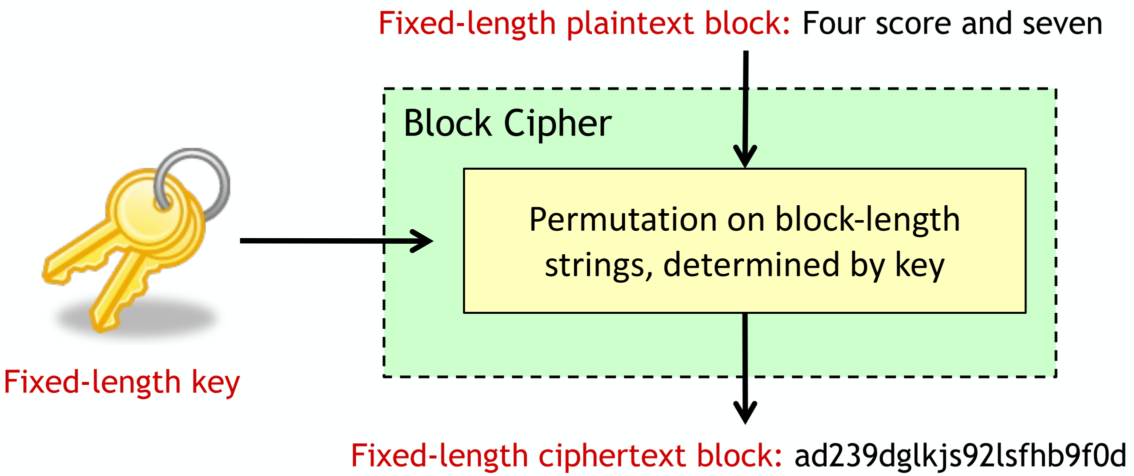
Applied Cryptography and Network Security

William Garrison
bill@cs.pitt.edu
6311 Sennott Square

Block modes of operation and MACs

University of Pittsburgh

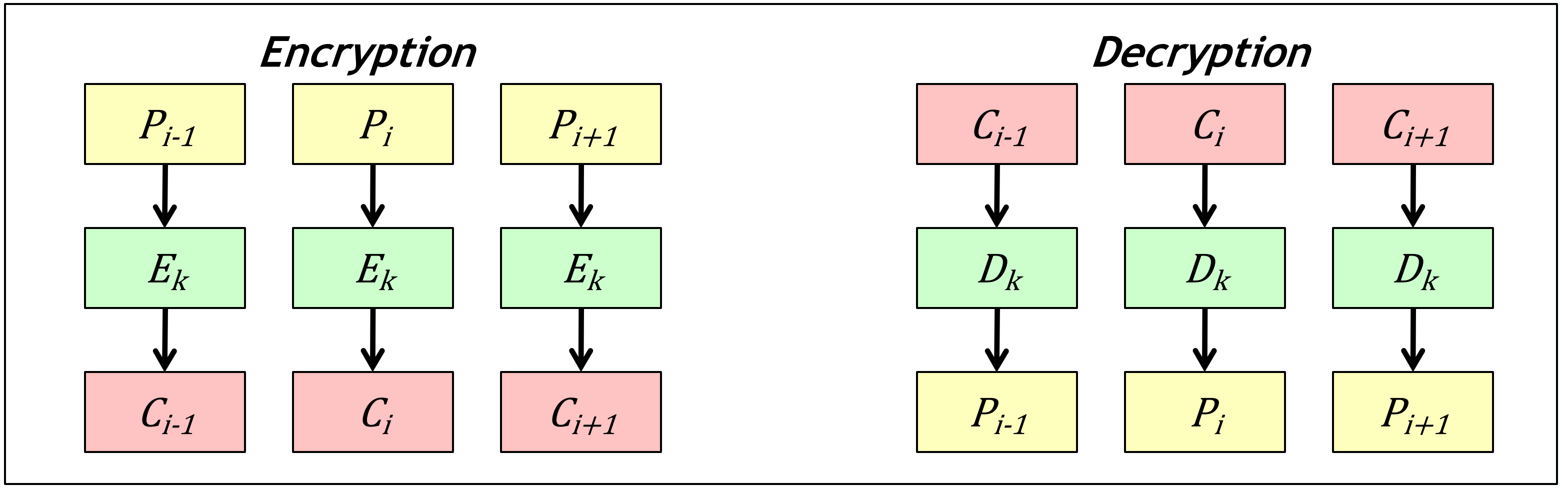
Block Cipher Modes of Operation



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?

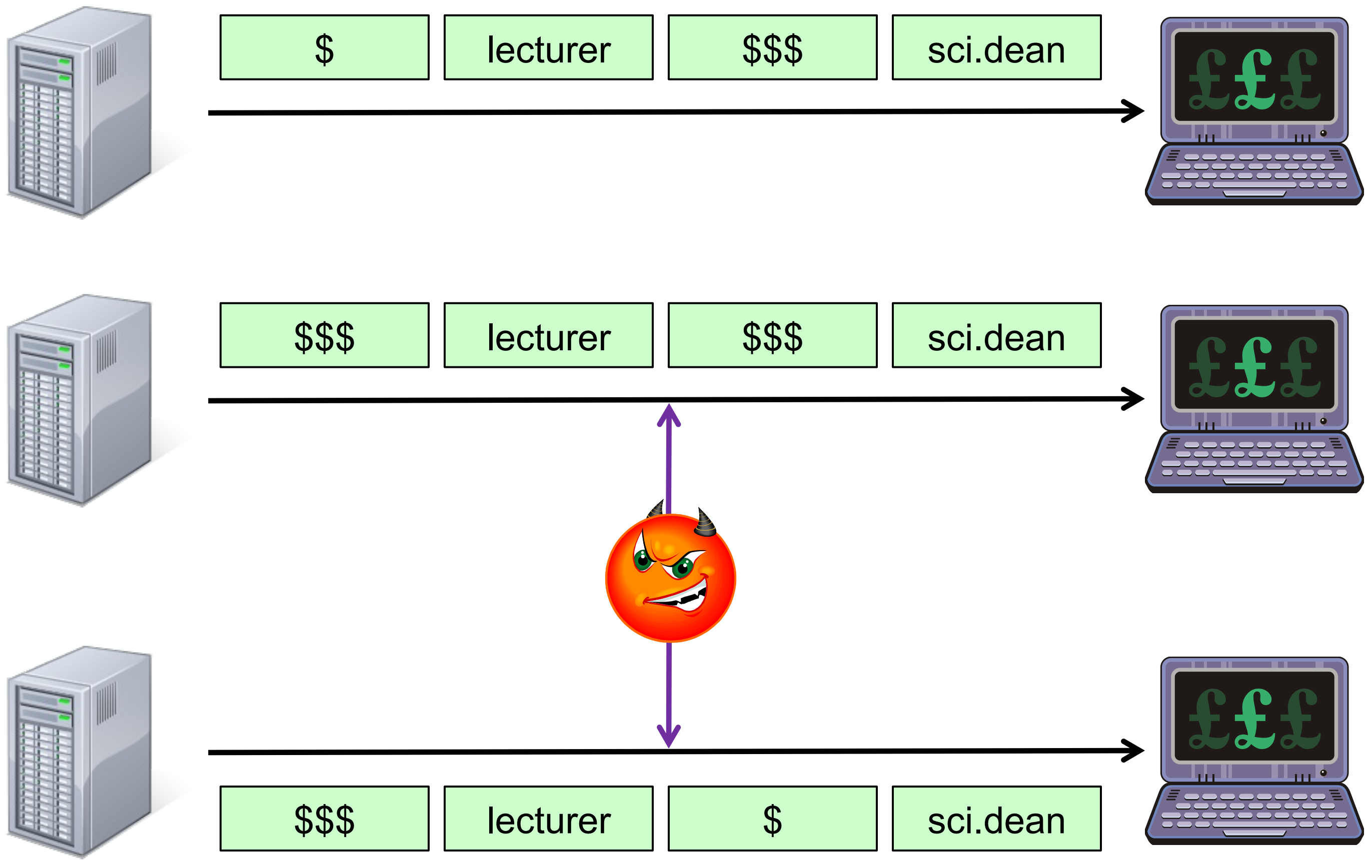
- KPA: Known plaintext/ciphertext pairings
- CPA: Ability to encrypt guesses, and semantic security
- Block replay attacks

This is called a code book

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

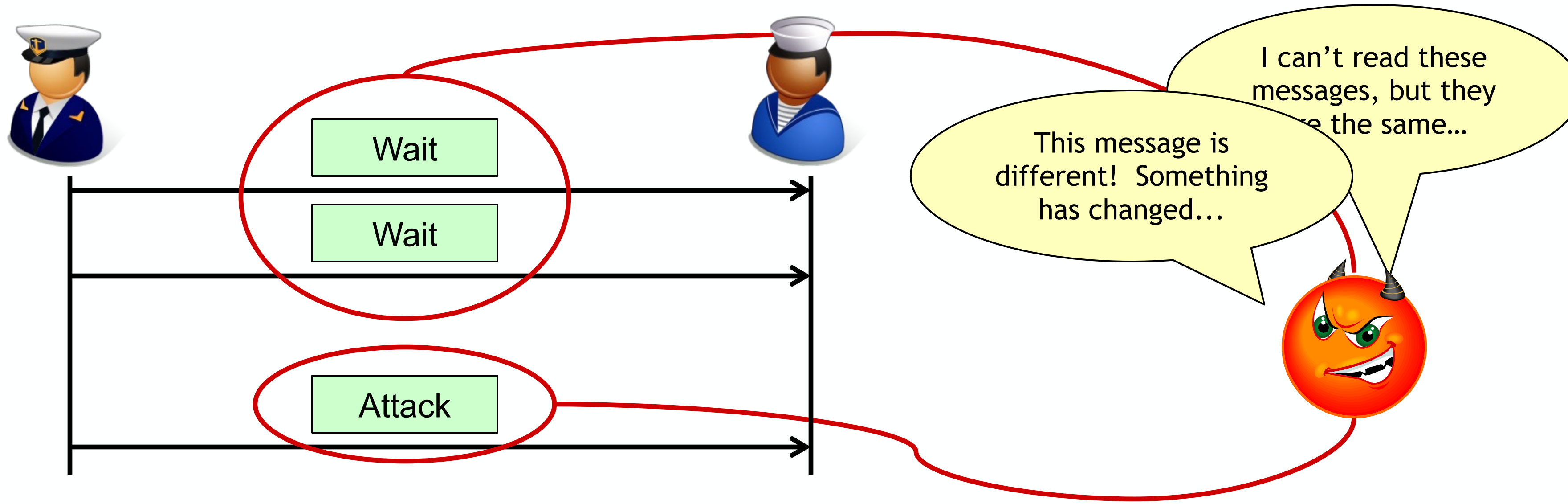
The use of ECB mode can lead to block replay or substitution attacks

Example: Salary data transmitted using ECB



Why is the ability to build a codebook dangerous?

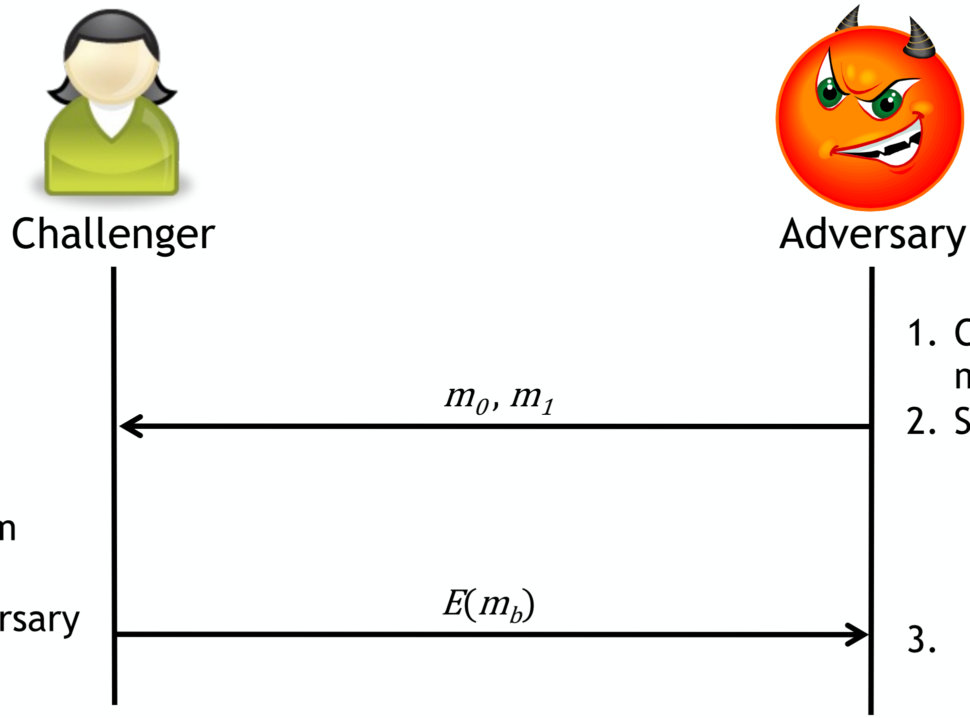
Observation: When using ECB, the same block will always be encrypted the same way



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

Semantic Security / Ciphertext Indistinguishability

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



Adversary:

1. Choose two equal-length messages m_0, m_1
2. Send messages to challenger

Challenger:

1. Choose $b \in \{0,1\}$ uniformly at random
2. Encrypt m_b
3. Send $E(m_b)$ to adversary

Adversary:

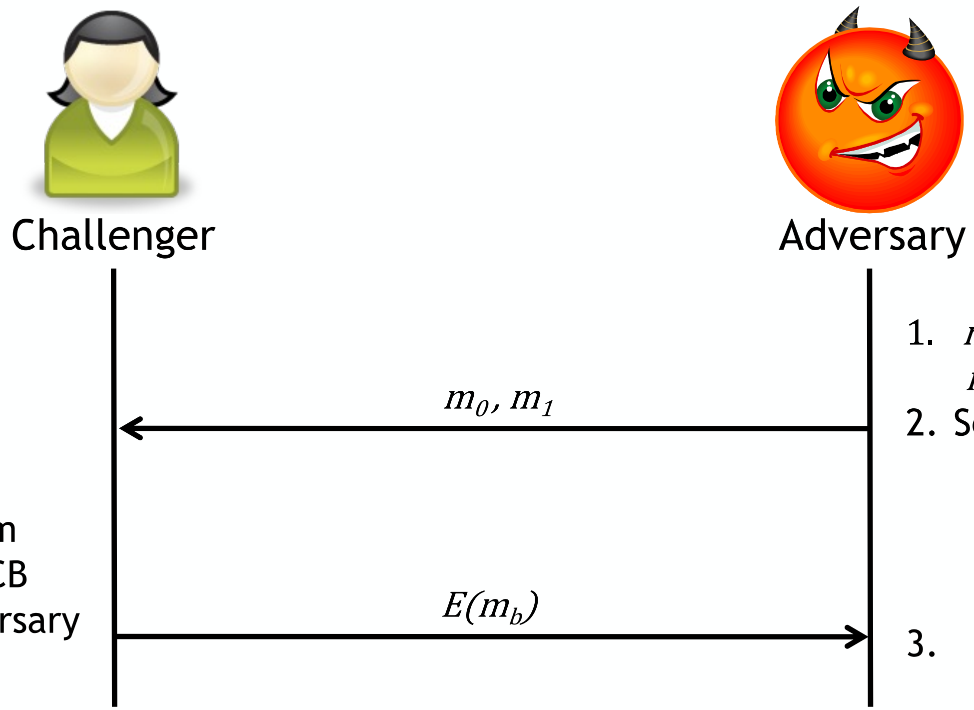
3. Guess b

The adversary "wins" if they have a non-negligible advantage in guessing b. More concretely, they win if $P[b' = b] > ½ + \varepsilon$.

If the adversary does not have an advantage, the cipher is said to provide indistinguishability under chosen-plaintext attack (IND-CPA).

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

Question: Can you demonstrate this?



Adversary:

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

Challenger:

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

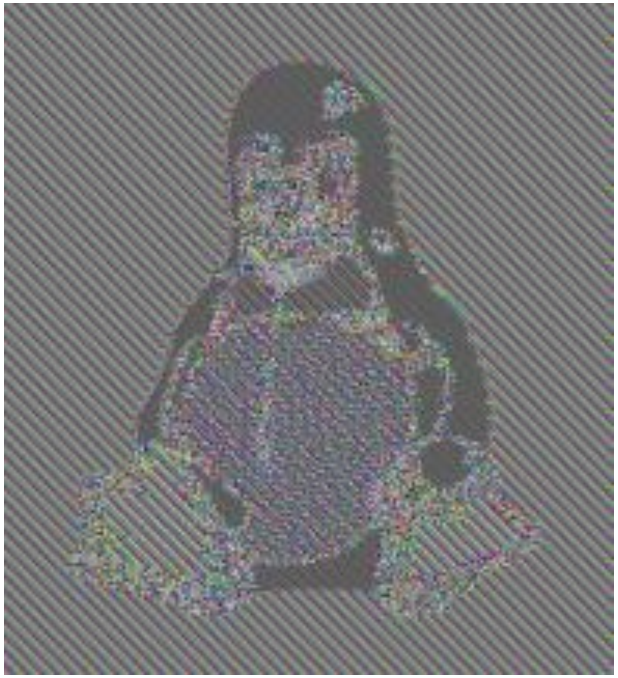
Adversary:

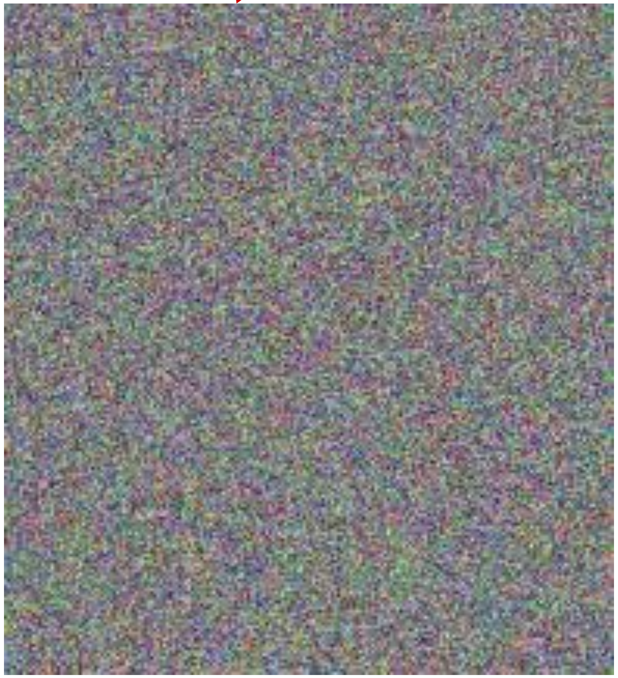
3. If first block = second block, then $b = 0$. Otherwise, $b = 1$.

$P[b' = b] = 1$

Problems with ECB, depicted graphically

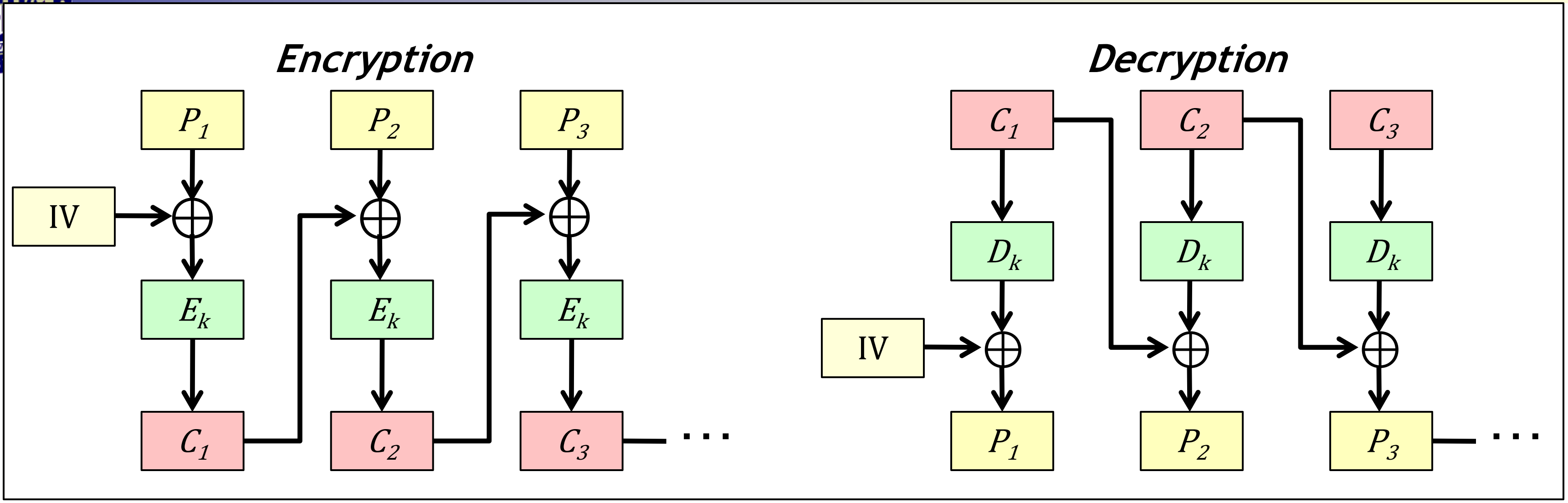
This is what
we want





This is what
ECB does

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

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

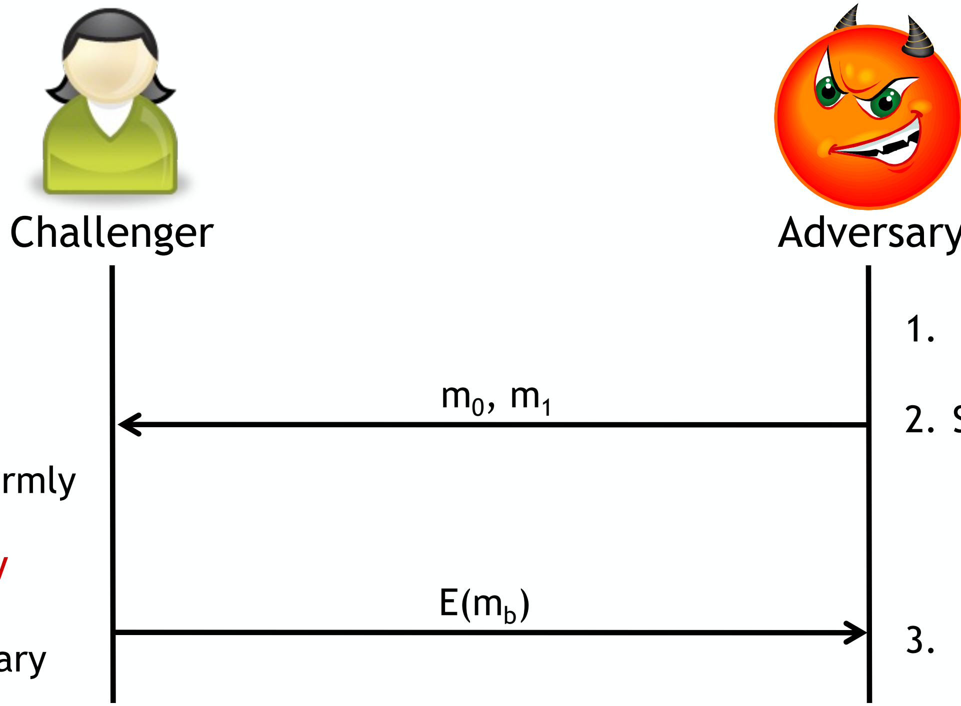
Need to encrypt a random block to get things started

- 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



Adversary:

1. m_0 = "HelloHello"
   m_1 = "HelloWorld"
2. Send messages to challenger
3. If first block = second block, then $b = 0$. Otherwise, $b = 1$.

Challenger:

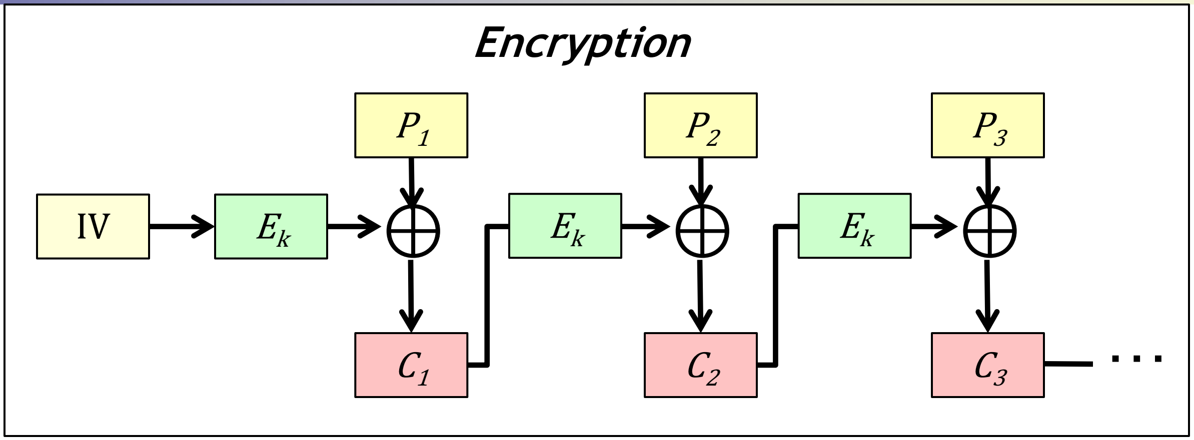
1. Choose $b \in \{0,1\}$ uniformly at random
2. Generate a random IV
3. Encrypt m_b using CBC
4. Send $E(m_b)$ to adversary

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

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

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:

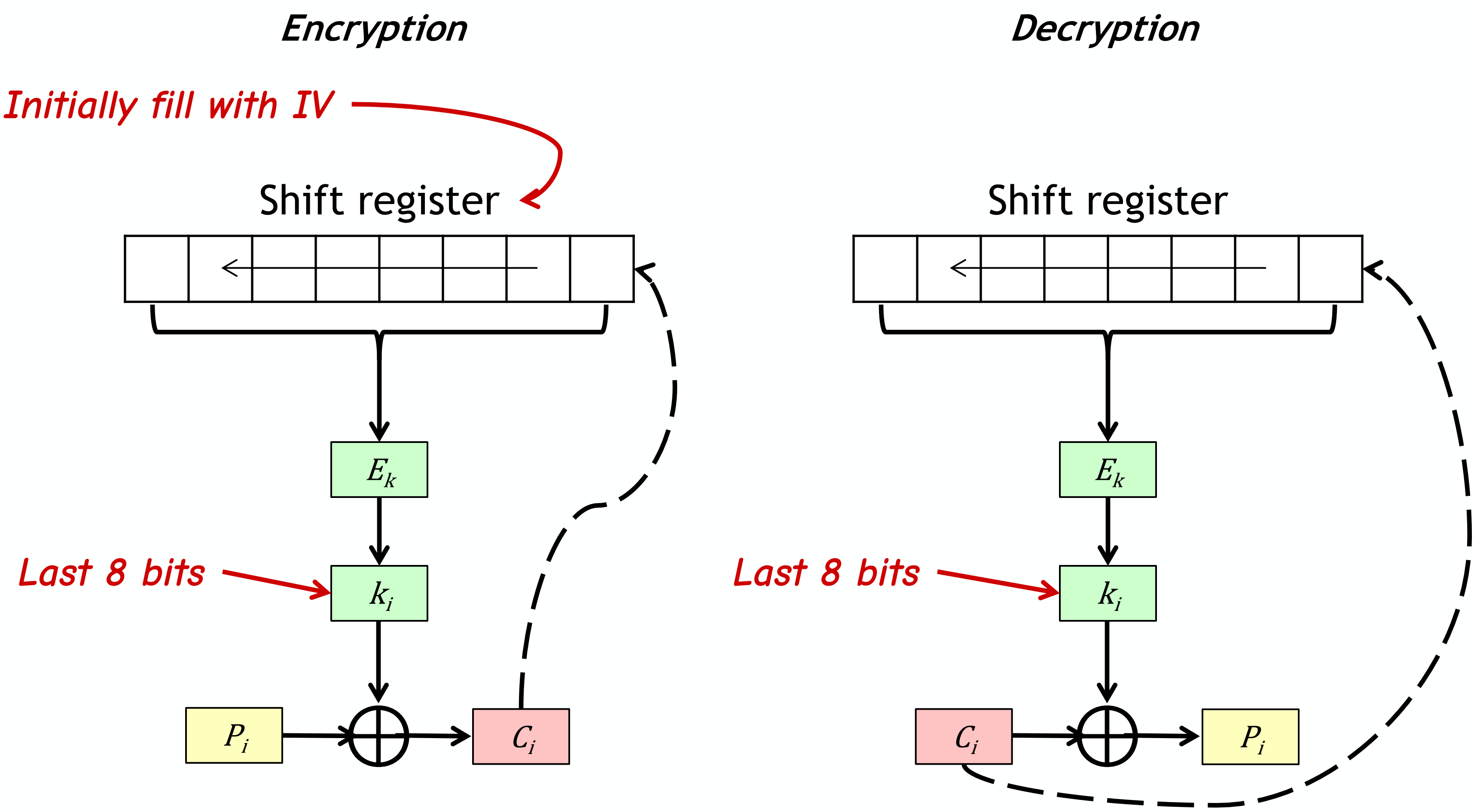
- $C_i = P_i \oplus \mathrm{E}_k(C_{i-1})$
- $P_i = C_i \oplus \mathrm{E}_k(C_{i-1})$

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

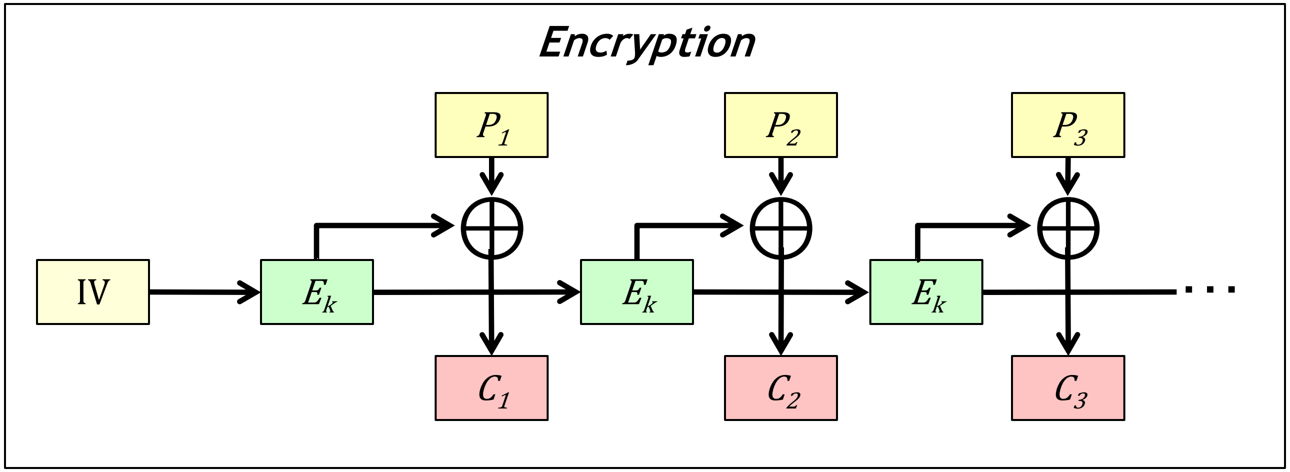
This is great, since we don't need to wait for n bits of plaintext to encrypt!

- ***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



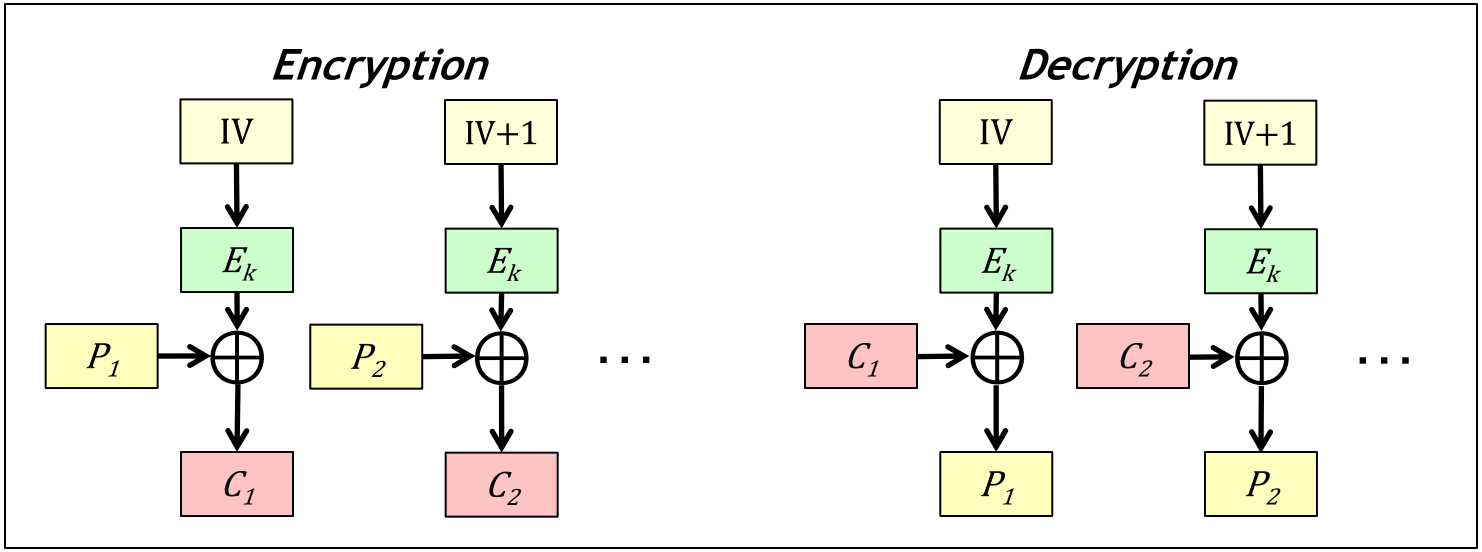
How does this work?

- $C_i = P_i \oplus S_i, \;\; S_i = \mathrm{E}_k(S_{i-1})$
- $P_i = C_i \oplus S_i, \;\; S_i = \mathrm{E}_k(S_{i-1})$

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

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
  - $c_k = p_k \oplus \mathrm{E}(\mathrm{IV} + k)$
- 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…

A couple other block modes worth mentioning…

XEX (XOR, Encrypt, XOR)

- Designed for full disk encryption, where we need to read/write any block quickly
- Keystream depends on the location on the disk
- Prevents targeted modifications when plaintext is known
  - (See §4 HW #6 regarding how this can be done in CTR)

XTS (XEX with Ciphertext Stealing)

- XEX is inefficient if the cipher block size doesn't go in evenly into the disk block size
  - Wasted partial disk block
- Ciphertext stealing is a trick for altering the final two cipher blocks to fit a smaller space

So… Which mode of operation should I use?

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

- And even then, be very cautious. It's best to switch.

Use CBC if either:

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

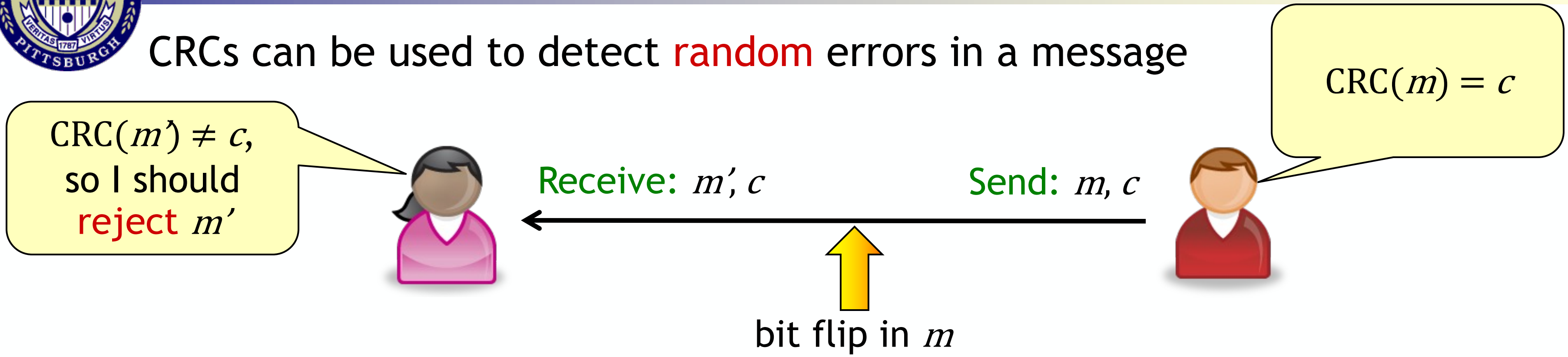
CFB (usually 8-bit CFB) is the best choice for encrypting streams of characters entered at, e.g., a text terminal

XTS is standard for full-disk encryption

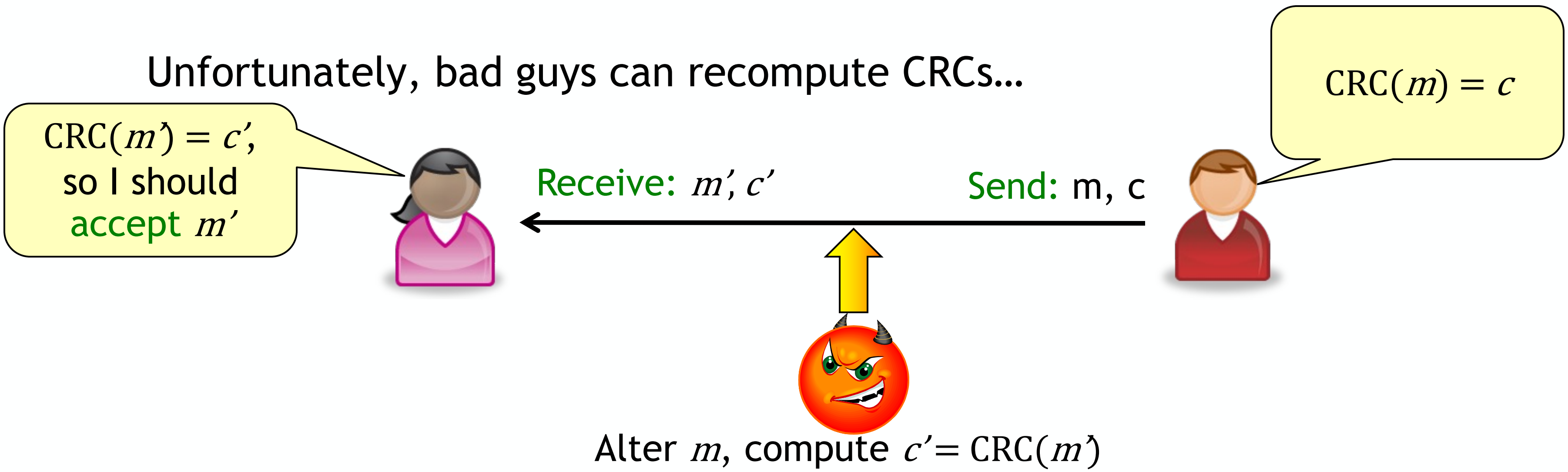
Stay up to date with modern best practices!

Encryption does not guarantee integrity/authenticity

CRCs can be used to detect random errors in a message

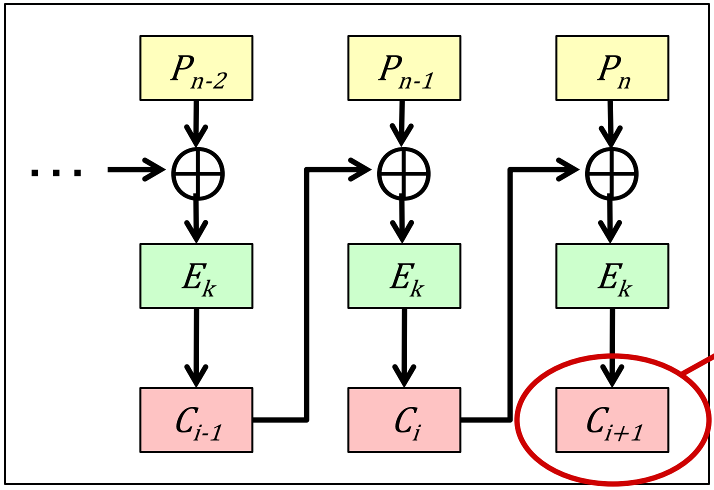


Unfortunately, bad guys can recompute CRCs…



Solution: Cryptographic message authentication codes (MACs)

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



The last block of a CBC encryption is called the CBC residue

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 😞

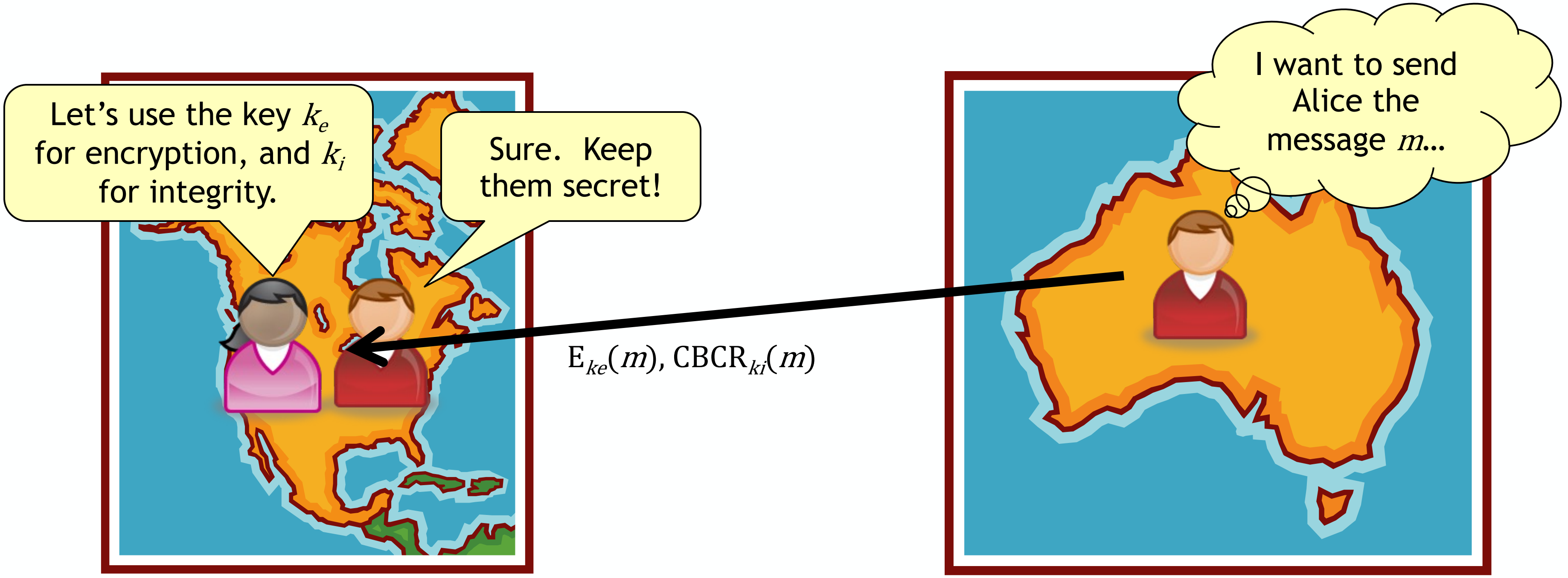
To use CBC for confidentiality and integrity, we need two keys

- 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 twice the runtime?

- Some block modes combine confidentiality and integrity
    - e.g., CCM, GCM, but are similarly slow; see §4.4
- Fix #1: Exploit parallelism if there is access to multiple cores
- Fix #2: Faster hash-based MACs (next up!)

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?

Coming up soon: 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

Next: Hashing and more efficient MACs