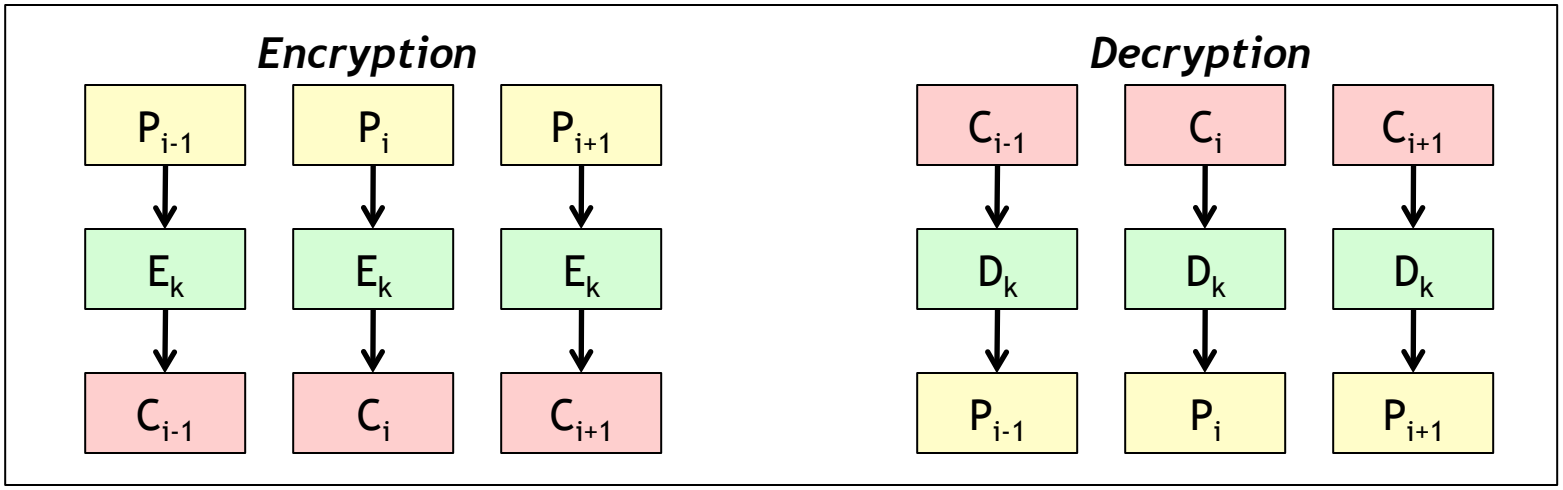
Block Cipher Modes of Operation



Question: What happens if we need to encrypt more than one block of plaintext?

Block ciphers have many modes of operation

The most obvious way of using a block cipher is called electronic codebook mode (ECB)



Benefit: Errors in ciphertext do not propagate past a single block

What is wrong with ECB?

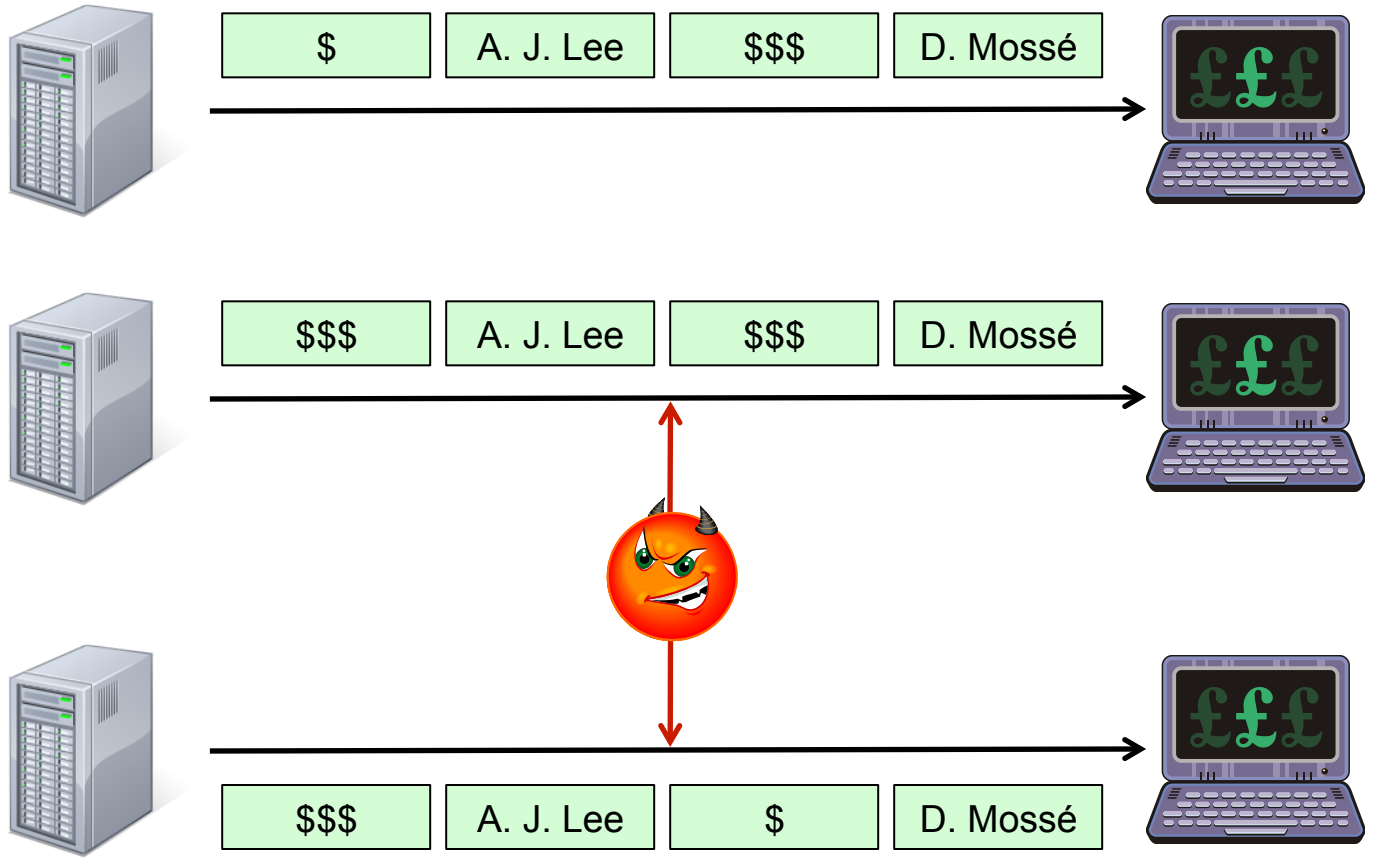
- Known plaintext/ciphertext pairings
- Block replay attacks

This is called a code book!

In general, using ECB mode is not a great idea…

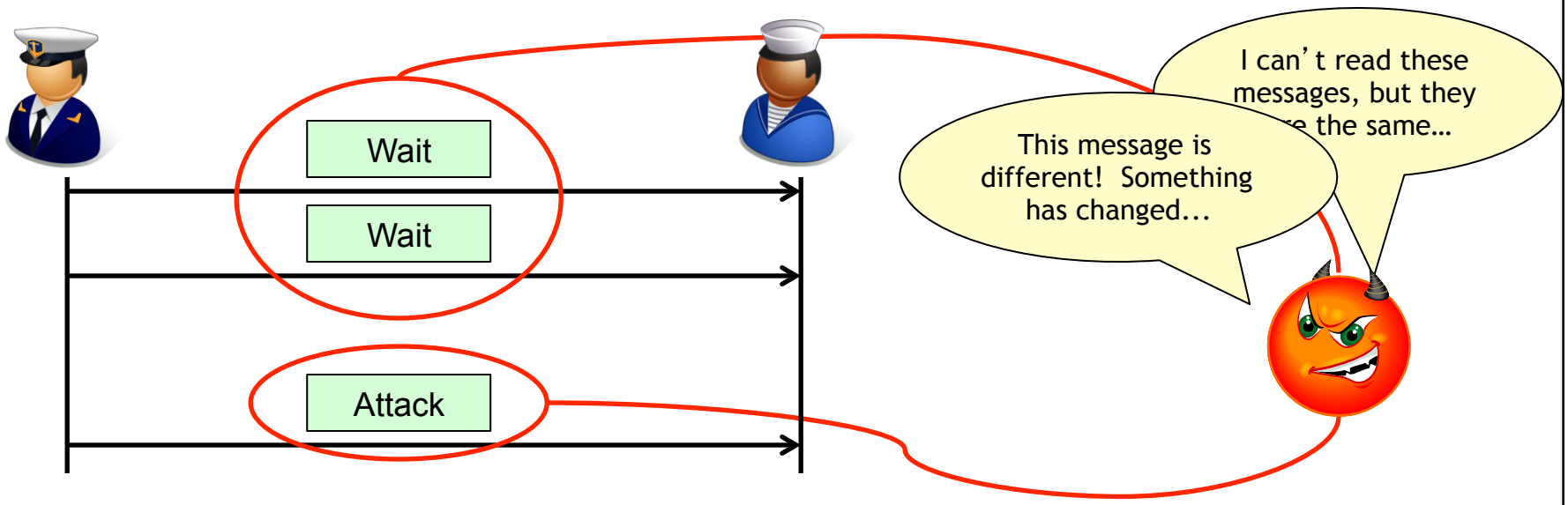
The use of ECB mode can lead to block replay or substitution attacks

Example: Payroll data transmitted using ECB



Why is the ability to build a codebook dangerous?

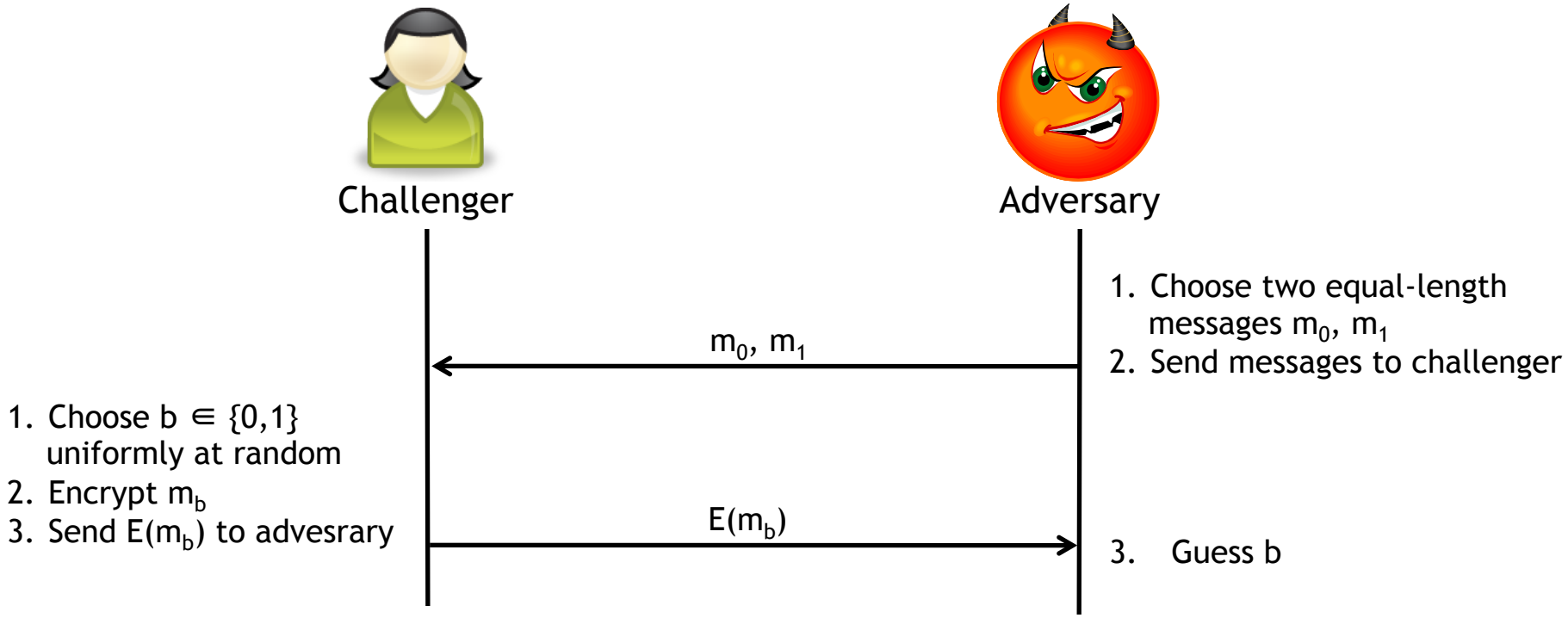
Observation: When using ECB, the same block will always be encrypted the same way



To protect against this type of guessing attack, we need our cryptosystem to provide us with semantic security.

Semantic Security

The semantic (in)security of a cipher can be established as follows:



The adversary wins if he has a non-negligible advantage in guessing b. More concretely, he wins if $P[b' = b] > \frac{1}{2} + \varepsilon$.

If the adversary does not have an advantage, the cipher is said to be semantically secure.

The "covert channel" attack shows up because block ciphers running in ECB mode are not semantically secure!

Question: Can you demonstrate this?

Challenger

Adversary

1. m_0 = "HelloHello"
   m_1 = "HelloWorld"
2. Send messages to challenger

m_0, m_1

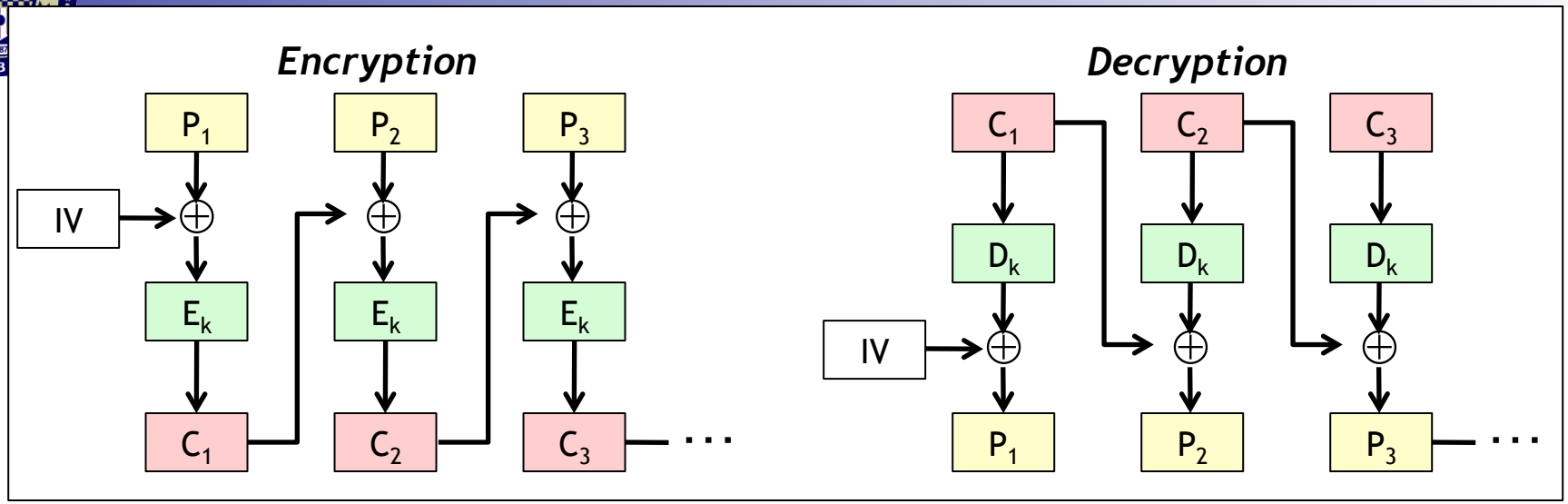
1. Choose b ∈ {0,1} uniformly at random
2. Encrypt m_b using ECB
3. Send $E(m_b)$ to advesrary

$E(m_b)$

3. If first block = second block, then b = 0. Otherwise, b = 1.

Guess what? P[b' = b] = 1!

Cipher Block Chaining (CBC) mode addresses the problems with ECB



In CBC mode, each plaintext block is XORed with the previous ciphertext block prior to encryption

- $C_i = E_k(P_i \oplus C_{i-1})$
- $P_i = C_{i-1} \oplus D_k(C_i)$

Need to encrypt a random block to get things started

- This initialization vector needs to be random, but not secret (Why?)

CBC eliminates block replay attacks

- Each ciphertext block depends on previous block

Semantic security, redux

Challenger

Adversary

1. m_0 = "HelloHello"
   m_1 = "HelloWorld"
2. Send messages to challenger

m_0, m_1

1. Choose b ∈ {0,1} uniformly at random
2. Generate a random IV
3. Encrypt m_b using CBC
4. Send $E(m_b)$ to adversary

$E(m_b)$

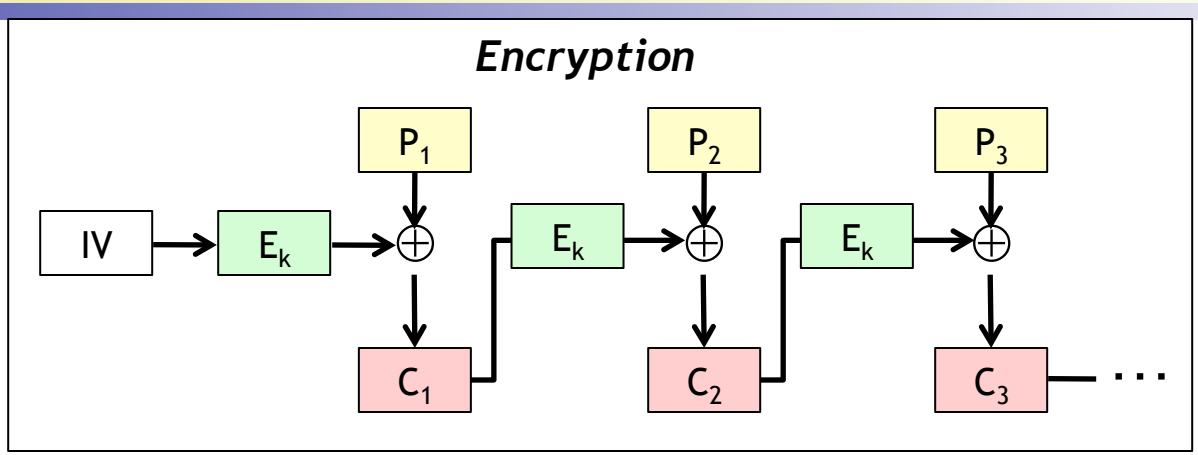
3. If first block = second block, then b = 0. Otherwise, b = 1.

Note that the adversary's "trick" does not work anymore (Why?)

- $c_{01} = E(IV \oplus m_{01})$
- $c_{02} = E(c_{01} \oplus m_{02})$

Essentially, the IV randomizes the output of the game, even if it is played over multiple rounds

Cipher Feedback Mode (CFB) can be used to construct a self-synchronizing stream cipher from a block cipher



To generate an *n*-bit CFB based upon an *n*-bit block cipher algorithm, as above, we have that:

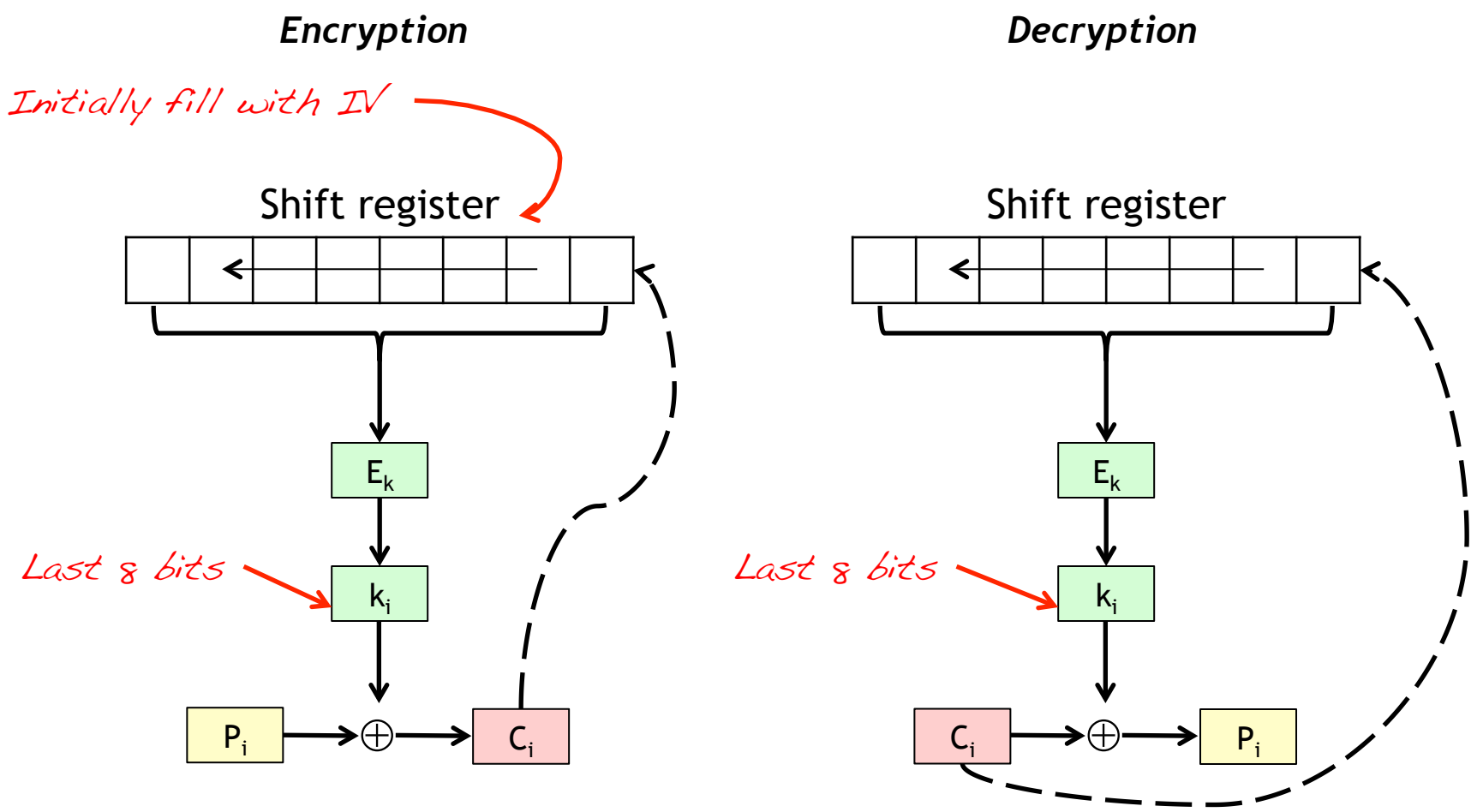
- $C_i = P_i \oplus E_k(C_{i-1})$
- $P_i = C_i \oplus E_k(C_{i-1})$

What is really interesting is that this technique can be used to develop an *m*-bit cipher based upon an *n*-bit block cipher, where $m \leq n$ by using a shift-register approach

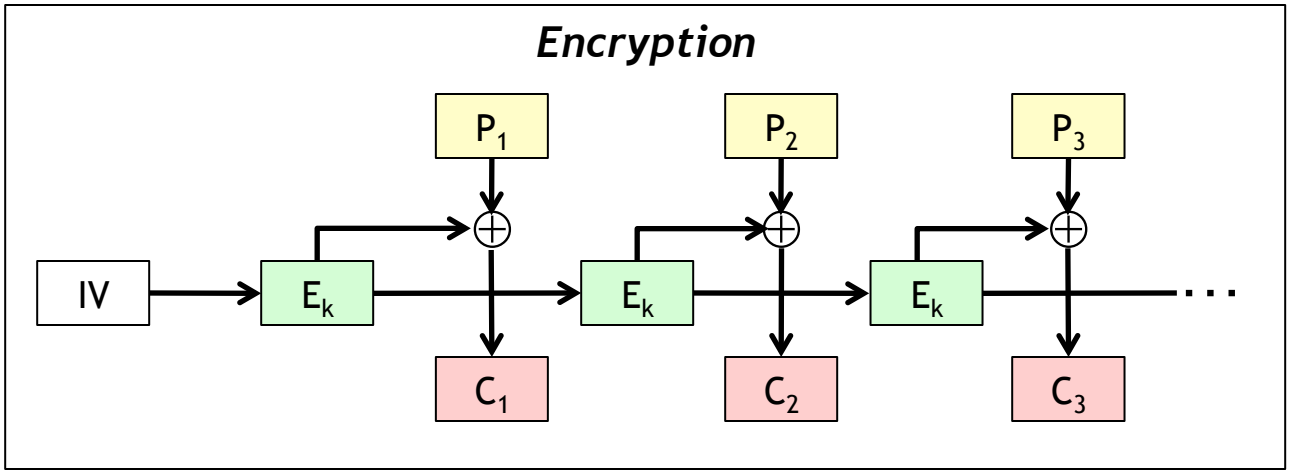
This is great, since we don't need to wait for *n* bits of plaintext to encrypt!

- ***Example:*** Typing at a terminal

Using an n-bit cipher to get an m-bit cipher (m < n)



Output Feedback Mode (OFB) can be used to construct a synchronous stream cipher from a block cipher



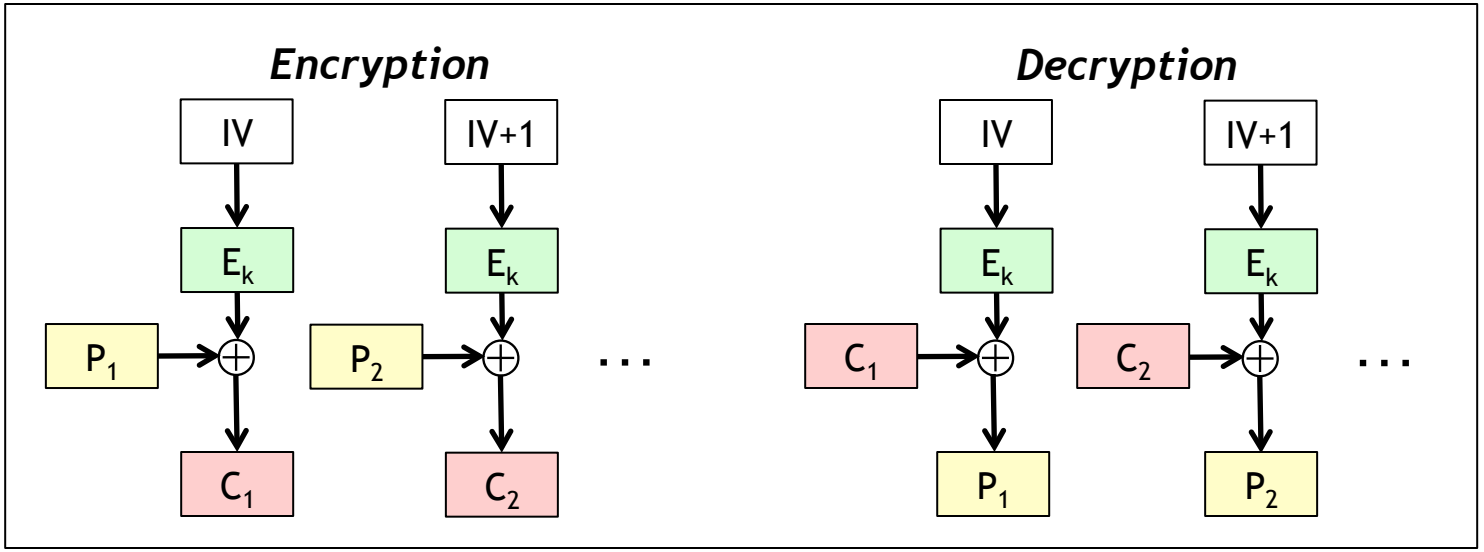
How does this work?

- $C_i = P_i \oplus S_i$, $S_i = E_k(S_{i-1})$
- $P_i = C_i \oplus S_i$, $S_i = E_k(S_{i-1})$

Benefit: Key stream generation can occur offline

Pitfall: Loss of synchronization is a killer…

Counter mode (CTR) generates a key stream independently of the data



Pros:

- We can do the expensive cryptographic operations offline
- Encryption/decryption is just an XOR
- It is possible to encrypt/decrypt starting anywhere in the message

Cons:

- Don't use the same (key, IV) for different files (Why?)

CTR mode has some interesting applications

Example: Accessing a large file or database

Operation: Read block number *n* of the file

- CTR: One encryption operation is needed
  - $p_n = c_n \oplus E(IV + n)$
- CBC: One decryption operation is needed
  - $p_n = c_{n-1} \oplus D(c_n)$

In most symmetric key ciphers encryption and decryption have the same complexity

If n is large, this is problematic…

Operation: Update block *k* of *n*

- CTR: One encryption operation is needed
  - $c_k = p_k \oplus E(IV + k)$
- What about CBC?
  - First, we need to decrypt all blocks after *k* (*n* – *k* decryptions)
  - Then, we need to encrypt blocks *k* through *n* (*n* – *k* + 1 encryptions)

Operation: Encrypt all *n* blocks of a file on a machine with *c* cores

- CTR: O(*n* / *c*) time required, as cores can operate in parallel
- CBC: O(*n*) time required on one core…

So… Which mode of operation should I use?

Unless you are encrypting short, random data (e.g., a cryptographic key) do not use ECB!

Use CBC if either:

- You are encrypting files, since there are rarely errors on storage devices
- You are dealing with a software implementation

CFB (usually 8-bit CFB) is the best choice for encrypting streams of characters entered at, e.g., a text terminal

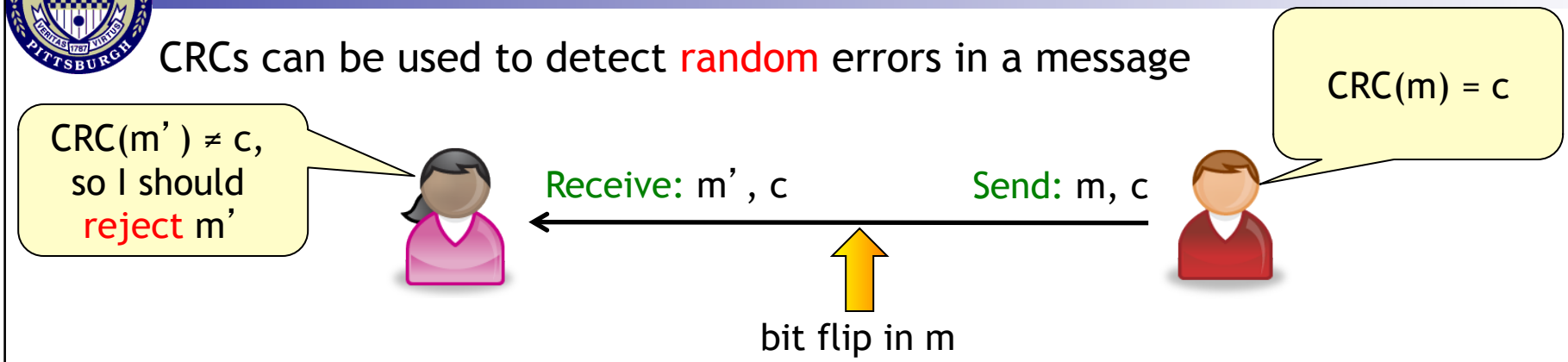
In error prone environments, OFB is probably your best choice

---

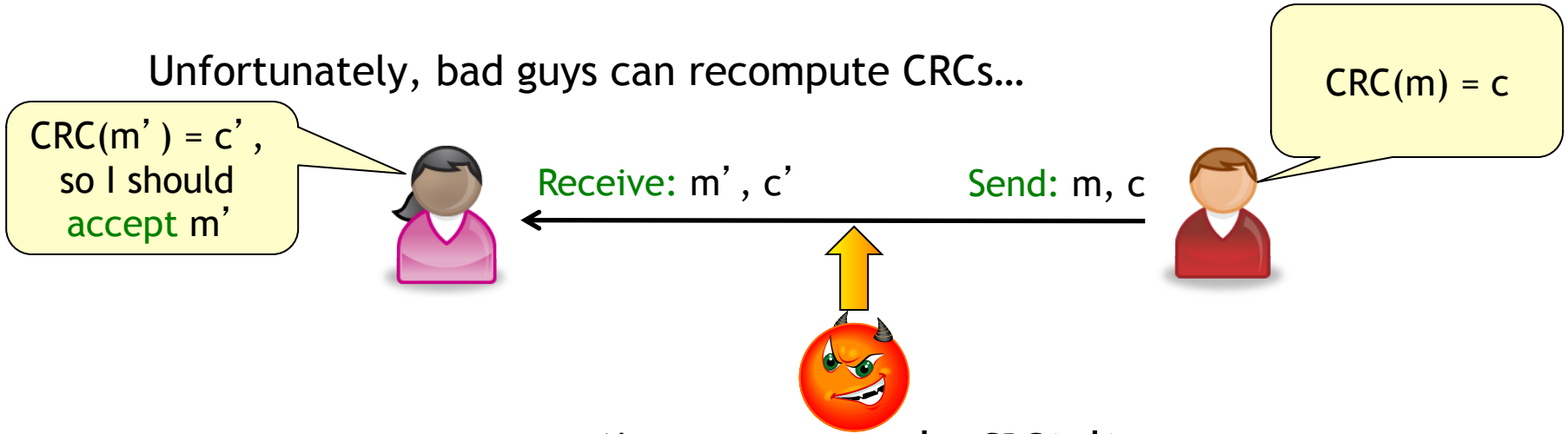
Want more information? See chapter 9 of *Applied Cryptography*.

Encryption does not guarantee integrity/authenticity

CRCs can be used to detect random errors in a message

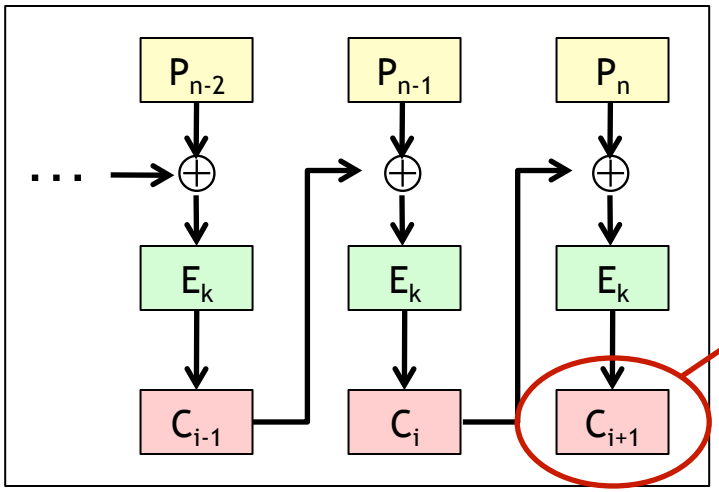


Unfortunately, bad guys can recompute CRCs…



Solution: Cryptographic message authentication codes (MACs)

The CBC residue of an encrypted message can be used as a cryptographic MAC



How does this work?

- Use a block cipher in CBC mode to encrypt *m* using the shared key *k*
- Save the CBC residue *r*
- Transmit *m* and *r* to the remote party
- The remote party recomputes and verifies the CBC residue of *m*

Why does this work?

- Malicious parties can still manipulate *m* in transit
- However, without *k*, they cannot compute the corresponding CBC residue!

The bad news: Encrypting the whole message is expensive!

How can we guarantee the confidentiality and integrity of a message?

Does this mean using CBC encryption gives us confidentiality and integrity at the same time?

Unfortunately, it does not ☹

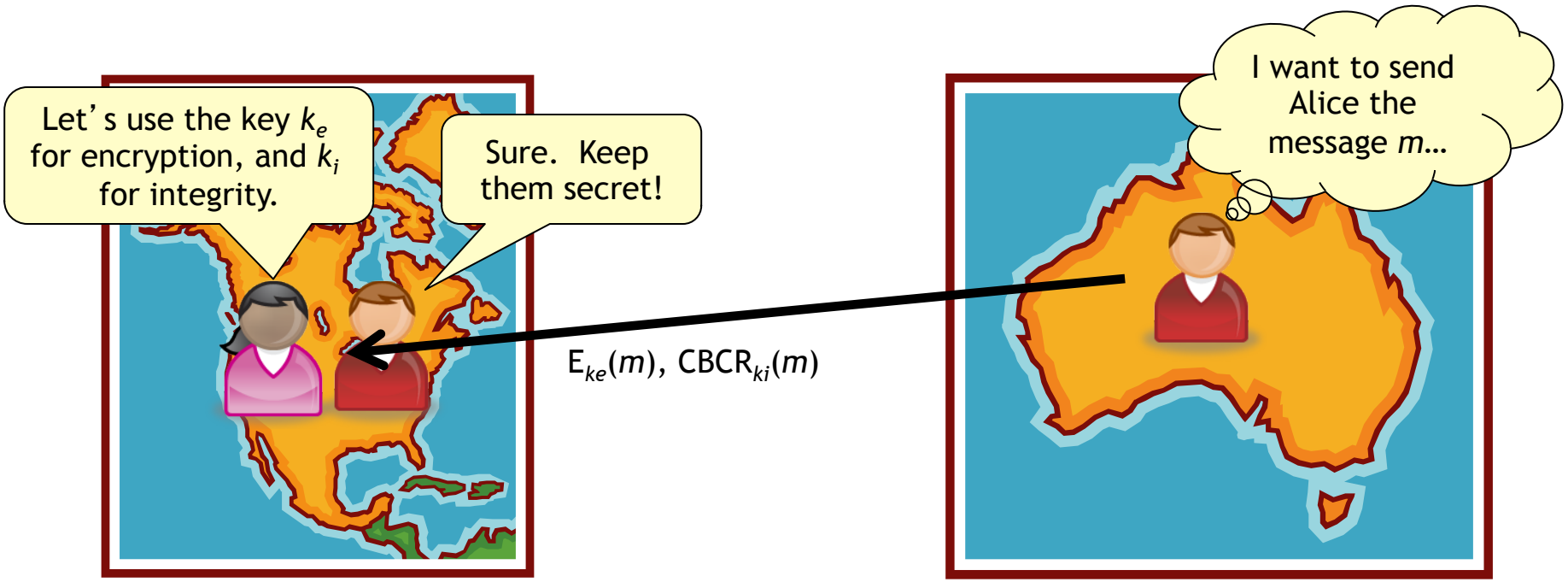
To use CBC for confidentiality and integrity, we need two keys

- Encrypt the message M using k_1 to get ciphertext $C_1 = \{c_{11}, \ldots, c_{1n}\}$
- Encrypt M using k_2 to get $C_2 = \{c_{21}, \ldots, c_{2n}\}$
- Transmit $\langle C_1, c_{2n} \rangle$

But wait, isn't that expensive?

- Fix #1: Exploit parallelism if there is access to multiple cores
- Fix #2: Faster hash-based MACs (next week)

Putting it all together…



All is well?

Today we learned how symmetric-key cryptography can protect the confidentiality and integrity of our communications

So, the security problem is solved, right?

Unfortunately, symmetric key cryptography doesn't solve everything…

1. How do we get secret keys for everyone that we want to talk to?
2. How can we update these keys over time?

Next week: Public key cryptography will help us solve problem 1

Later in the semester, we'll look at key exchange protocols that help with problem 2

Groups!