CS 177
Computer Security
Lecture 4

Stefano Tessaro
tessaro@cs.ucsb.edu

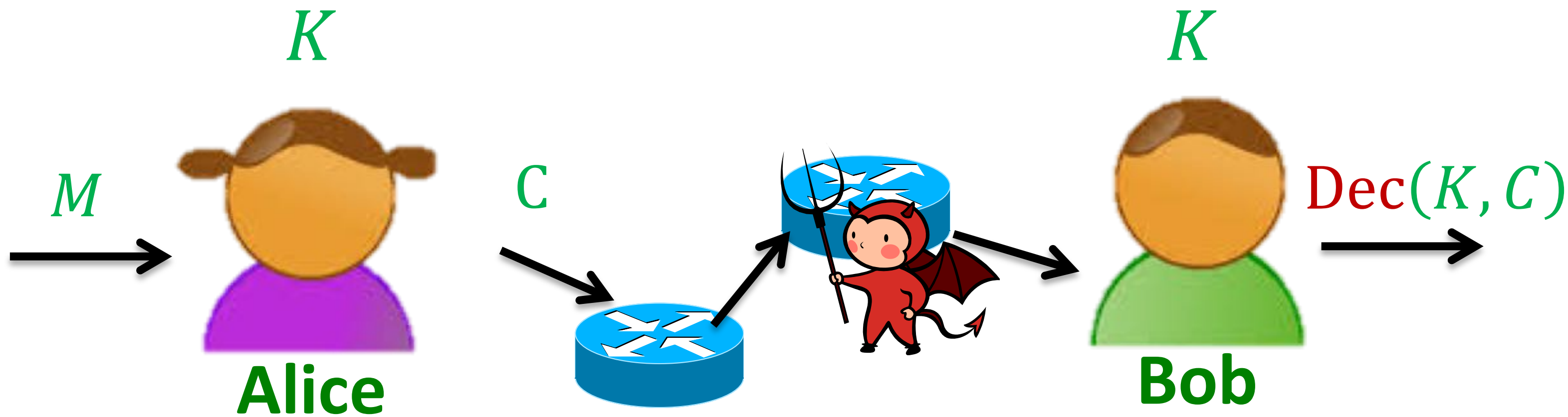
Announcements

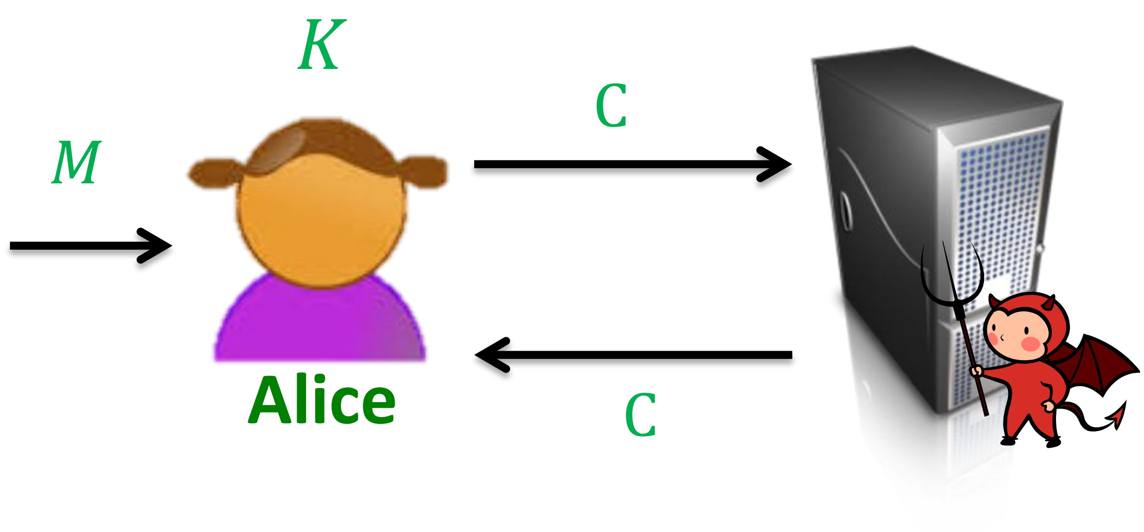
- HW1 due on Saturday
  - Instructions soon online
- HW2 posted tomorrow
- Office hours announced on Piazza
  - Kevin’s office hours today are canceled due to travel.

Today – Integrity

- MACs
- Authenticated encryption
  - Encrypt-then-MAC
  - Dedicated modes of operation
- Case studies
  - WEP security
  - Encrypt-then-MAC

Next: Is confidentiality everything we want?



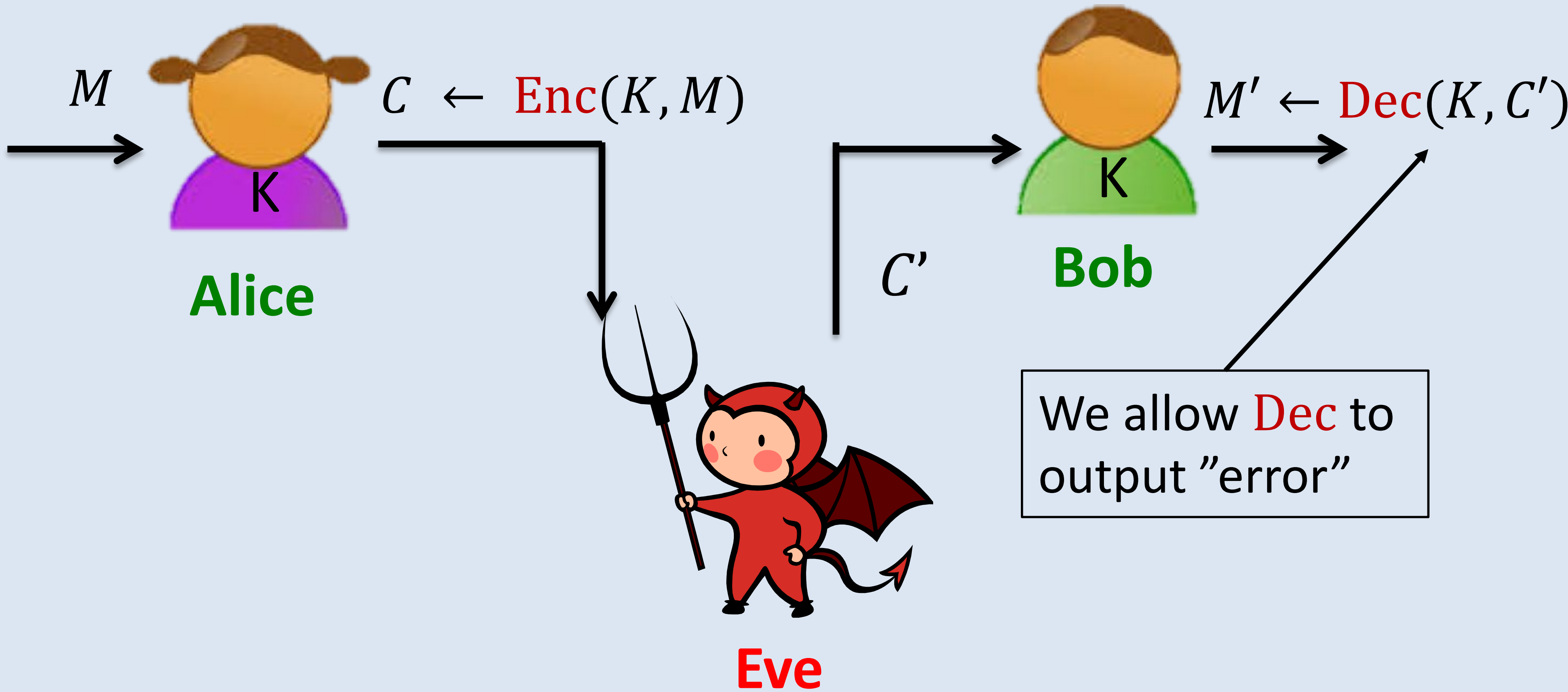


Confidentiality is <u>not</u> the only goal

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 Integrity – Abstract scenario



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

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.

Today – Authenticated Encryption

AE = confidentiality + integrity

One of the trickiest topics in cryptography:

- Many mistakes here have led to attacks
- Badly treated by current textbooks
- Misunderstanding is historically rooted

Central tool to achieve integrity: **Message-authentication codes** (MACs)

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.

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

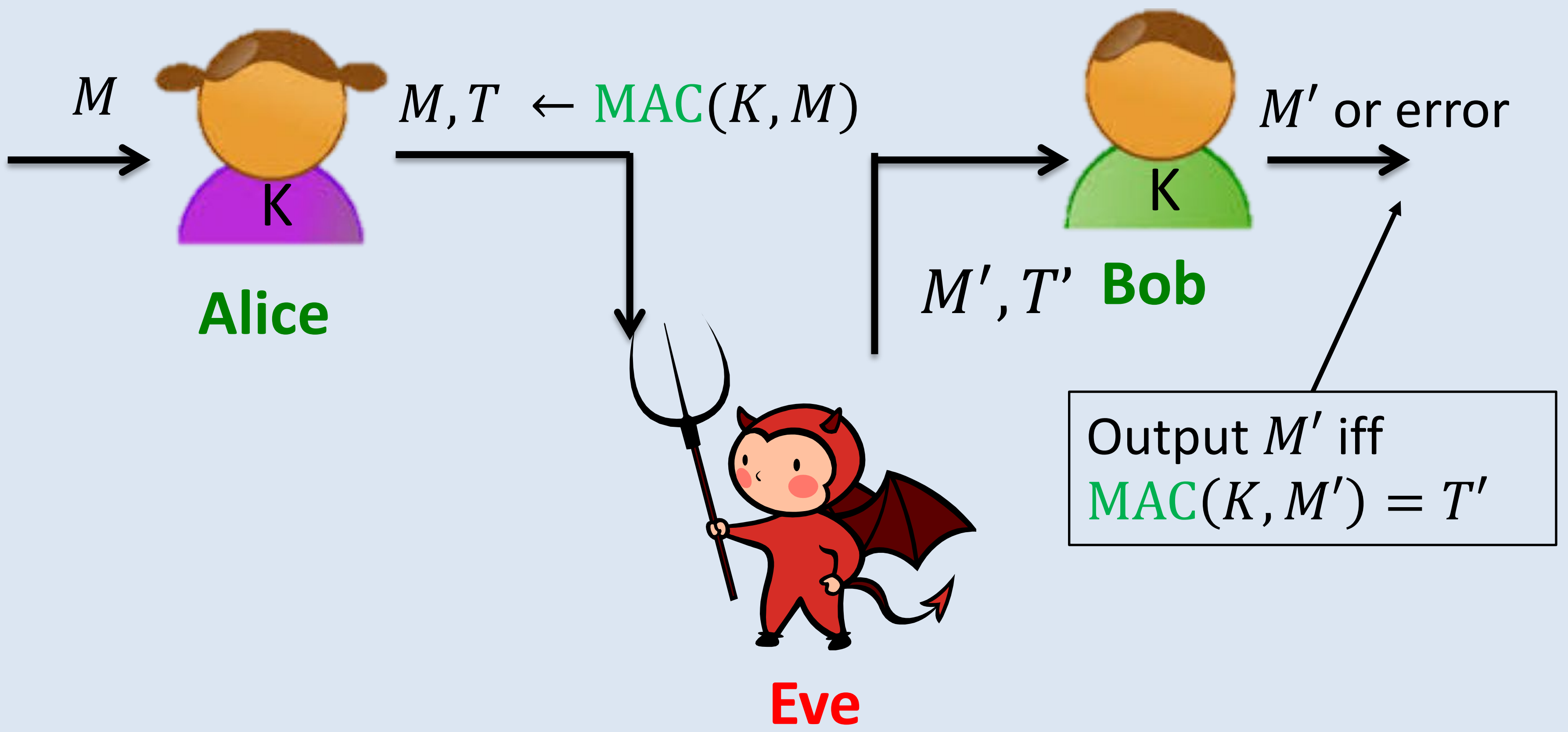
$$\mathrm{MAC}(K, M) = \mathrm{MAC}_K(M) = T$$

key

message

tag

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!"

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

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

Any guess likely incorrect!

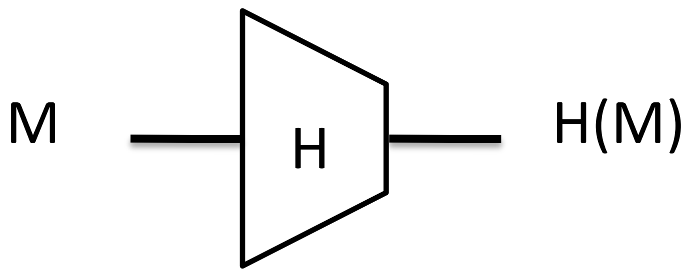
Baseline

- Knowing the key allows to compute/recompute the message tag.
- Not knowing the key makes the tag unpredictable (unless we have seen it already).

Hash functions and message authentication

Many MACs are built from **cryptographic hash functions**

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)

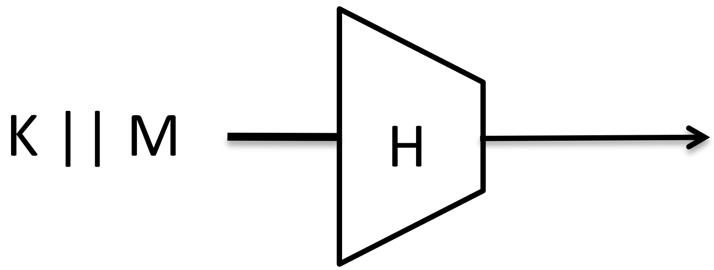
Hash-function side-note

- MD5 and SHA-1 are broken
  - Never use them in anything you are going to develop and/or deploy!
  - https://www.youtube.com/watch?v=NbHL0SYlrSQ
- SHA-256, SHA-512, SHA-3, BLAKE2 all ok
- SHA-256/SHA-512 most widely used

Message authentication with hash functions

Goal: Use a hash function H to build MAC.

MAC(K, M) = H(K || M)



In other words: The MAC is the hash of the concatenation of the key and the message.

Good option for SHA-3 / BLAKE2

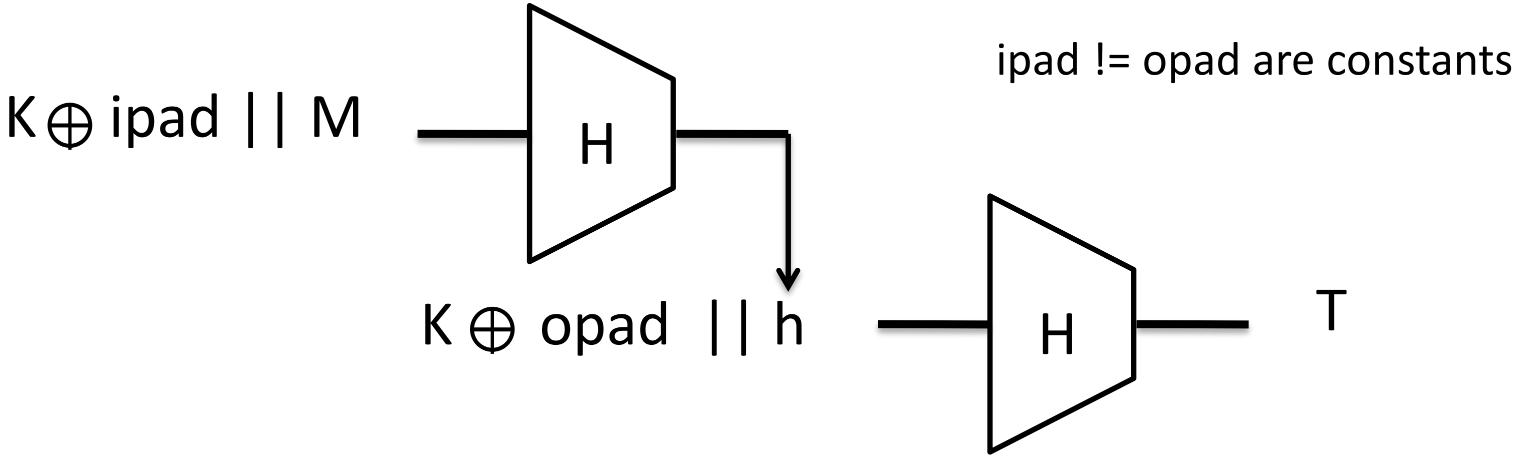
Completely insecure for SHA-256/SHA-512 (and legacy hashes)

See homework!

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

No attacks in particular for SHA-256/SHA-512

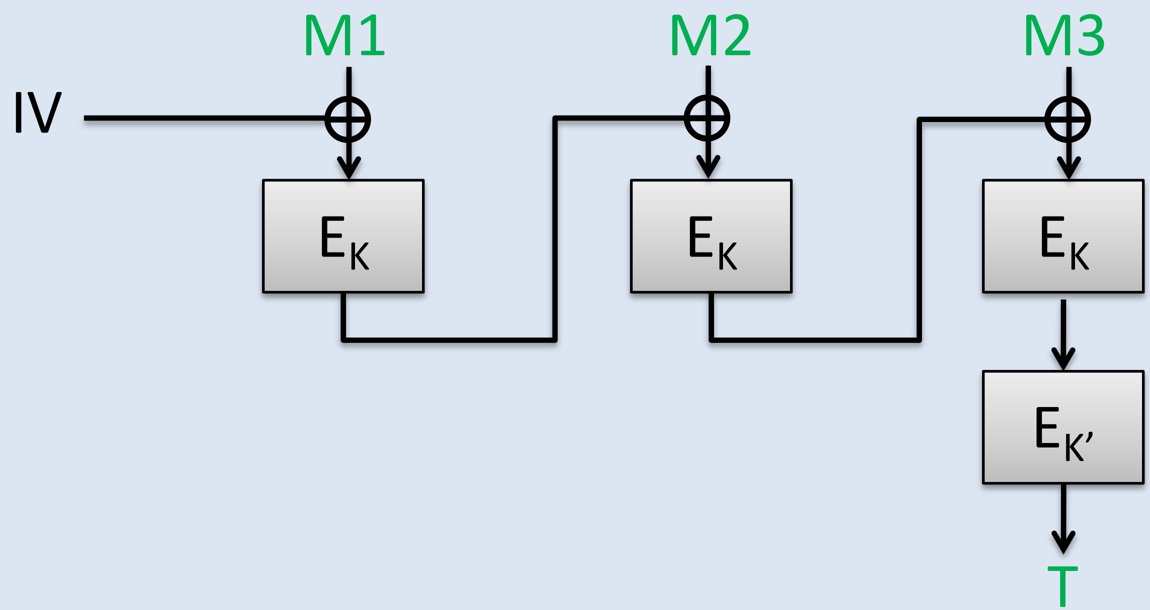
Important

Hash function $\neq$ 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 (the above is typically called "EMAC"), but usage is becoming less and less prominent.

In fact, there are even more MACs …

- See this short humorous talk (happened in Corwin …) https://www.youtube.com/watch?v=2gLumNRmqjs
- Mostly you will deal with HMAC and (occasionally) CBC.

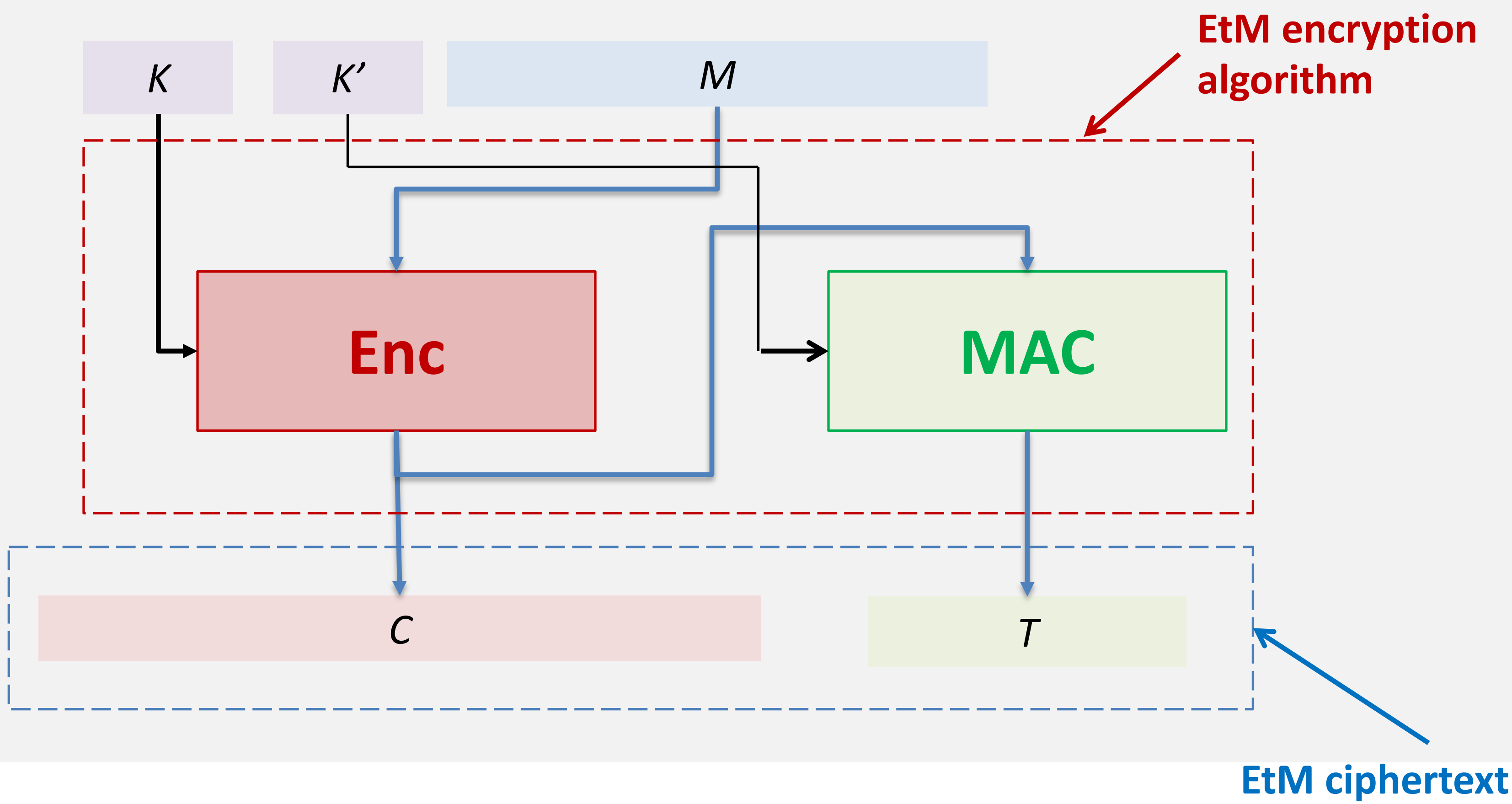
How to achieve integrity?

Combine a MAC and a semantically secure encryption scheme!

Best solution: **Encrypt-then-MAC**

Encrypt-then-MAC

EtM key consists of two keys
(one for Enc, one for MAC)



Decryption: Given $C^* = (C, T)$, first check T valid tag for C using K'
- If so, decrypt C, and output result
- If not, output "error"

Encrypt-then-MAC – why is it secure?

EtM is secure as long as encryption scheme is semantically secure, and MAC is unforgeable!

Integrity. If the attacker sees $C^* = (C, T)$, and wants to change this to a valid $C^{**} = (C', T')$ where $C' \neq C$, then it needs to forge the MAC, i.e., produce a new tag T' for C'.

Confidentiality. $C^* = (C, T)$ does not leak more information about plaintext than C, because T is computed from C directly, and does not add extra information about plaintext.

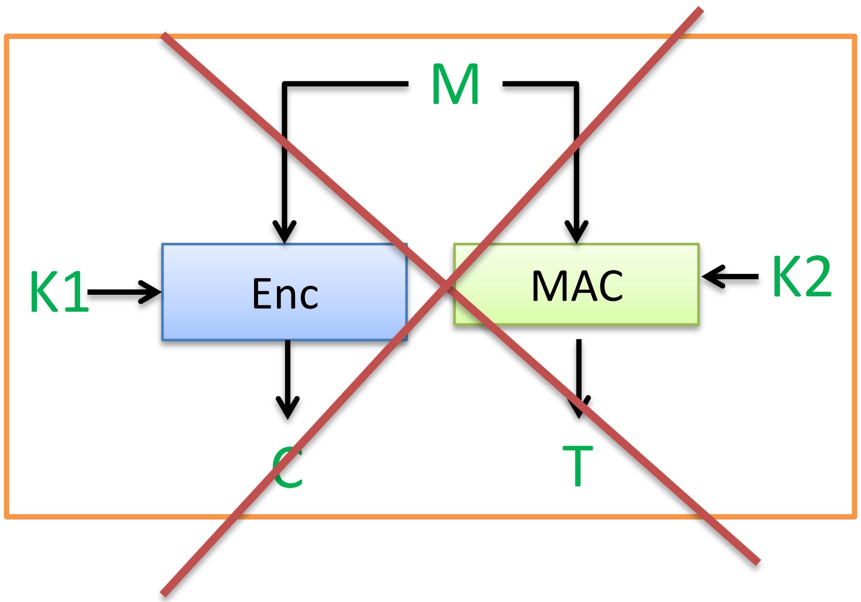
Encrypt-then-MAC

Valid combinations are e.g.

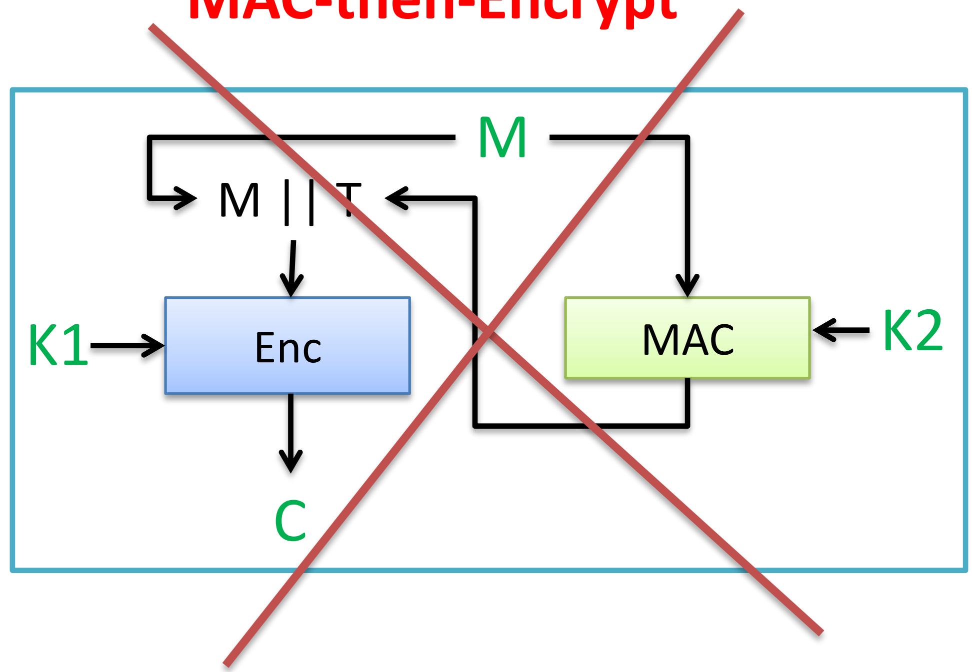
{AES-CTR, AES-CBC} + {AES-CBC-MAC, SHA-256-HMAC, SHA-512-HMAC}

Authenticated Encryption – Bad solutions

Encrypt-AND-MAC



MAC-then-Encrypt



Still, they are used all over the place, but just don't use them ☺

Authenticated Encryption – Ad-hoc solutions

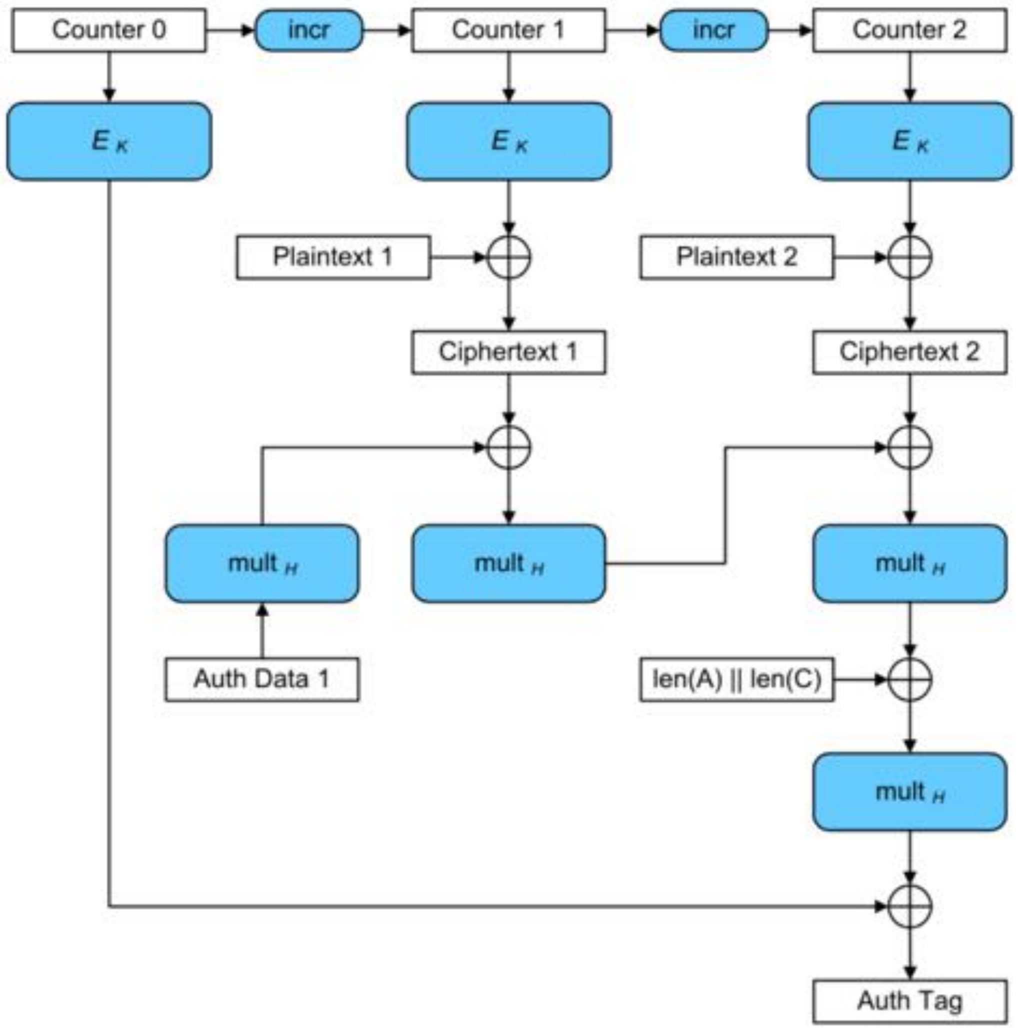
Typically faster than EtM!

Attack	Inventors	Notes
OCB (Offset Codebook)	Rogaway	One-pass
GCM (Galois Counter Mode)	McGrew, Viega	CTR mode plus specialized MAC
CWC	Kohno, Viega, Whiting	CTR mode plus Carter-Wegman MAC
CCM	Housley, Ferguson, Whiting	CTR mode plus CBC-MAC
EAX	Wagner, Bellare, Rogaway	CTR mode plus OMAC

Ongoing competition (CAESAR) meant to develop robust AE standard

Common solution – GCM

Essentially CTR-mode + a very lightweight MAC (widely used in TLS, Wgig, SSH,…)



The magic is inside mult, uses cool algebra! G in GCM stands for Galois!

source:
wikipedia.org

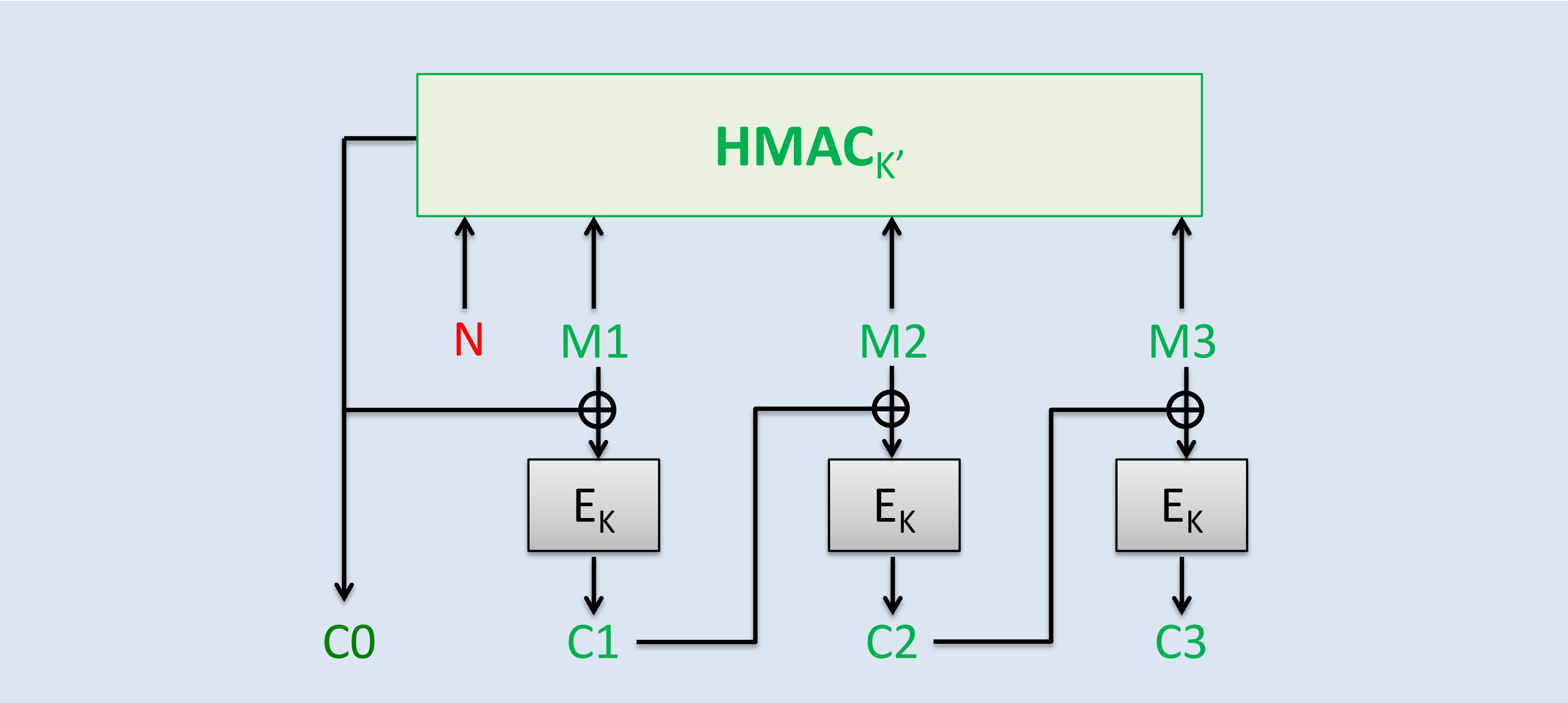
Modern interface – AEAD

https://tools.ietf.org/html/rfc5116

- Allows to send along authenticated *associated metadata* with the ciphertext which is not encrypted, used for functionality and efficiency.
- AEAD interface asks for externally supplied IV (often called a "nonce"), rather than generating it at random internally
  - Greater flexibility
  - Must ensure nonce are <u>never</u> re-used (GCM breaks if nonces are re-used!)
  - Right notion: Misuse resistant AEAD. If nonces are re-used, worst that can happen is that re-encrypting same message twice is detected.

Misuse-resistant AEAD – SIV

Example: SIV mode by Rogaway-Shrimpton



Example instantiation (here, w/o associated data): K, K' are secret keys, N is the nonce (to be sent along, typically random but does not need to be).

This offers strongest possible security for AE, inherent cost: Two passes (C0 needs to be computed first!)

AE misuse

- Not using AE, or using AE incorrectly, is one of the major sources of cryptographic disasters.
  - Nonce repeats are very common mistakes in WPA2!
- Two examples next: WEP and padding oracle attacks

Case study I: WEP

Wired Equivalent Privacy

- Authenticated encryption to protect wireless communication in original IEEE 802.11 Wifi standard.
- Subject to a number of flaws, allow gaining access / decrypting traffic within minutes

Breaking 104 bit WEP in less than 60 seconds

Erik Tews, Ralf-Philipp Weinmann, and Andrei Pyshkin
<e.tews,weinmann,pyshkin@cdc.informatik.tu-darmstadt.de>

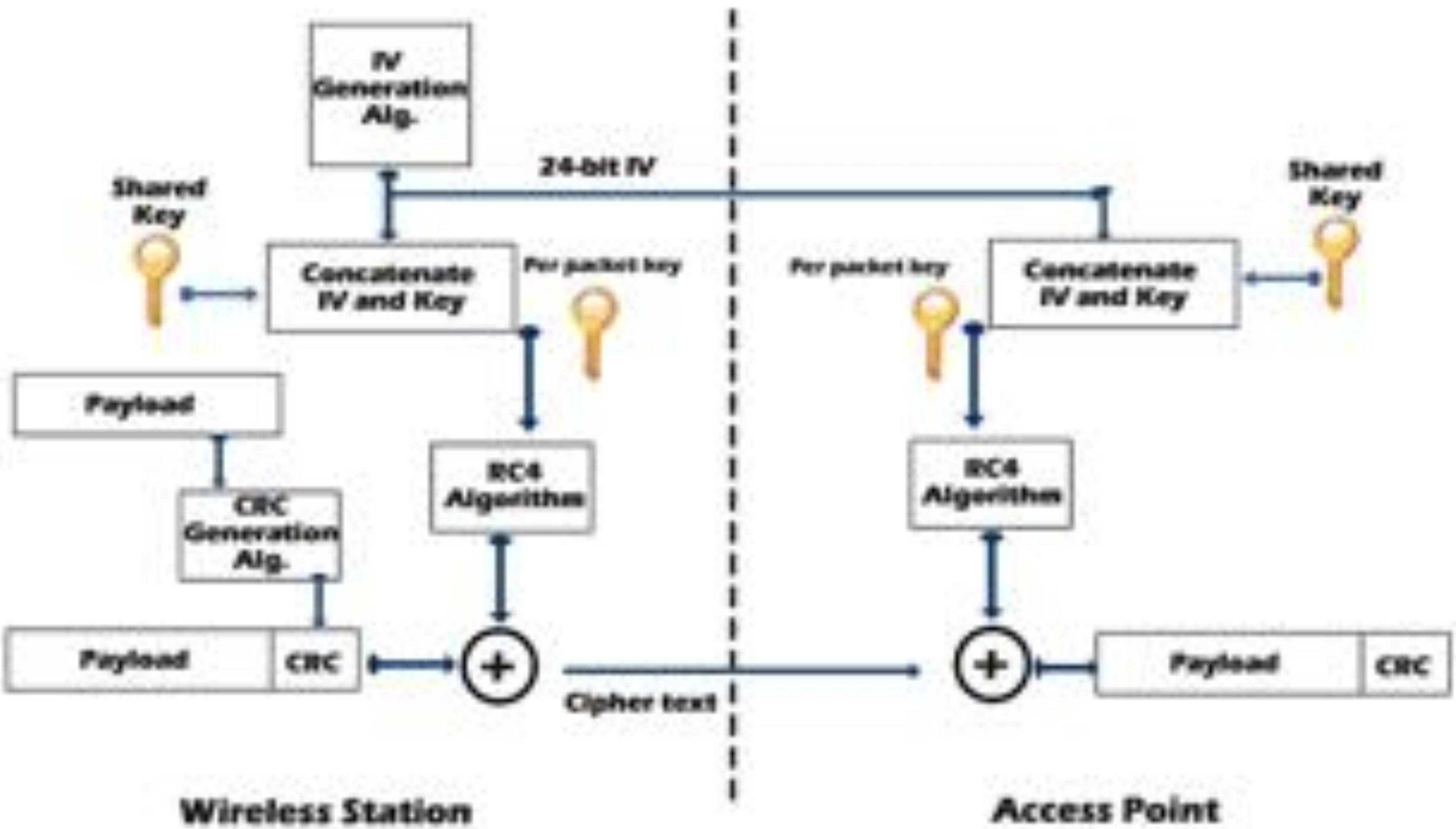
Technische Universität Darmstadt,
Fachbereich Informatik
Hochschulstrasse 10
D-64289 Darmstadt

Abstract. We demonstrate an active attack on the WEP protocol that is able to recover a 104-bit WEP key using less than 40.000 frames with a success probability of 50%. In order to succeed in 95% of all cases, 85.000 packets are needed. The IV of these packets can be randomly chosen. This is an improvement in the number of required frames by more than an order of magnitude over the best known key-recovery attacks for WEP. On a IEEE 802.11g network, the number of frames required can be obtained by re-injection in less than a minute. The required computational effort is approximately 2^{20} RC4 key setups, which on current desktop and laptop CPUs is neglegible.

1 Introduction

Wired Equivalent Privacy (WEP) is a protocol for encrypting wirelessly transmitted packets on IEEE 802.11 networks. Although it is known to be insecure and has been superseded by Wi-Fi Protected Access (WPA) [8], it still is in widespread use. In a WEP-protected network, radio stations share a common key, the *root key* Rk. A successful recovery of this key gives

WEP "Encryption"



http://www.cs.wustl.edu/~jain/cse574-06/ftp/wireless_security/fig14.gif

WEP Problems – Confidentiality

- RC4 is a by now insecure stream cipher, meant to produce pseudorandom key-stream
- Even if RC4 secure, re-using IV would produce the same key-stream
- 24-bit is too short. IV re-used after $2^{12} = 4096$ encryptions

WEP Problems – Integrity

- CRC (cyclic-redundancy check) does not provide integrity
  - It is a linear function L of the plaintext
  - Ciphertext has form
    $$C = (M||L(M)) \oplus P$$
    where P is bitstream mask generated by RC4
  - To transform C into an encryption of $M \oplus M'$, simply compute
    $$C' = C \oplus (M'||L(M')) = ((M \oplus M')||L(M) \oplus L(M')) \oplus P$$
    $$= (M \oplus M'||L(M \oplus M')) \oplus P$$

Case study II: Padding-oracle attacks

Sometimes, need of integrity is <u>not</u> obvious

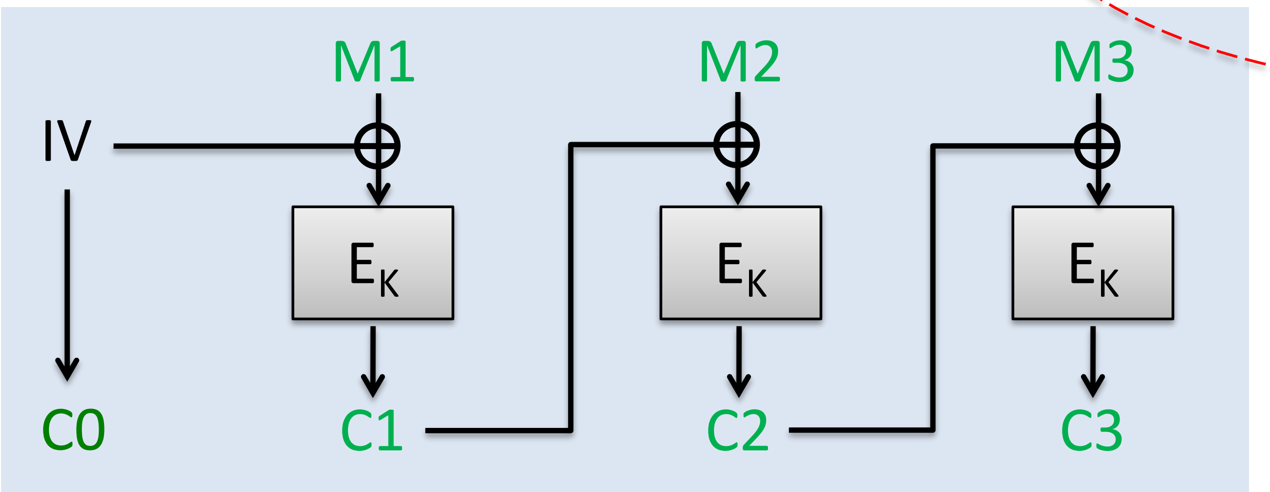
Ciphertext Block Chaining (CBC)

Very popular alternative to CTR (for no real reason)

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**
$\quad P[i] \leftarrow E_K(M[i] \oplus C[i-1])$
return $C[0], C[1], \ldots, C[r]$

Padding!

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



How to pad?

Non-trivial! Common solution: think of M as being made of bytes. Typical block length: 128 bits = 16 bytes

Attempt #1:

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

Good idea? NO! Ambiguity! Following two pieces of data padded to the same!

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

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

Note: It does not matter these give the same string, encryption should work independent of data interpretation. You get what you encrypt!

How to pad?

PKCS #7 padding

Look how many bytes are missing. Here, 10 = 0x0A

Fill remaining k bits with value k

48 65 6C 6C 6F 20 43 53 31 37 37 20 73 74 75 64	65 6E 74 73 21 00	0A 0A 0A 0A 0A 0A 0A 0A 0A 0A

To decode: Recover value of last byte $k \in \{01, 02, \dots, 0A, 0B, \dots, 0F, 10\}$, remove last k bytes if they are all equal. If something does not match, abort!

What if data length is already a multiple of 16?

48 65 6C 6C 6F 20 43 53 31 37 37 20 73 74 02 02	10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10

Cannot leave it unchanged, as last two bytes are interpreted as padding!

Invalid padding

Fact: Not every 32-byte string is a valid padding
Decryption must output “error” if it recovers incorrect padding.

48 65 6C 6C 6F 20 43 53 31 37 37 20 73 74 75 64	65 6E 74 73 21 00 0A 0A 0A 0A 0A 0A 0A 0A 0A 02

Will give error! Why?

48 65 6C 6C 6F 20 43 53 31 37 37 20 73 74 75 64	65 6E 74 73 21 00 0A 0A 0A 0A 0A 0B 0A 0A 0A 0A

Will give error! Why?

48 65 6C 6C 6F 20 43 53 31 37 37 20 73 74 75 64	65 6E 74 73 21 00 0A 0A 0A 0A 0A 0A 0A 0A 0A 01

No error! You never get an error when ending with 01!

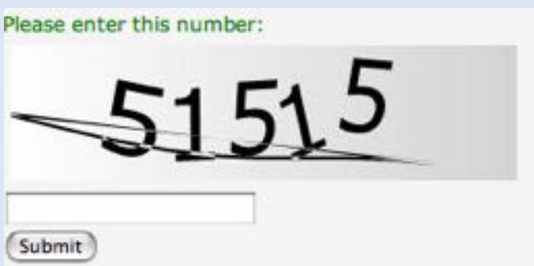
Example – CAPTCHAs

Motivated by real attack against Ruby on Rails, ASP.NET

Contains chal image, and hard-codes
$C^* \leftarrow \text{CBC}-\text{Enc}(K, \text{"51515"})$

Server generates challenge number (e.g., 51515), and jpg-image with distorted version of challenge

auth_form.html



guess="51517", cipher=C^*

Secret key K

OK / N_OK / ERROR

Check C^* decrypts to guess

right / wrong guess

= padding invalid

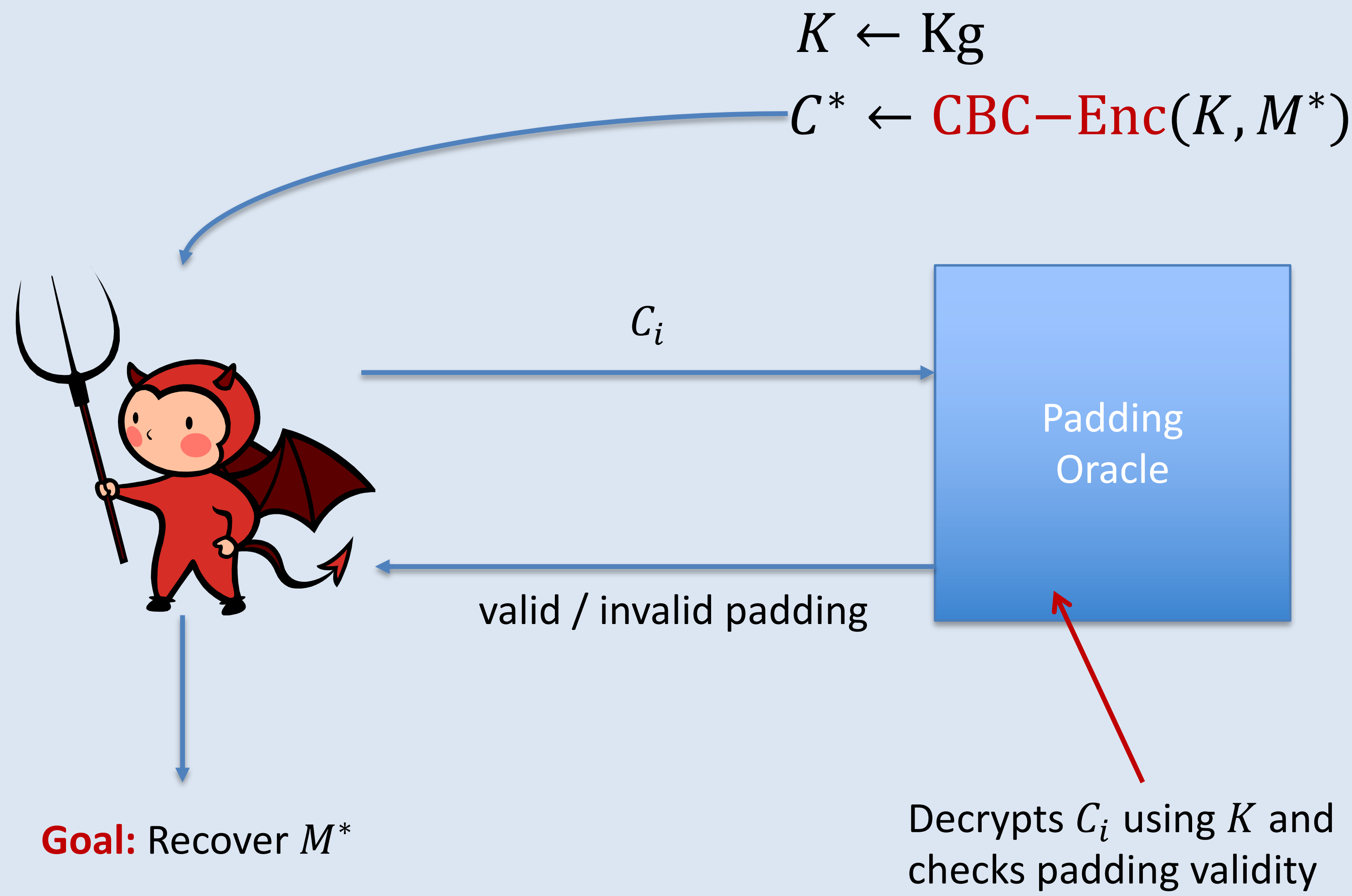
Why does this happen? Servers cannot hold state of web applications to save memory, so it's stored (in encrypted form) in generated contents, and sent back along with form data

CAPTCHAs continued

- Padding correctness information can be used to recover C^* without actually solving the puzzle = "Padding-oracle attack"
  - e.g., can be done by algorithm without fancy deep-learning vision techniques ;)
- Possible solution: Merge N_OK and ERROR
  - Still not perfect, other side information (e.g., response timing) can leak which one is the case
  - e.g., Lucky13 is a padding-oracle attack against MAC-then-Encrypt in TLS using timing http://www.isg.rhul.ac.uk/tls/Lucky13.html

Padding oracles – The (abstract) attack setting

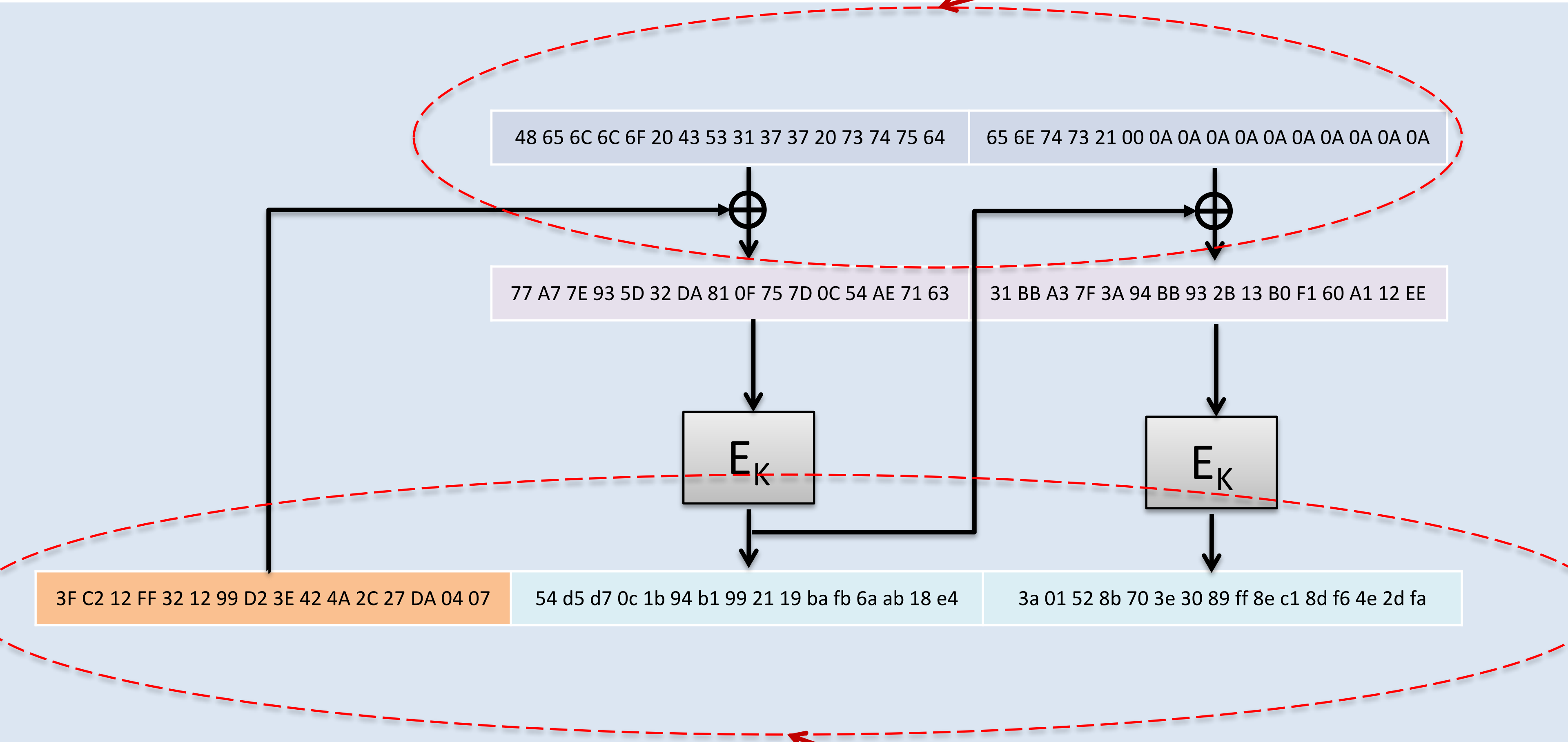
[Vaudenay, 2002]



Where does a padding oracle come from?

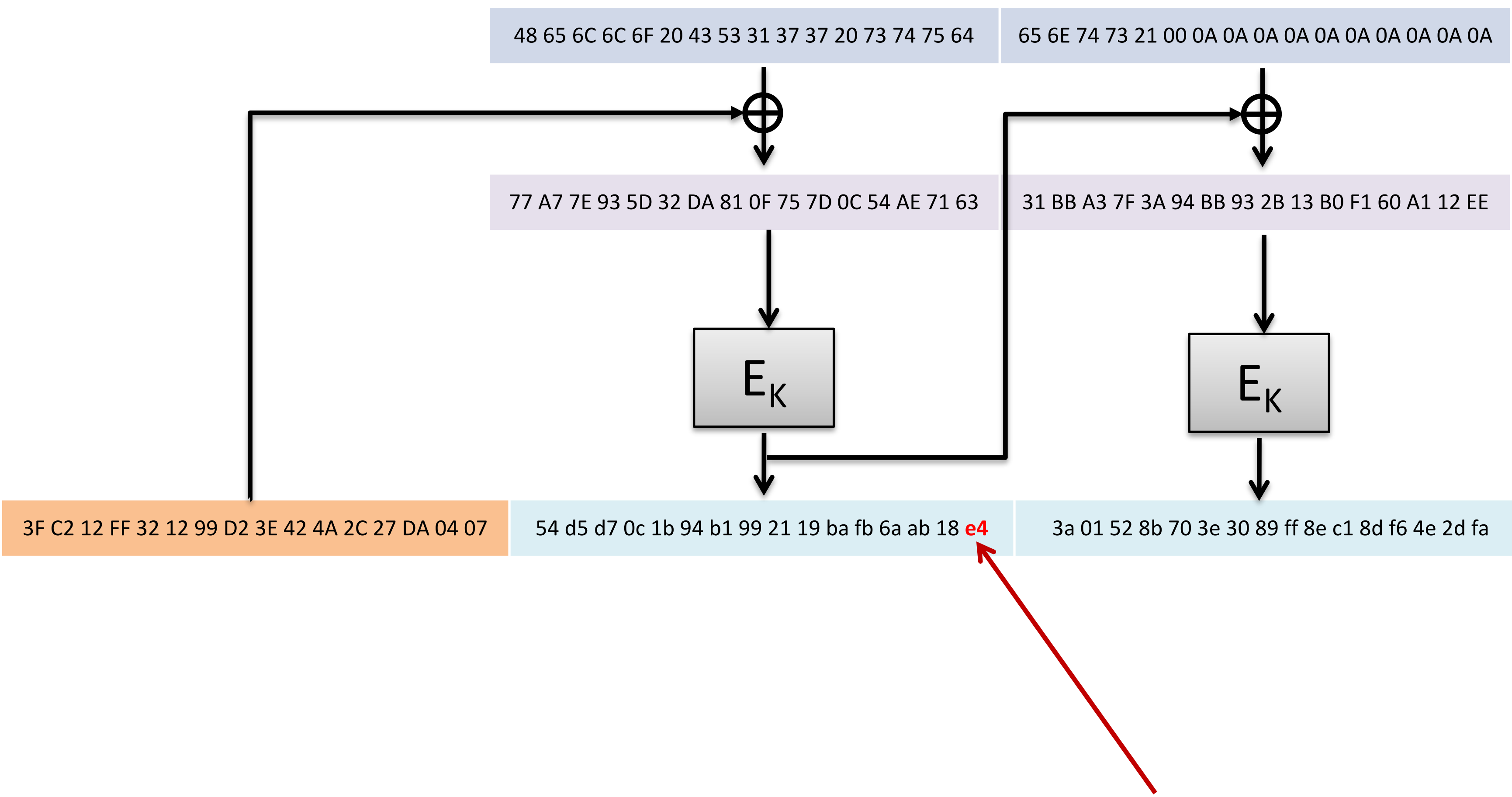
The attack – Setting

we want (all of) this



all we know is this (+ access to the padding oracle!)

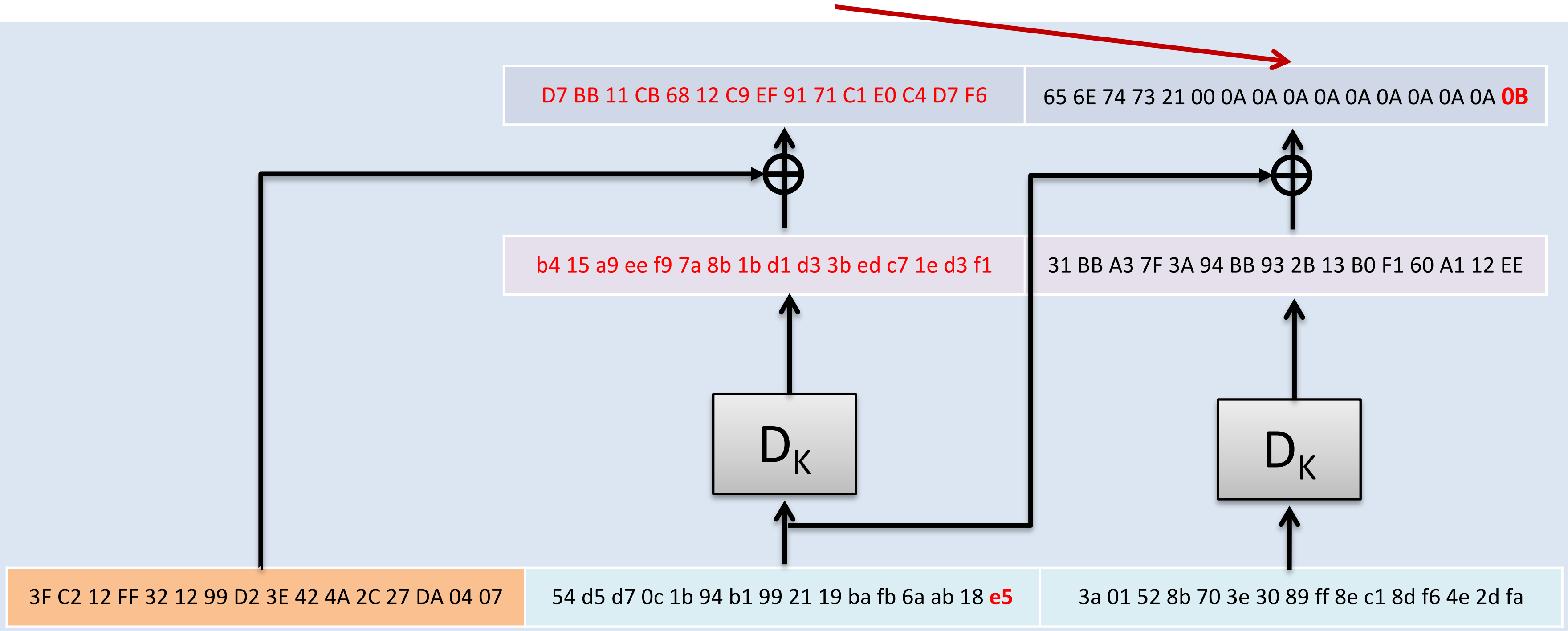
The attack – Last byte



Assume we change last byte from **e4** to **e5**, what happens?

The attack – Last byte

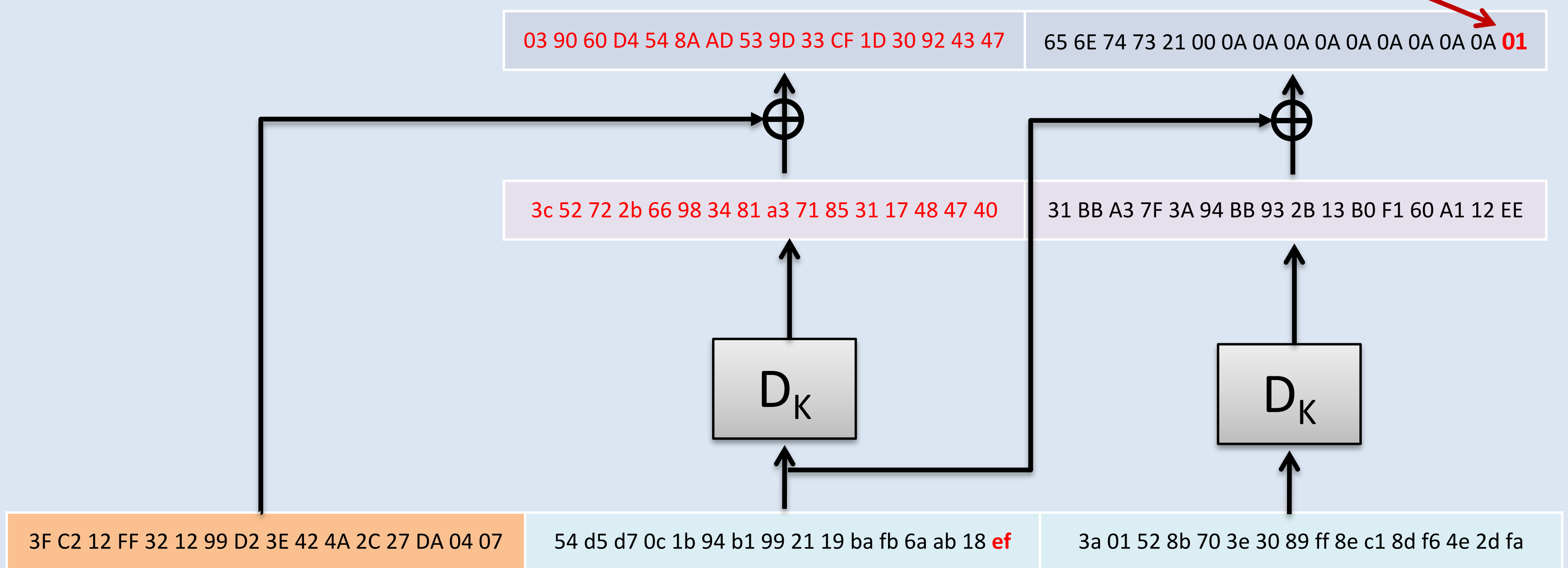
Note: Padding is invalid – since last two bytes are 0A 0B, so this new ciphertext would result in a padding error if submitted to the padding oracle



Can we modify the last byte of the second-last ciphertext block so that decryption of new ciphertext does not lead to padding error? And gain useful information?

Answer – Yes!

Padding is now valid! Padding oracle would return valid given modified ciphertext

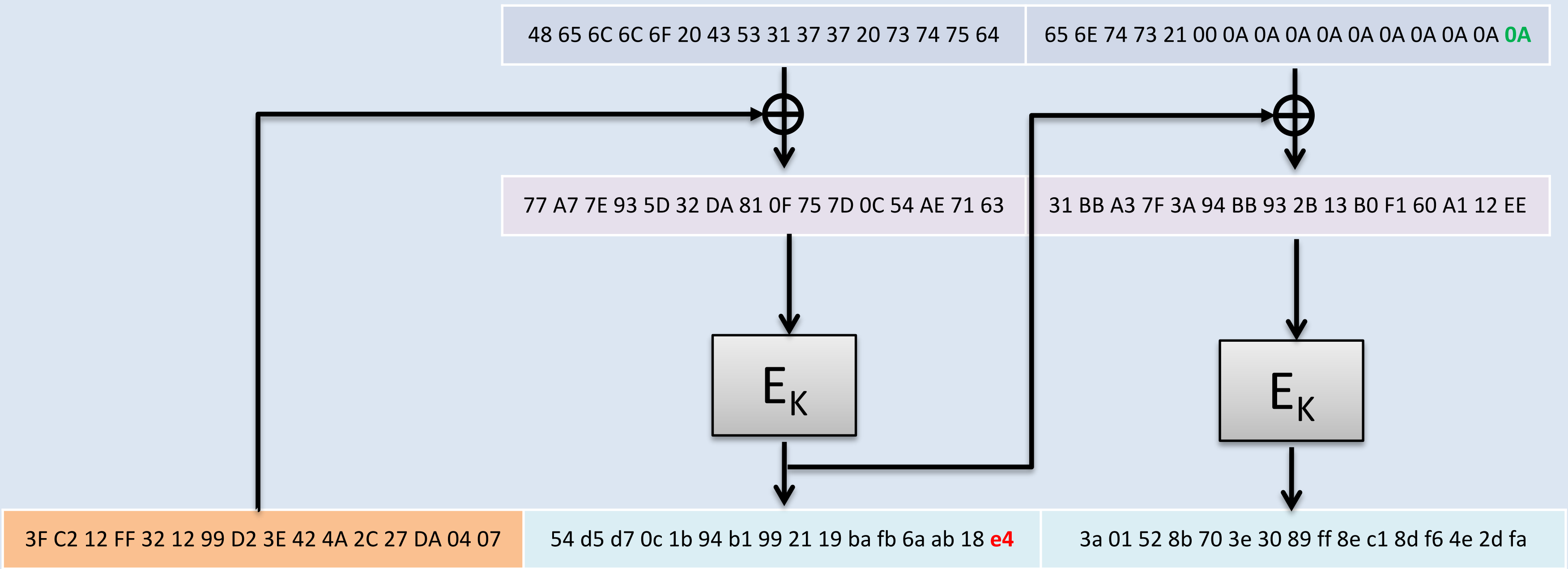


Idea: Modify last 2nd ciphertext-block byte from E4 to E4 xor 0A xor 01 = EF

Last byte of recovered plaintext becomes

EE xor (E4 xor 0A xor 01) = (EE xor E4) xor (0A xor 01) = 0A xor 0A xor 01 = 01

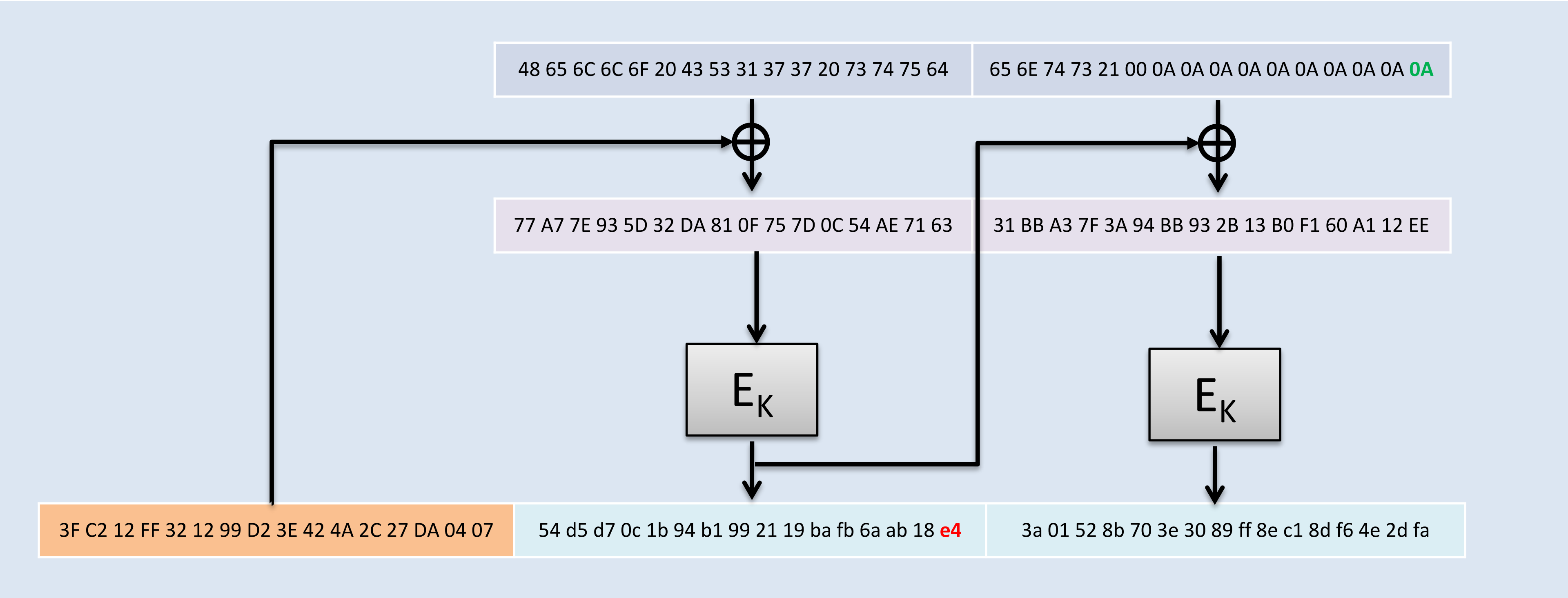
Wait … what have we learnt?



Fact: If the last byte of the (padded) plaintext is **X**, and we modify the last byte **Y** of the second ciphertext block to **X** xor **Y** xor 01, then this makes the last byte **01**, and thus always results in a ciphertext with <u>valid padding</u>

Fact: If we modify the last byte **Y** of the second ciphertext block to **X'** xor **Y** xor 01 for **X'** $\neq$ **X, 01** , then we get a ciphertext whose last byte is different from 01, 0A, thus getting an <u>invalid padding</u>!

The attack



To recover the last byte of the last block: For all possible guesses **X**, change the last byte **Y** of the second-last ciphertext block to **X** xor **Y** xor 01, and submit result ciphertext to padding oracle.

If padding oracle says ciphertext has valid padding, take **X** as the value of the last byte

[Takes at most 255 trials – efficient!]

Caveat: There may be two possible candidate **X** (see before), how do you know which one is the right one? See homework!

In the homework

- How to recover other bytes?
  - Once you know the last byte, it is easy to recover the second-last byte of the last block by extending the above trick! And so on. With <= 256 trials per byte, so complexity of the attack is <u>linear</u> in ciphertext length.
  - Note: It may be that search returns two plausible candidates. In the above case,
    E4 xor 0A xor 01 = EF and E4 xor 01 xor 01 = E4
  - How to decide which one to pick?
- Nothing special about last block: To recover earlier blocks, just throw away later ciphertext blocks

Solutions

- **BAD:** Use counter-mode
  - Counter-mode does not pad
  - However it does not guarantee integrity
  - Some CTR implementation actually do pad …
- **BAD:** Try not to leak padding oracles
  - Harder (timing, other measureable side effects ...)
- **GOOD:** Use authenticated encryption!
  - All ciphertexts submitted by the attacker will be rejected

Bottom line – <u>Always</u> use authenticated encryption!

Many crypto libraries will <u>not</u> force you to do so (e.g., pycrypto!) and you may need to implement it yourself

- Exceptions: OpenSSL (but it’s a mess), GnuPG (libgcrypt), NaCl (by D.J.Bernstein, implements very ad-hoc version)

If AE unavailable, just use CBC/CTR-mode + HMAC

- Requires a few lines of code
- Combines widely supported algorithms
- Don’t try to implement GCM yourself (don’t!), unless you really know what you are doing

A sad reality

Just because a protocol is available, it does not mean it is secure.

- Once a bad protocol is adopted, it has a hard time being replaced.
- Protocols are adopted all the time because these are hard concepts to understand, and engineers make mistakes.