CS 177
Computer Security
Lecture 3

Stefano Tessaro

tessaro@cs.ucsb.edu

Today – More symmetric cryptography

- Definitions of confidentiality
- Modes of operations for block ciphers
- Introduction to encryption integrity
- Message-authentication codes

Solution: Block cipher

Informal definition. A **block cipher** is a substitution cipher where the plaintext is made of blocks from a very large alphabet, but with a very compact key.

Formally, a special case of encryption algorithm:

- Kg outputs a random key of length k bits
  Typically $k = 128$ or $k = 256$
- Enc is such that for all keys K, $\mathrm{Enc}(K, .)$ is a one-to-one function $\{0,1\}^n \rightarrow \{0,1\}^n$ [The substitution table]
  n = **block length**; usually $n = 128$

That is, the blocks are n-bit strings

The magic of block ciphers

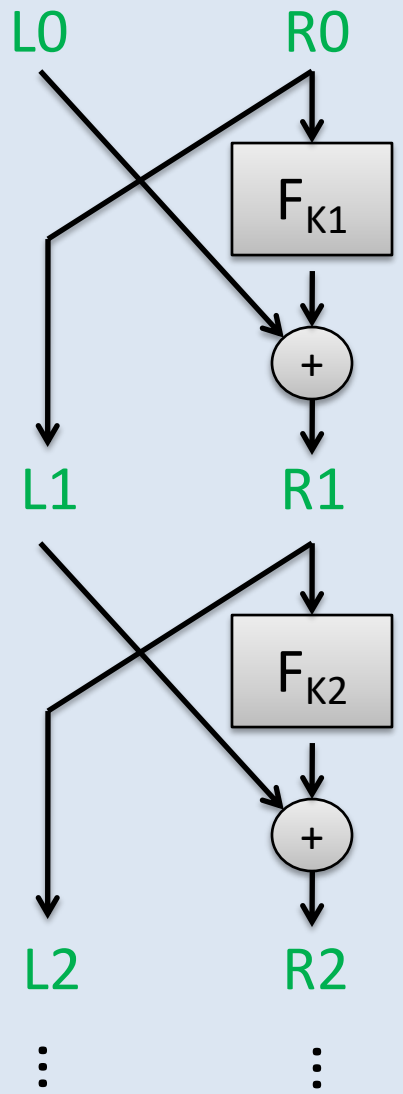
- There are $2^{128}!$ permutations over 128-bit strings.
  If the key described the table for randomly chosen such permutation, the key would be roughly 128×2^{128} bits long
- A block cipher is "as good as" choosing such a randomly chosen permutation, but only needs a short key, e.g., 128 bits
- How do we build them?
- Two examples next: **DES** and **AES**

Data encryption standard (DES)

Originally called Lucifer

- Team at IBM, led by Horst Feistel
- Input from NSA
- Standardized by NIST in 1976

$$n = 64$$
$$k = 56$$



Split 64-bit input into L0,R0 of 32 bits each

Repeat round 16 times

Each round applies function F using separate round keys K1 … K16 derived from main key.

DES is essentially broken

Attack	Attack type	Complexity	Year
Biham, Shamir	Chosen plaintexts, recovers key	2^{47} plaintext, ciphertext pairs	1992
DESCHALL	Unknown plaintext, recovers key	$2^{56/4}$ DES computations 41 days	1997
EFF Deepcrack	Unknown plaintext, recovers key	~4.5 days	1998
Deepcrack + DESCHALL	Unknown plaintext, recovers key	22 hours	1999

3DES (use DES 3 times in a row with more keys) expands keyspace to 118 bits and still found in practice
(E.g., PIN-based card transactions)

Bottom line: While 3DES may still be fine, just stay away from it if you can!

Advanced Encryption Standard (AES)

Response to 1990s attacks:

- NIST has design competition for new block cipher standard
- 5 year design competition
- 15 designs, Rijndael design chosen
- AES is very fast (AES-NI native instruction on modern chips)

Advanced Encryption Standard (AES)

Rijndael (Rijmen and Daemen)

$$n = 128$$
$$k = 128, 192, 256$$

For $k = 128$ uses 10 rounds of:

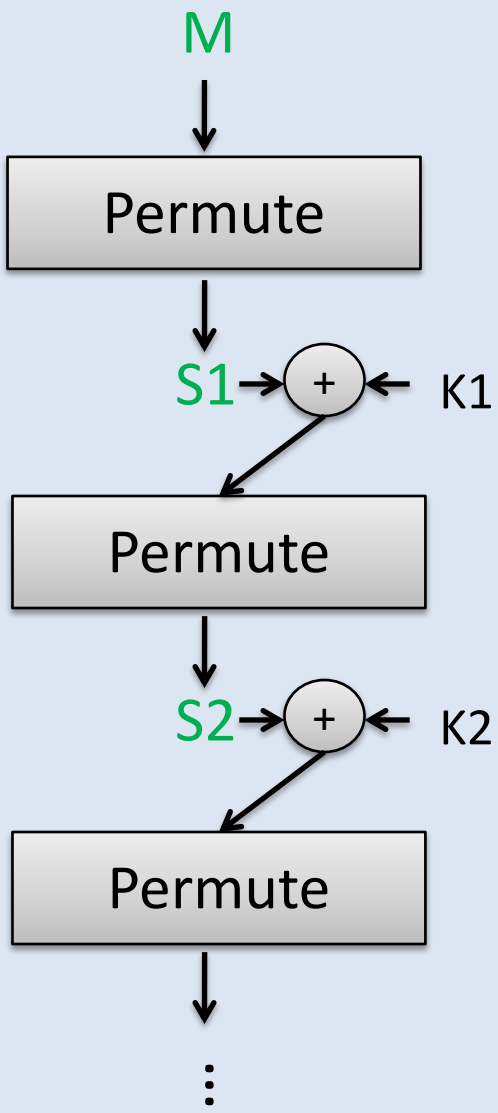
1) Permute:

SubBytes (non-linear S-boxes)

ShiftRows + MixCols (invertible linear transform)

2) XOR in a round key derived from K

(Actually last round skips MixCols)



Best attacks against AES

Attack	Attack type	Complexity	Year
Bogdanov, Khovratovich, Rechberger	key recovery	$2^{126.1}$ time + some data overheads	2011

- Brute force requires time 2^{128}
- Approximately factor 4 speedup

Bottom line: AES is very secure, and there has been surprisingly little progress on breaking it!

Pro-tip: If someone wants to use any different than AES, insist to know why and ask a cryptographer you trust. (Never trust home-brewed crypto.)

Exception: Lightweight applications (IoT, memory encryption) often require simpler ciphers, but no clear standard

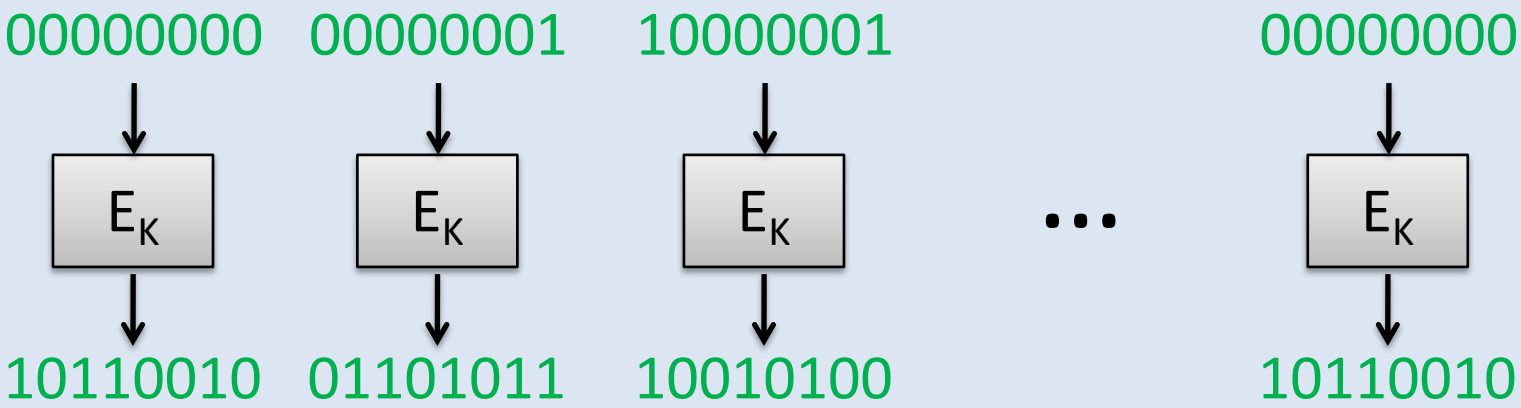
Block Cipher Security Goal

Pseudorandom permutation (PRP)
Informally: A block cipher (e.g., AES) with block length n under a random secret key behaves as an ideal monoalphabetic substitution cipher with n-bit alphabet

- Tricky to formalize mathematically: See CS178
- This can hold only for adversaries which have limited (yet large) adversarial resources, e.g., they are not able to recover the key (otherwise they will be able to tell)
- But as a system designer, you have to think of block ciphers as providing this functionality

Pseudorandom Permutations

Key K: Secret and random!



Important property: as long as the key is unknown and randomly chosen, outputs on different inputs look like **random and independent strings**

Example – Play with AES

128 bits = 16 bytes

secret key 1 = f6 cc 34 cd c5 55 c5 41 82 54 26 02 03 ad 3e cd

secret key 2 = 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c

Plaintext	Ciphertext (key 1)	Ciphertext (key 2)
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	5b c0 68 39 7b 4f 93 c4 ce d1 6a 79 94 1a 48 30	7d f7 6b 0c 1a b8 99 b3 3e 42 f0 47 b9 1b 54 6f
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01	33 93 ed 94 85 c8 90 a7 d0 33 9a 78 c0 63 33 d2	57 12 7d 40 34 b1 be bf ae f4 66 b9 c7 72 6f c6
33 93 ed 94 85 c8 90 a7 d0 33 9a 78 c0 63 33 d2	56 dc a7 27 4d eb d5 cd 71 9c 7e a5 7a 54 67 4d	8c f0 60 ca 67 67 8b 0a c6 64 9d 8f ae 76 d1 f8

How do we encrypt > 16 bytes?

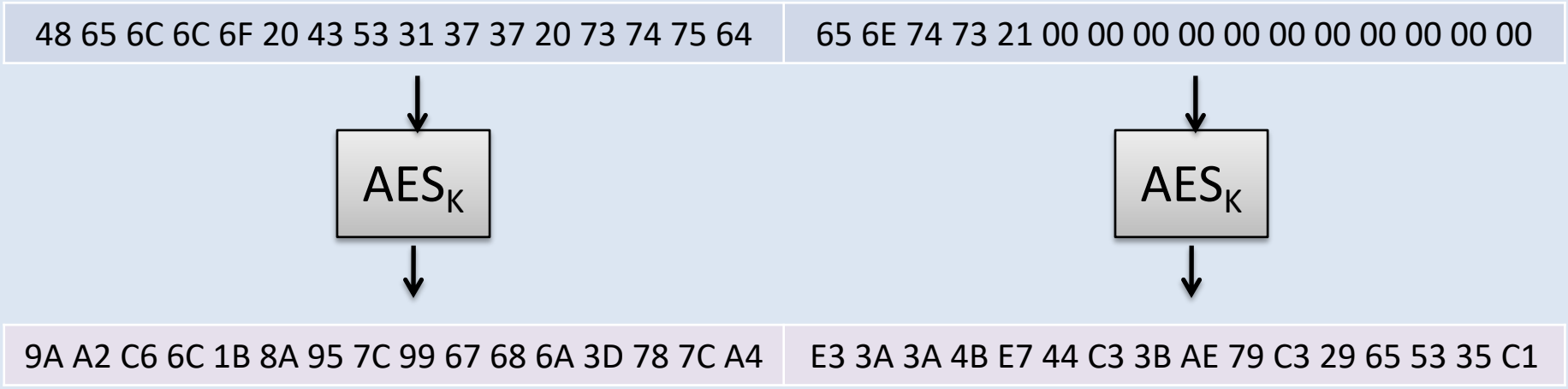
Long messages – Electronic Codebook (ECB)

Message M = "Hello CS177 students!"
We use AES, $n = 128 = 16$ bytes

	$M[1]$	$M[2]$
$M =$	48 65 6C 6C 6F 20 43 53 31 37 37 20 73 74 75 64	65 6E 74 73 21 00

$K =$ 0F 15 71 C9 47 D9 E8 59 0C B7 AD D6 AF 7F 67 98

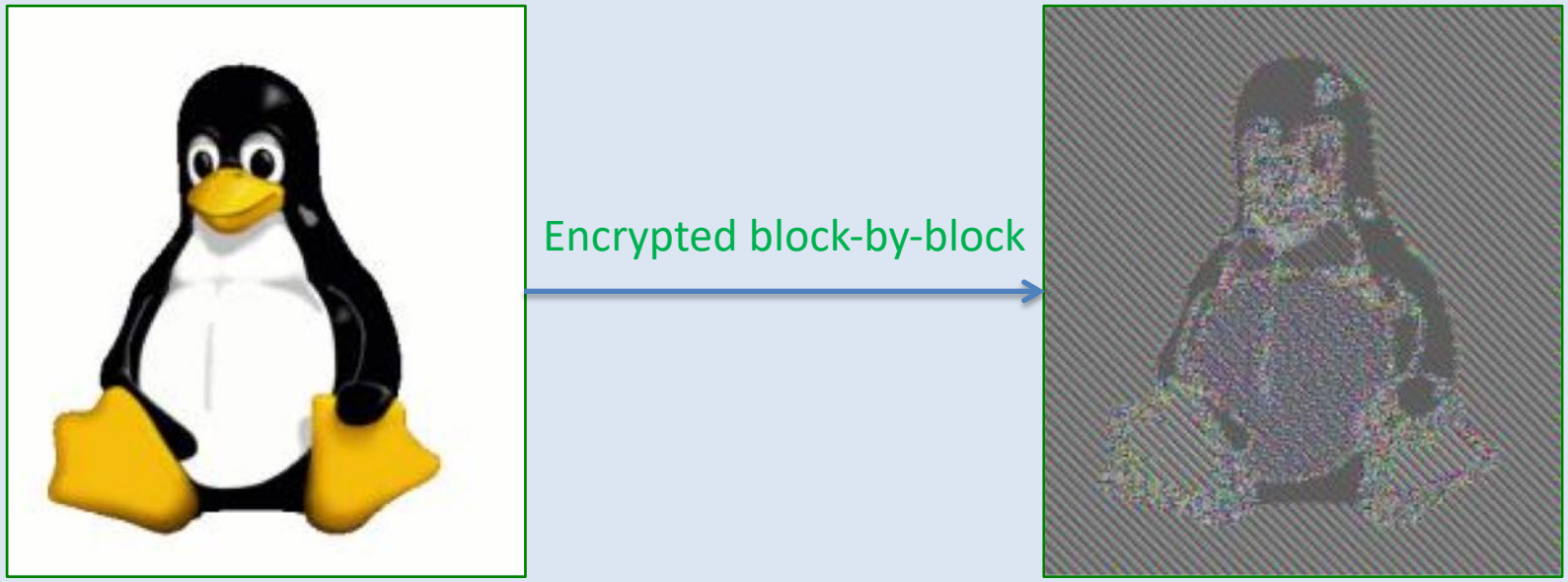
To encrypt M using the key K:



Is this a good idea?!

Drawbacks of block ciphers

The same 16-byte sequence is always encrypted in the same way.



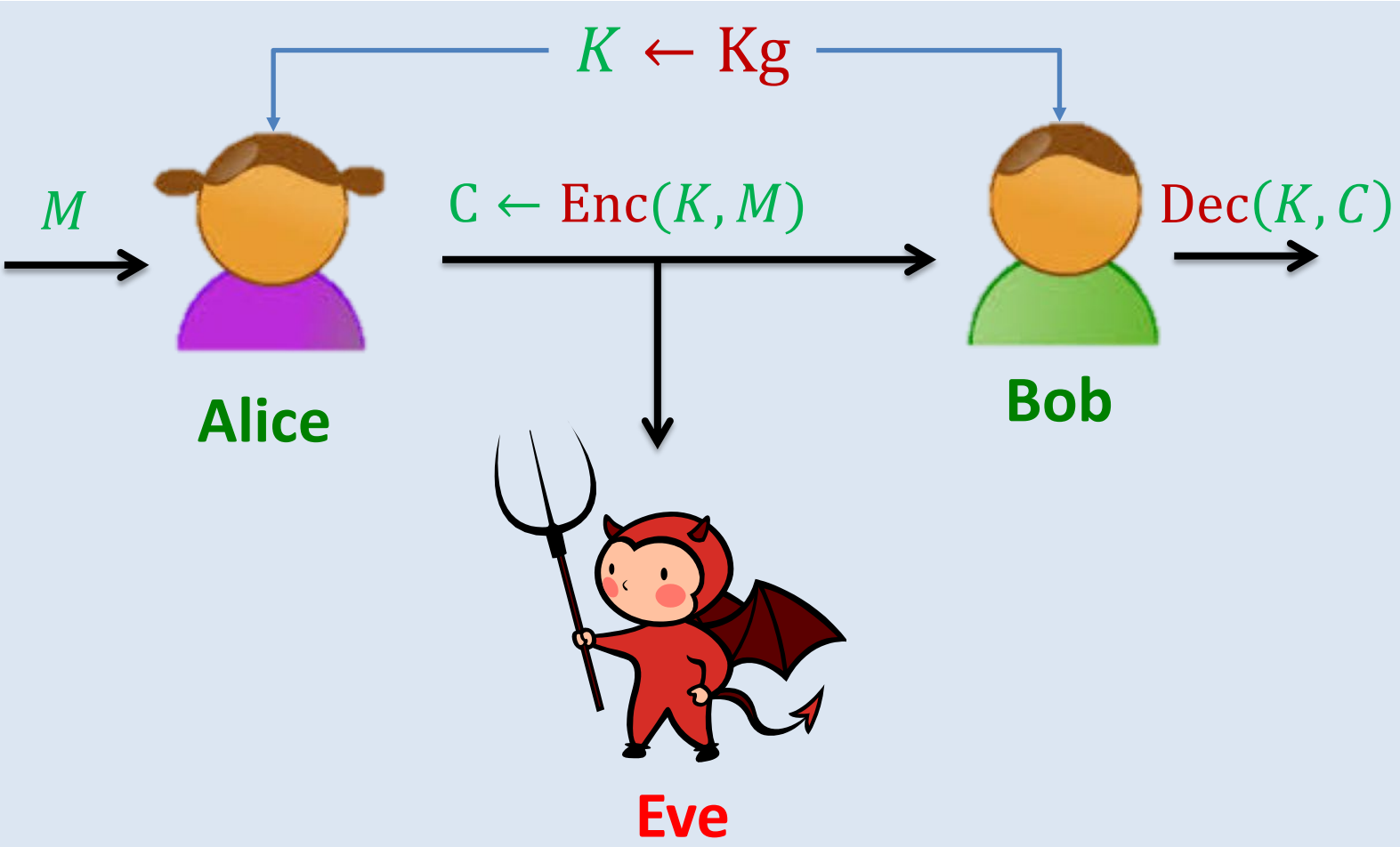
http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation

So why bother? Great building block for better approaches! (next)

Summary – Take home

- Block ciphers: How to recognize one!
- Examples: DES (insecure) and AES (secure)
- Security as a pseudorandom permutation
- ECB and its insecurity

Symmetric Encryption – Confidentiality



Modes of operation

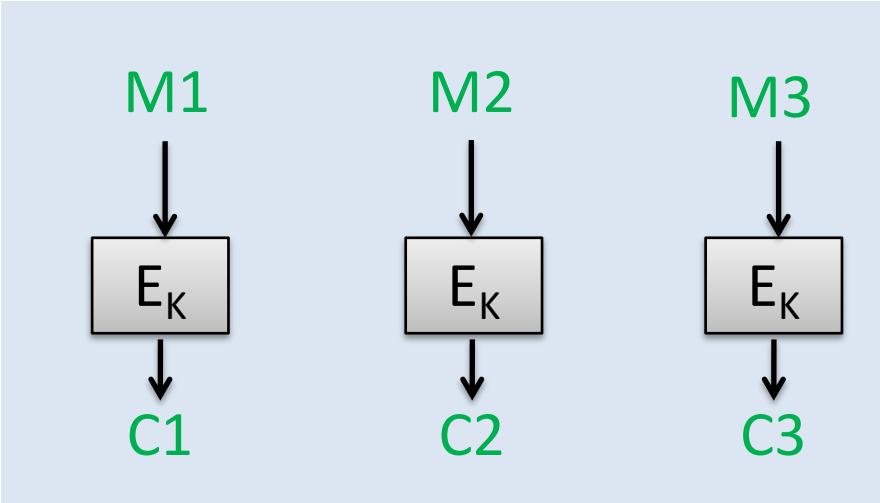
Common design paradigm in cryptography:
Design **modes of operation**

A mode of operation is an algorithm which uses a block cipher (assumed to be secure, typically in the sense of being a good PRP) to solve a more complex task.

(Bad) Example: Electronic Codebook (ECB)

Here, E is a block cipher with key length k and block length n

Algorithm $\mathrm{Enc}(K, M)$:
Split M into blocks $M[1], \ldots, M[r]$
// all blocks are n-bits
Pick random $\mathrm{IV} \in \{0,1\}^n$
for $i = 1, \ldots, r$ **do**
    $C[i] \leftarrow E_K(M[i])$
return $C[1], \ldots, C[r]$

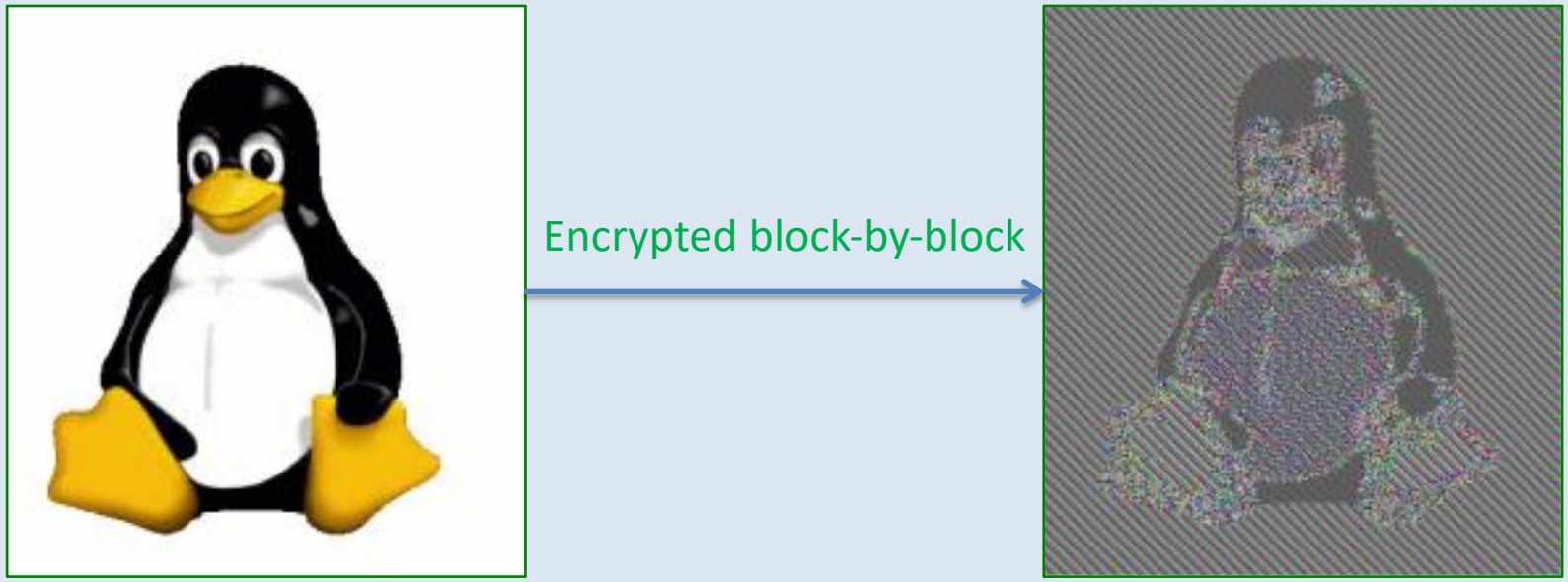


Anything missing?

What is the decryption algorithm? What is the key-generation algorithm?

ECB Insecurity

Two identical 16-byte sequences are still going to be encrypted in the same way. **Is this ever a problem?**



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

What does it mean to be secure? What should have we seen instead?

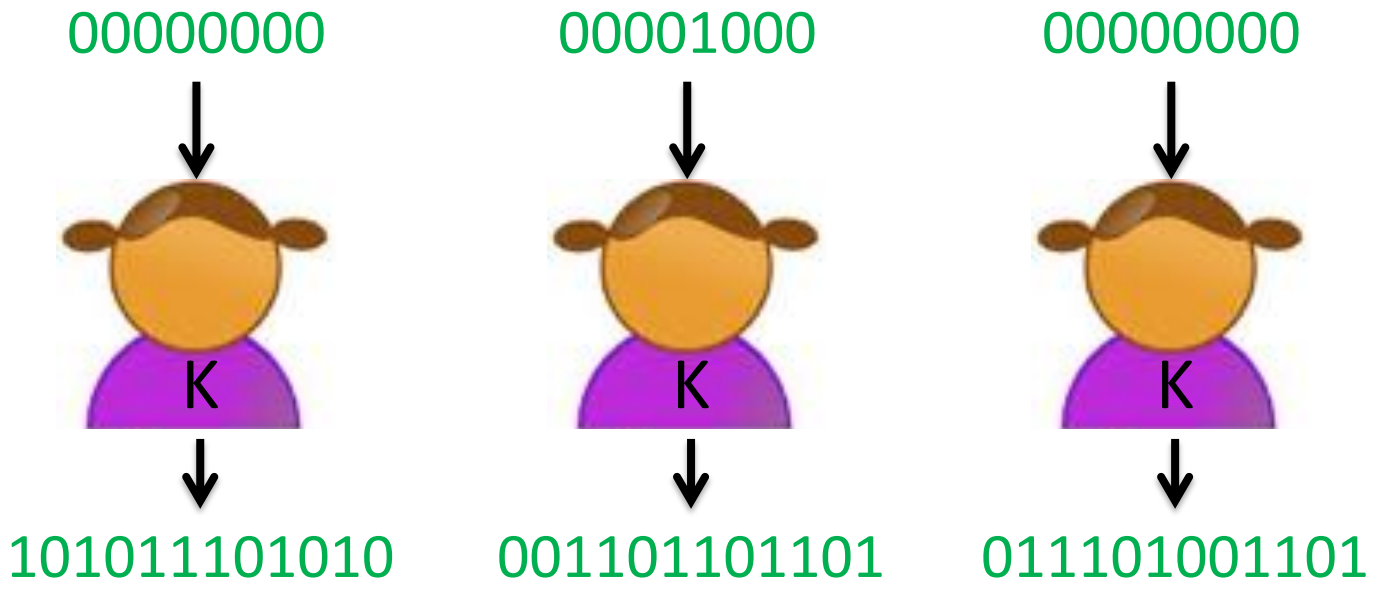
Right definition: Semantic Security
[Goldwasser-Micali, 1982]

Turing Award (2013)

“Right” definition of message confidentiality

Semantic Security
[Goldwasser-Micali, 1982]

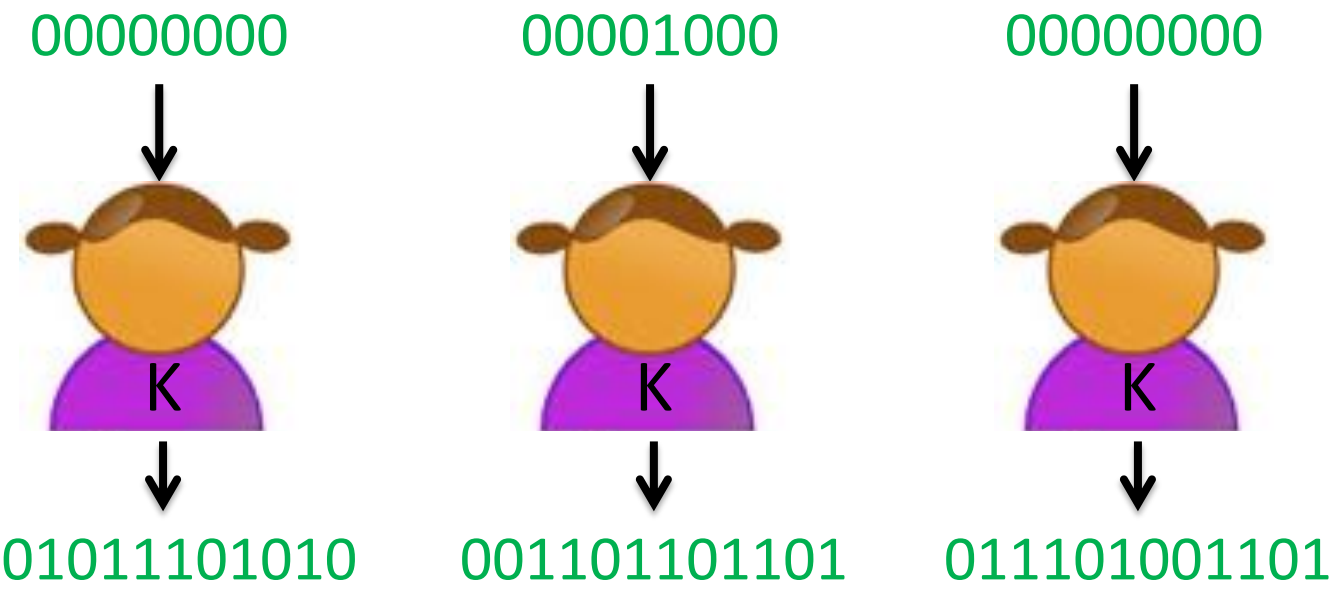
Encryption reveals **nothing about the message**, not even whether the same message was encrypted earlier!



Every time we encrypt (regardless of new or old message), ciphertext "looks random" to a computationally bounded adversary.

Semantic Security

[Goldwasser-Micali, 1982]



To ensure this, encryption must use **randomization**
(many possible ciphertexts for same message)

Ciphertext must be longer than plaintext!

Next – Semantically Secure Encryption

How do we achieve it from block ciphers?

A basic fact – Bitwise XOR

Bitwise XOR $\oplus$ of two strings $X, Y \in \{0,1\}^n$

Example, $n = 8$

X	0	1	0	0	1	0	1	1
Y	1	0	1	0	1	0	0	1
$X \oplus Y$	**1**	**1**	**1**	**0**	**0**	**0**	**1**	**0**

XOR Magic - Masking

Fix a string $X \in \{0,1\}^n$

Imagine we pick $K \in \{0,1\}^n$ uniformly at random

That is each of the 2^n possible strings is equally likely

Question: How does $X \oplus K$ look like?

Answer: $X \oplus K$ can become every string Y with probability $1/2^n$

XOR Magic - Masking

Example. $X = 01001011$

What is the probability that $X \oplus K = 11110000$?

X	0	1	0	0	1	0	1	1
K	**1**	**0**	**1**	**1**	**1**	**0**	**1**	**1**
$X \oplus K$	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**

This is exactly the probability that $K = 10111011$, which is exactly $1/2^n$

But the same is true for every X and every choice of $X \oplus K$ …

The learnt lesson -- we will need this soon

Masking: "If we bitwise-xor a random string K to any string X, the outcome $C = X \oplus K$ is random and independent of the original X, and thus hides everything about X."

We will use this!!!!

A little bit of notation

For string $X \in \{0,1\}^n$ and natural number $a \in \mathbb{N}$, define $X + a$ as the n-bit string obtained by:

1. Interpreting X as the binary encoding of an integer b_X in $\{0,1, \ldots, 2^n - 1\}$
2. Compute the binary encoding Y of $b_X + a$
3. Let $X + a$ be the n least significant bits of Y

Examples: $n = 4$

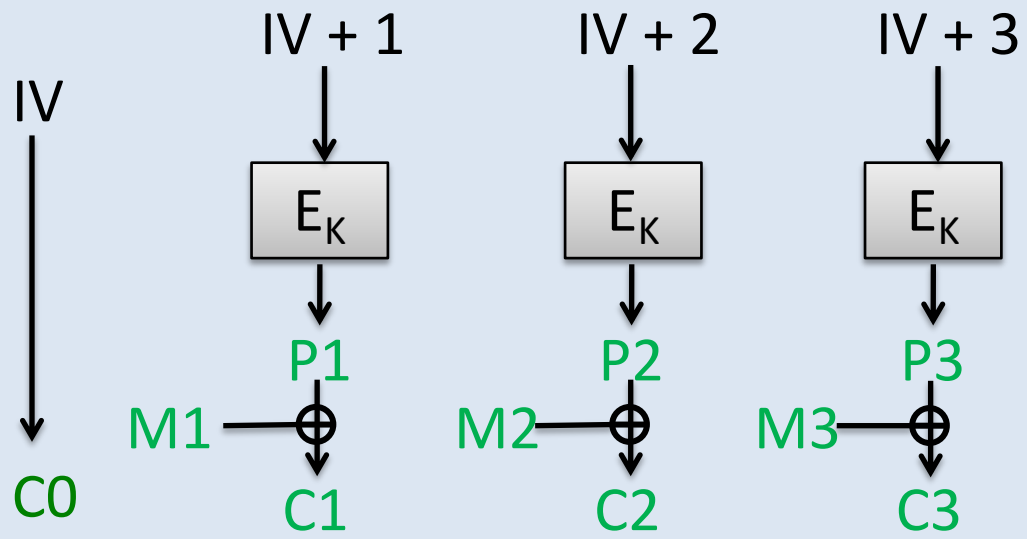
$0000 + 1 = 0001$	$b_X = 0$	$b_X + a = 1$	$Y = 0001$
$0010 + 5 = 0111$	$b_X = 2$	$b_X + a = 7$	$Y = 0111$
$1111 + 2 = 0001$	$b_X = 15$	$b_X + a = 17$	$Y = 10001$

Counter Mode Encryption (CTR)

Algorithm $\mathrm{Enc}(K, M)$:
Split M into blocks $M[1], \ldots, M[r]$
// all blocks except possibly $M[r]$ are n-bits
Pick random $\mathrm{IV} \in \{0,1\}^n$
$C[0] \leftarrow \mathrm{IV}$
for $i = 1, \ldots, r$ **do**
    $P[i] \leftarrow E_K(\mathrm{IV} + i)$
    $C[i] \leftarrow M[i] \oplus P[i]$
return $C[0], C[1], \ldots, C[r]$

Note: If $M[r]$ shorter than n bits, then also shorten $P[r]$ as necessary

Think: How do we decrypt?



CTR – Example

Message M = "Hello CS177 students!"
We use CTR mode with AES, $n = 128 = 16$ bytes

	$M[1]$	$M[2]$
$M =$	48 65 6C 6C 6F 20 43 53 31 37 37 20 73 74 75 64	65 6E 74 73 21 00

$K =$ 0F 15 71 C9 47 D9 E8 59 0C B7 AD D6 AF 7F 67 98

To encrypt M using the key K:

We have chosen at random IV = CC 32 FA B3 E9 12 47 81 FF 1B 3C D6 AA 98 42 03

$\text{IV} + 1 =$ CC 32 FA B3 E9 12 47 81 FF 1B 3C D6 AA 98 42 04

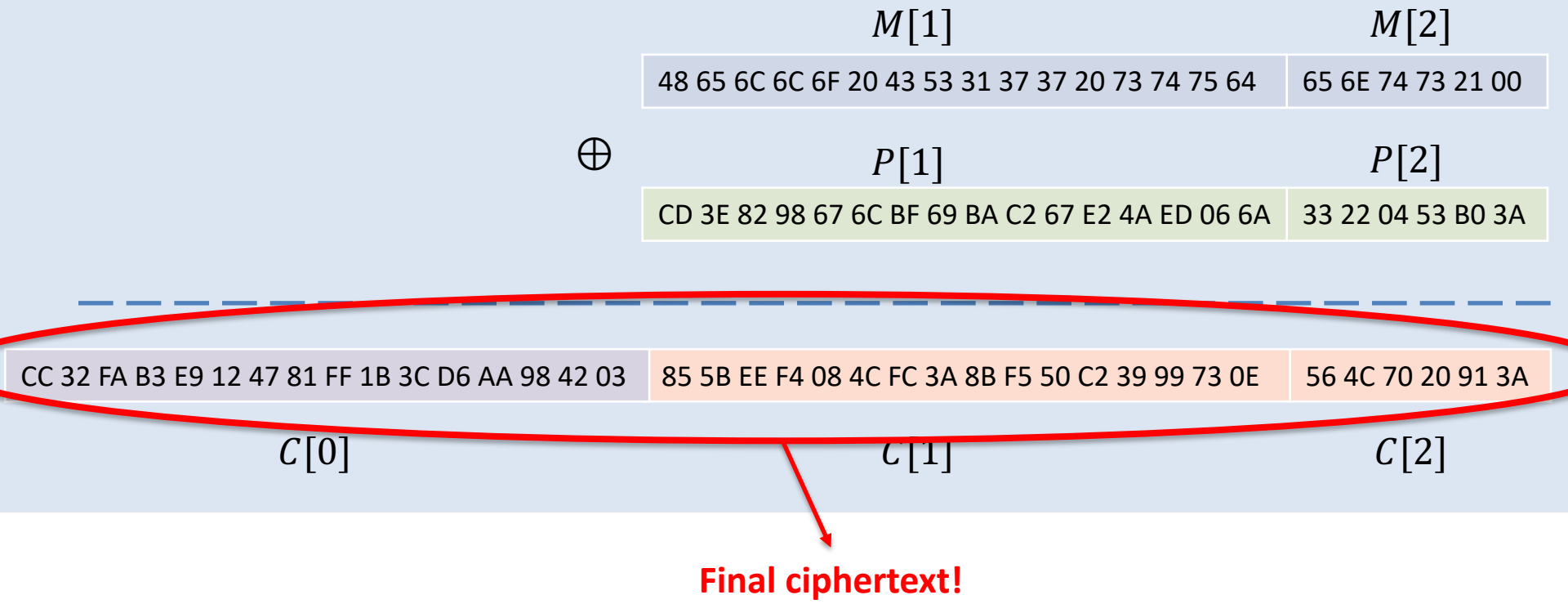
$\text{IV} + 2 =$ CC 32 FA B3 E9 12 47 81 FF 1B 3C D6 AA 98 42 05

$P[1] = AES_K(\text{IV} + 1) =$ CD 3E 82 98 67 6C BF 69 BA C2 67 E2 4A ED 06 6A

$P[2] = AES_K(\text{IV} + 2) =$ 33 22 04 53 B0 3A 1D DA 84 A9 40 A8 52 75 0B F7

CTR – Example (cont'd)

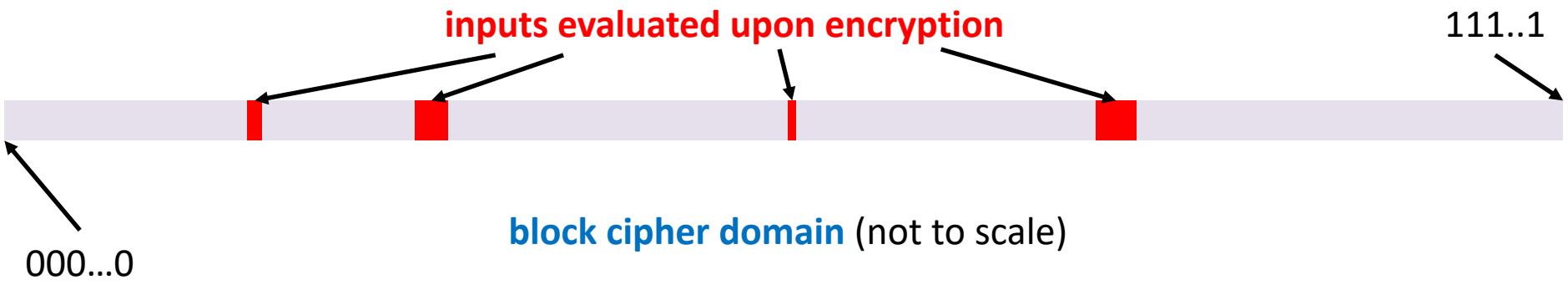
Need to compute final ciphertext



Clearly, re-encrypting it again will result in a (very) different IV, and thus in a completely unrelated mask, since AES is evaluated (with high probability) **at completely different locations** IV, IV + 1, IV + 2, …

CTR – Why is it secure?

Masking strings $P[1], \ldots, P[r]$ generated upon each encryption will come from disjoint parts of the block cipher domain (with high probability), and thus look random and independent



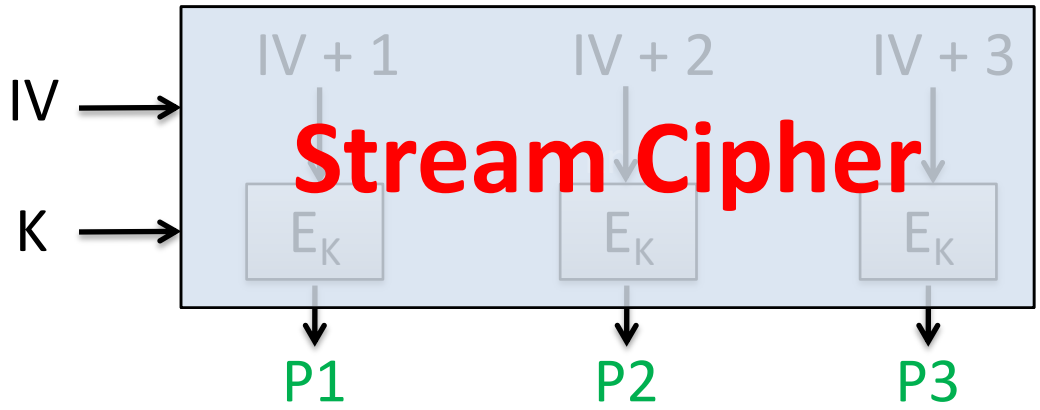
Therefore: Every encryption adds a new, independent random mask to the plaintext, and thus (by our previously established fact about masking), every ciphertext looks like a fresh random string!

Counter Mode and PyCrypto

```
from os import urandom
from Crypto.Cipher import AES
from Crypto.Util import Counter
import binascii

plaintext = "Hello CS177 students!"

# generate random key
encrypt_key = urandom(16)

# create counter object using a random, 16-byte long IV
iv = urandom(16)
ctr = Counter.new(128, initial_value=int(binascii.hexlify(iv), 16))

# create ``AES object''
cipher = AES.new(encrypt_key, AES.MODE_CTR, counter=ctr)

# create ciphertext
ciphertext = iv + cipher.encrypt(plaintext)

# decrypt plaintext; warning, this requires re-setting the counter

d_iv = ciphertext[:16]
d_ctr = Counter.new(128, initial_value=int(binascii.hexlify(d_iv), 16))
d_cipher = AES.new(encrypt_key, AES.MODE_CTR, counter=d_ctr)
d_plaintext = d_cipher.decrypt(ciphertext[16:])
```

Stream Ciphers

Masking-generation part can be abstracted through the concept of a **stream cipher**



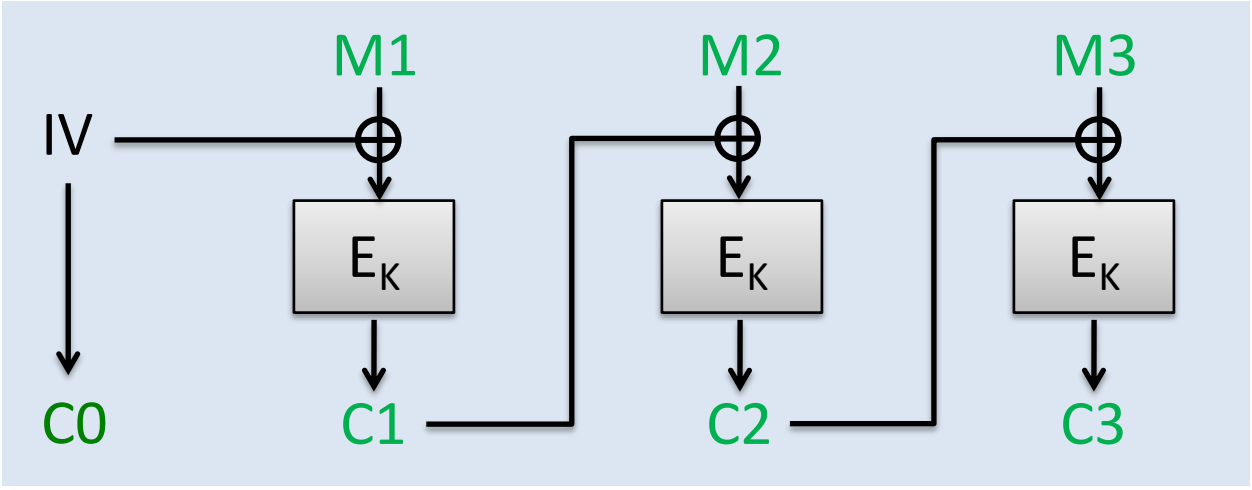
Efficient stream ciphers (not using block ciphers) exist:

- **RC4** -- completely broken, do not use, but has had a hard time dying out
  http://www.securityweek.com/new-attack-rc4-based-ssltls-leverages-13-year-old-vulnerability
- **Better: Salsa20/ChaCha**
  Especially fast on architectures without AES-NI hardware support
  Google likes it: https://security.googleblog.com/2014/04/speeding-up-and-strengthening-https.html

Alternative: Ciphertext Block Chaining (CBC)

Algorithm $\mathrm{Enc}(K, M)$:
Split M into blocks $M[1], \ldots, M[r]$
// all blocks are n-bits
Pick random $\mathrm{IV} \in \{0,1\}^n$
$C[0] \leftarrow \mathrm{IV}$
for $i = 1, \ldots, r$ **do**
$P[i] \leftarrow E_K(M[i] \oplus C[i-1])$
return $C[0], C[1], \ldots, C[r]$

Note: One needs to make the message length a multiple of n bits. This can be tricky (see next class!)



CBC and PyCrypto

```
from os import urandom
from Crypto.Cipher import AES
from Crypto.Util import Counter
import binascii

BLOCK_SIZE = 16
pad = lambda s: s + (BLOCK_SIZE - len(s) % BLOCK_SIZE) * chr(BLOCK_SIZE - len(s) % BLOCK_SIZE)
unpad = lambda s: s[:-ord(s[len(s) - 1:])]

plaintext = "Hello CS177 students!"

# generate random key
encrypt_key = urandom(16)

# create random, 16-byte long IV
iv = urandom(16)

# create ``AES object''
cipher = AES.new(encrypt_key, AES.MODE_CBC, iv)

# create ciphertext
ciphertext = iv + cipher.encrypt(pad(plaintext))

# decrypt plaintext; warning, this requires re-setting the counter

d_iv = ciphertext[:16]
d_cipher = AES.new(encrypt_key, AES.MODE_CBC, d_iv)
d_plaintext = unpad(d_cipher.decrypt(ciphertext[16:]))
```

CBC vs CTR

- CBC offers roughly same security as CTR
  - Security argument is less clear than in CTR
- For historical reasons, CBC more popular than CTR
- CTR is potentially faster than CBC, since the latter is inherently sequential, whereas CTR can be parallelized
- CBC requires padding the message to a length multiple of n, whereas CTR does not
  - This will be a problem

A warning

Do not trust everything you read on crypto APIs

- e.g., PyCrypto provides basic building blocks, but developers are not cryptographers
- Concepts often abstracted in different ways than they should, e.g. ”AES object” in PyCrypto

MODE_CTR

CounTeR (CTR). This mode is very similar to ECB, in that encryption of one block is done independently of all other blocks. Unlike ECB, the block *position* contribute information leaks about symbol frequency.

https://www.dlitz.net/software/pycrypto/api/2.6/

Very bad/dangerous explanation:

- CTR is very different from ECB
- What does independent even mean?
- Such explanations suggest ECB and CTR could both be used, which is extremely dangerous.

A somewhat better API

https://cryptography.io

```
import os
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend

backend = default_backend()
key = os.urandom(32)

# in fact, nonce/iv  does not even need to be random, unique is ok
iv = os.urandom(16)
cipher = Cipher(algorithms.AES(key), modes.CTR(iv), backend=backend)
encryptor = cipher.encryptor()
ct = encryptor.update(b"a secret message") + encryptor.finalize()

decryptor = cipher.decryptor()

d_plaintext = decryptor.update(ct) + decryptor.finalize()

print(d_plaintext)
```

A somewhat better API

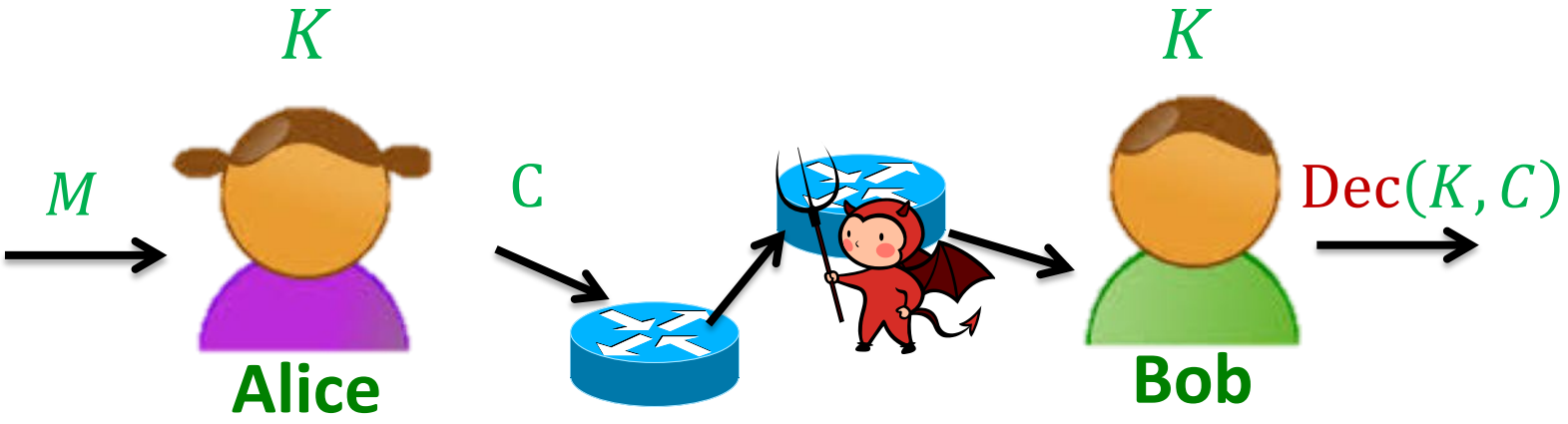
https://cryptography.io

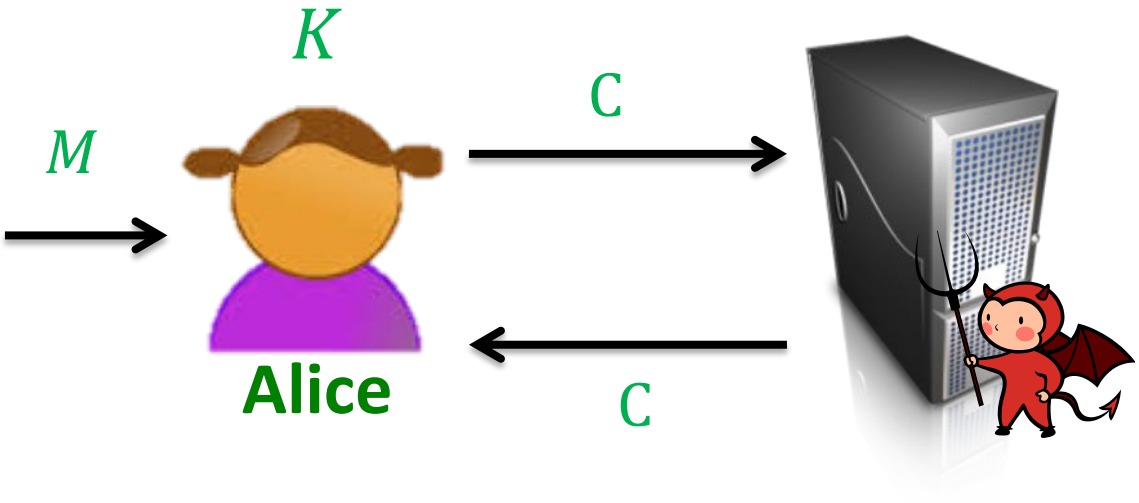
```
import os
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend

backend = default_backend()
key = os.urandom(32)
iv = os.urandom(16)
cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=backend)
encryptor = cipher.encryptor()
ct = encryptor.update(b"a secret message") + encryptor.finalize()

decryptor = cipher.decryptor()

d_plaintext = decryptor.update(ct) + decryptor.finalize()

print(d_plaintext)
```

Take-home messages – Encryption & confidentiality

- ECB is insecure
- Semantic security is the “right” notion for confidentiality
- Two examples of semantically secure schemes: CTR and CBC
- CTR works because of “masking trick” – try to get an intuition for this!
- Experiment with some crypto API and try to run these schemes

Next: Is confidentiality everything we want?





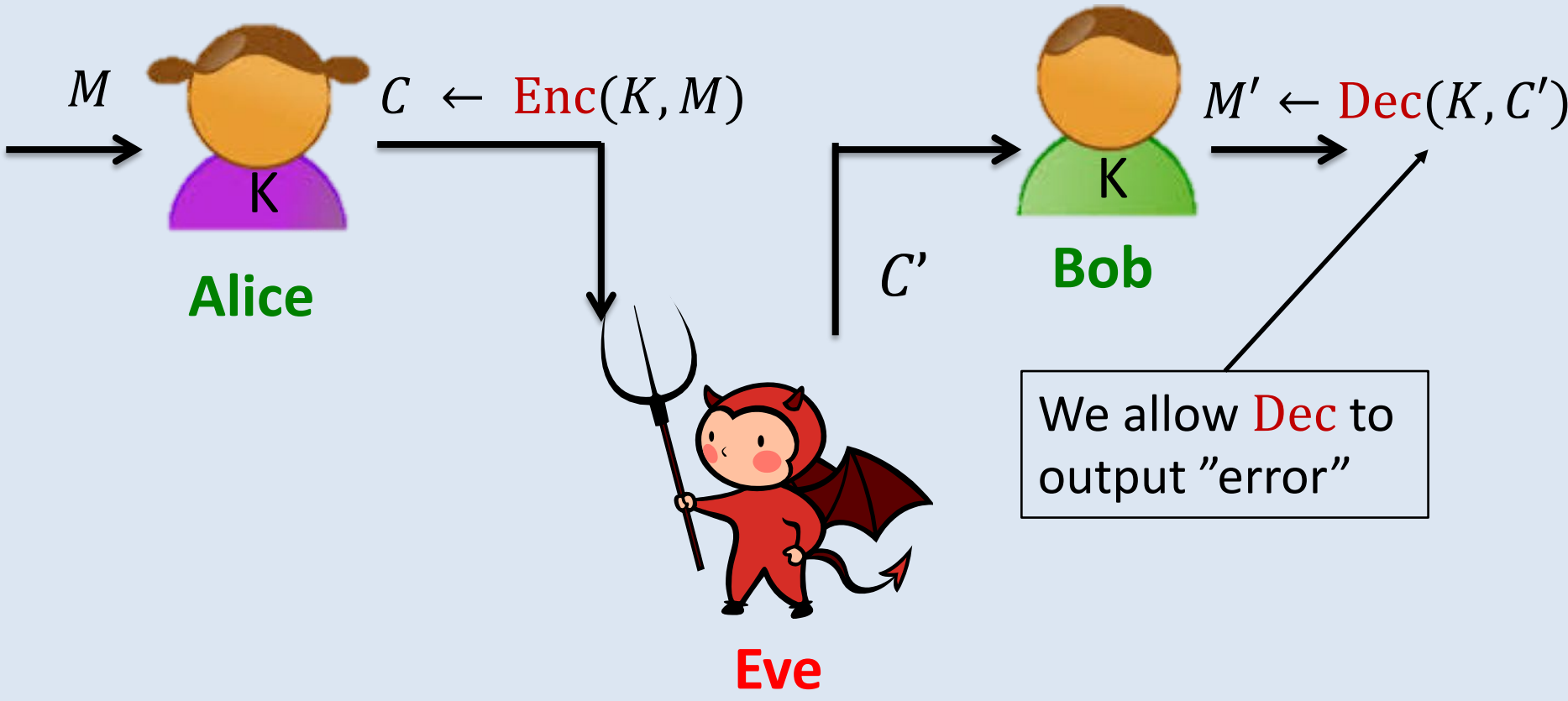
Confidentiality is not the only goal

We also want to make sure that the encryption scheme guarantees **integrity**

Imagine Eve tampers with ciphertexts sent by Alice to Bob, then Bob must be able to detect it!

Encryption which guarantees both confidentiality and integrity is referred to as Authenticated Encryption (AE)

Encryption Integrity – Abstract scenario



Scheme satisfies **integrity** if it is unfeasible for Eve to send C' not previously sent by Alice such that $\mathrm{Dec}(K, C') \neq \text{error}$

CTR and Integrity

Back to CTR example, imagine Eve sees the following ciphertext
[remember: it encrypts “Hello CS177 students!”, but Eve does not know this]

C

CC 32 FA B3 E9 12 47 81 FF 1B 3C D6 AA 98 42 03	85 5B EE F4 08 4C FC 3A 8B F5 50 C2 39 99 73 0E	56 4C 70 20 91 3A

CC 32 FA B3 E9 12 47 81 FF 1B 3C D6 AA 98 42 03	85 5B EE F4 08 4C FC 3A 8B F5 5**F** C2 39 99 73 0E	56 4C 70 20 91 3A

C'

Eve just changed four bits from 0 to 1, and sends C' to Bob.
Bob attempts to decrypt. What does he get?

CTR and Integrity – cont'd

85 5B EE F4 08 4C FC 3A 8B F5 5**F** C2 39 99 73 0E	56 4C 70 20 91 3A

$\oplus$

CD 3E 82 98 67 6C BF 69 BA C2 67 E2 4A ED 06 6A	33 22 04 53 B0 3A

Bob decrypts by adding the mask back

48 65 6C 6C 6F 20 43 53 31 37 3**8** 20 73 74 75 64	65 6E 74 73 21 00

Which is the ASCII encoding for "Hello CS17**8** students!"

What happened? Eve flipped a few bits and produced a valid encryption for something that Alice never meant to send. **NO integrity!**

Important message

"Classical" modes of operation like CTR and CBC never guarantee integrity, and should never be used by themselves.

Next: we will see how to fix it and build AE

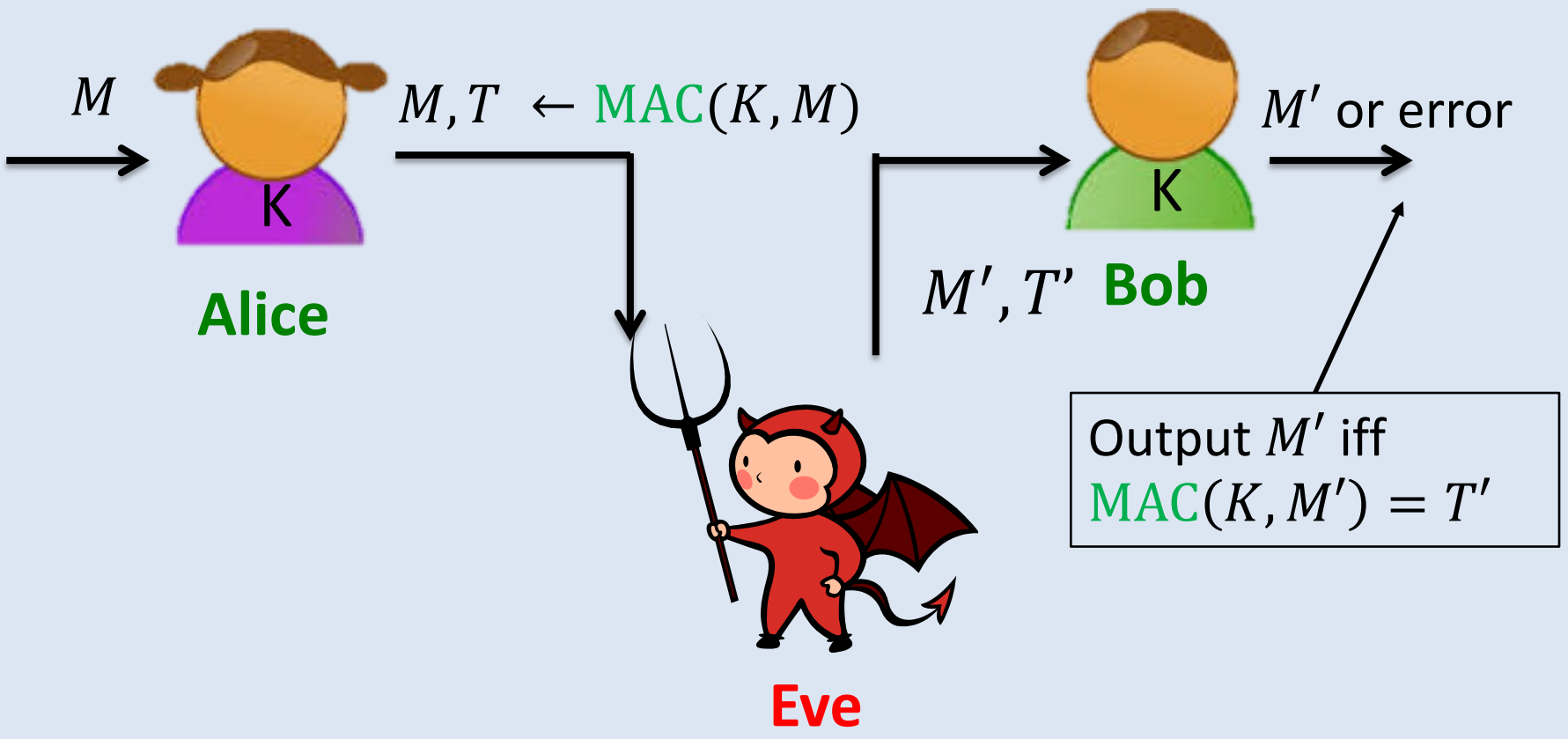
Message Authentication

Message Authentication Code (MAC) is an efficient algorithm that takes a secret key, a string of arbitrary length, and outputs an (unpredictable) short output/digest.

$$\text{MAC}: \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^n$$

$$\text{MAC}(K, M) = \text{MAC}_K(M) = T$$

Message Authentication – Scenario



MAC satisfies **unforgeability** if it is unfeasible for Eve to let Bob output M' not previously sent by Alice.

MAC example

Note: No encryption in this example, this is only about integrity!

M = "Hello CS177 students!"

$T = \mathrm{MAC}(K, M) =$ 5f 68 18 21 b7 f5 4f b1 10 3d fd fa 89 0e ca 1d 42 10 7d 2f

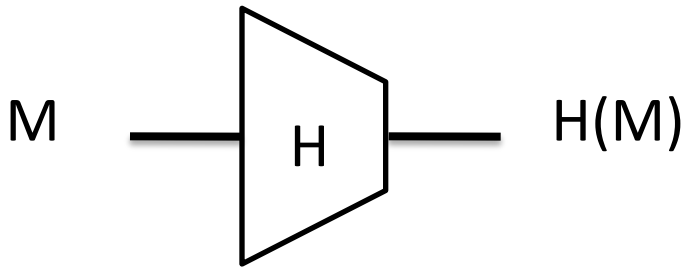
M' = "Hello CS17**8** students!"

$T' = \mathrm{MAC}(K, M') =$???

Any guess likely incorrect!

Hash functions and message authentication

Hash function H maps arbitrary bit string to fixed length string of size m



MD5: m = 128 bits
SHA-1: m = 160 bits
SHA-256: m = 256 bits
SHA-3: m >= 224 bits

Some security goals:

- collision resistance: can't find M != M' such that H(M) = H(M')
- preimage resistance: given H(M), can't find M
- second-preimage resistance: given H(M), can't find M' s.t. H(M') = H(M)

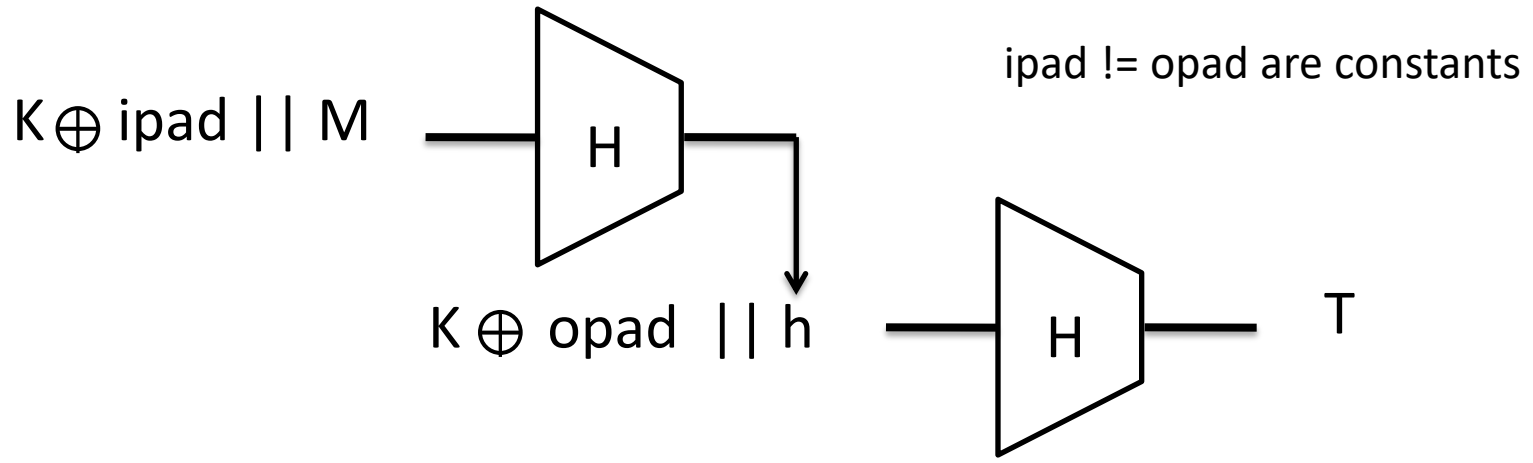
Hash-function side-note

- MD5 and SHA-1 are, for all practical purposes, broken
  - Never use them in anything you are going to develop and/or deploy!
- SHA-256, SHA-512, SHA-3 all ok

Message authentication with HMAC

Goal: Use a hash function H to build MAC.

HMAC(K,M) defined by:



Unforgeability holds if H is secure in some well-defined sense

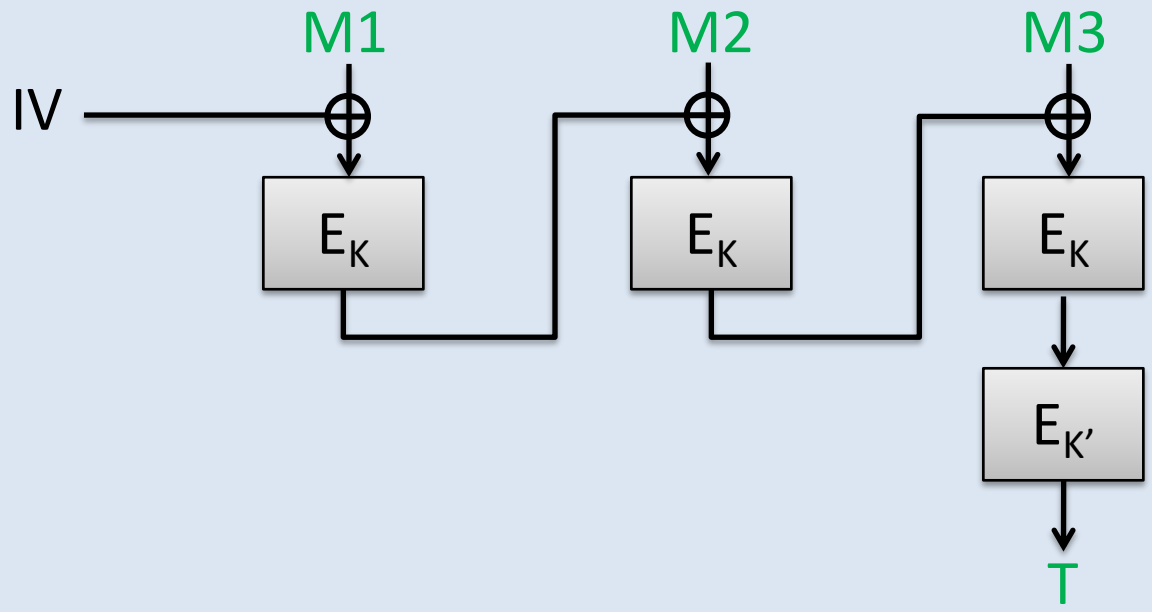
Important

Hash function ≠ MAC

A hash function takes no key, a MAC is a secret-key primitive

Helpful intuition: A MAC is like a hash function which can only be evaluated by those having the secret key.

Other option: CBC-MAC



Here: IV is a fixed value, K, K' are two different keys (can be generated from one, depending on the variant)

Theorem: Unforgeability holds if E is a secure PRP

Many variants of CBC-MAC exists, but usage is becoming less and less prominent.

Take-home messages – Integrity

- CTR/CBC never satisfy integrity (why?)
- We want to have confidentiality and integrity in encryption (aka, authenticated encryption)
- What is a hash function? What is a MAC?
- MACs are used to provide integrity of data, but never confidentiality!
- MACs can be built from hash functions (HMAC) or block ciphers (CBC)