Applied Cryptography
Spring Semester 2023
Lectures 18, 19, 20 and 21

Kenny Paterson (@kennyog)

Applied Cryptography Group

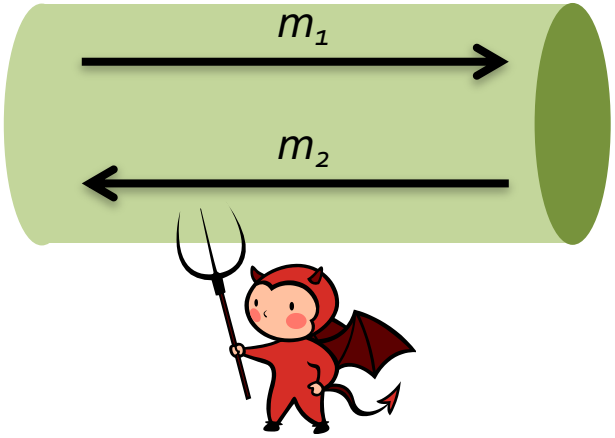
https://appliedcrypto.ethz.ch/

Overview of lectures 18-21

- Motivation for Authenticated Encryption (AE)
- Definitions for AE security
- Generic composition for AE schemes
- Nonce-based AEAD
- Further constructions for AE and AEAD schemes: AES-GCM
- (Extra slides: attacks on CBC-mode in SSL/TLS; the CCM scheme.)

Motivation for Authenticated Encryption

Authenticated Encryption (AE)



Security goals:

Confidentiality and integrity of messages exchanged between Alice and Bob.

Adversarial capabilities:

Adversary can arbitrarily delete, reorder, modify, etc, bits on the wire.

Adversary can mount chosen plaintext and chosen ciphertexts attacks – formalised via encryption and decryption oracles.

Tools we now have:

Encryption (e.g. CBC mode, CTR mode) and MAC algorithms (e.g. HMAC).

Formalising Symmetric Encryption

A symmetric encryption scheme consists of a triple of efficient algorithms: SE = (KGen,Enc,Dec), with the following characteristics:

KGen: key generation, typically selects a key K uniformly at random from a set $\mathcal{K}$; we assume $\mathcal{K} = \{0,1\}^k$.

Enc: encryption, takes as input key K, plaintext $m \in \mathcal{M} \subseteq \{0, 1\}^*$ and produces output $c \in \mathcal{C} \subseteq \{0, 1\}^*$.

Dec: decryption, takes as input key K, ciphertext $c \in \mathcal{C} \subseteq \{0, 1\}*$ and produces output $m \in \mathcal{M}$ or an error, denoted $\perp$.

Correctness: we require that for all keys K, and for all plaintexts m,

$$\mathrm{Dec}_K(\mathrm{Enc}_K(m)) = m.$$

Authenticated Encryption – Informal Definition

A symmetric encryption scheme SE is said to offer Authenticated Encryption security, or be AE-secure if:

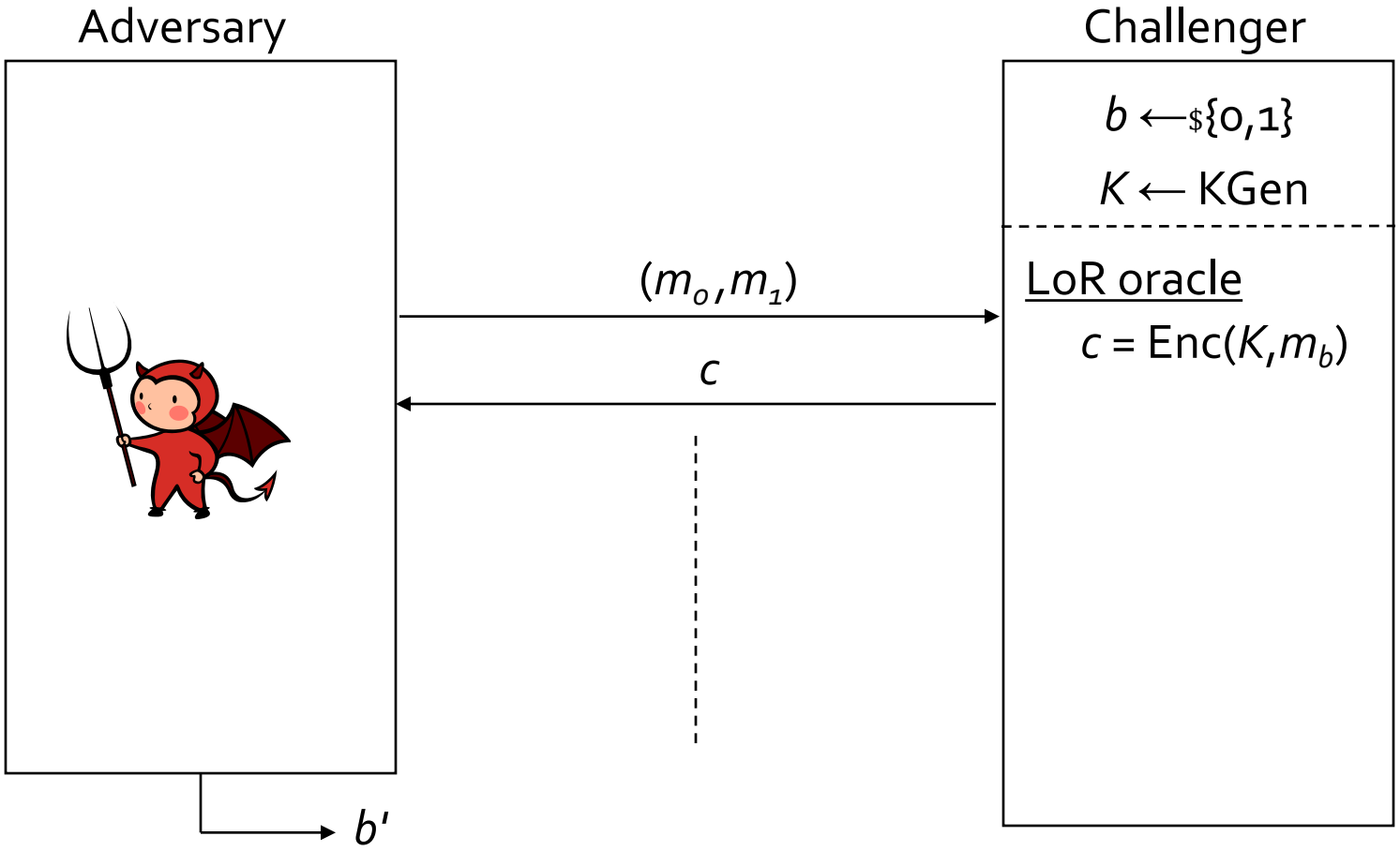
It is IND-CPA secure

AND

An attacker with access to an encryption oracle cannot *forge* any new ciphertexts.

- Motivation for IND-CPA security should by now be clear.
- As should need for some kind of integrity to prevent "bit flipping" attacks like those we've seen on IND-CPA secure modes (CTR, CBC).
- Why do we care about ciphertext forgeries?
- What about chosen ciphertext attacks?

Reminder: IND-CPA security for Symmetric Encryption



$$\text{Adv}^{\text{IND-CPA}}_{\text{SE}}(A) := 2|\Pr[b'=b] - ½|$$

Motivating stronger security

- In ECB, CBC and CTR modes, we have already seen how an active adversary can *manipulate* ciphertexts.
  - For CTR mode, bit flipping in plaintext is trivial by bit flipping in the ciphertext: modify c to c XOR Δ to change the underlying plaintext from p to p XOR Δ .
- Sometimes the attacker can learn information from how these modified ciphertexts are decrypted.
  - In ECB mode, can replay/reorder blocks of ciphertext, leading to same effect on plaintext blocks. This can lead to plaintext recovery, cf. https://mega-awry.io/
  - CBC mode: padding oracle attacks lead to plaintext recovery.
- An attacker can also create completely new ciphertexts from scratch: a random string of bits of the right length is a valid ciphertext for *some* plaintext for ECB, CBC and CTR modes!

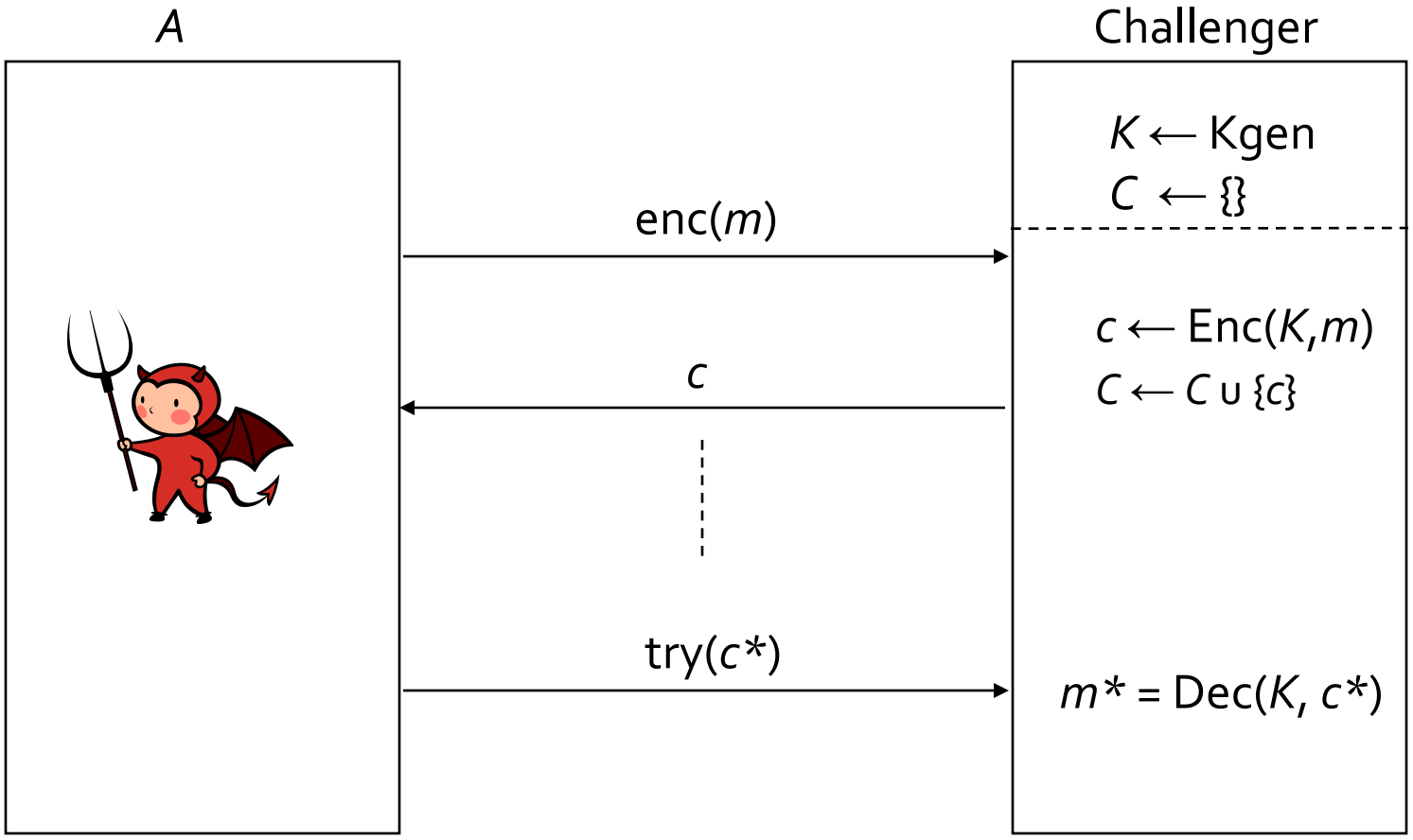
Motivating stronger security

- These kinds of attack do not break IND-CPA security, but are clearly undesirable if we want to build secure channels.
- A modified plaintext may result in wrong message being delivered to an application, or unpredictable behaviour at the receiving application.
- We really want some kind of *non-malleable* encryption, guaranteeing integrity as well as confidentiality.
- Two basic security notions: *integrity of plaintexts* and *integrity of ciphertexts*.

Integrity of Ciphertexts – INT-CTXT

- Adversary *A* has repeated access to an encryption oracle and a **single access** to a "try" oracle.
  - Encryption oracle enc(·) takes any *m* as input, and outputs $\text{Enc}_K(m)$.
  - Try oracle try(·) takes any *c** as input (and has no output).
- Adversary's task is to submit *c** to its try(·) oracle such that:
  1. *c** is distinct from all the ciphertexts *c* output by enc(·) oracle; and
  2. Dec(*K*,*c**) decrypts to a message $m^* \neq \perp$.
- Hence adversary wins if it can create a "ciphertext forgery" – a valid ciphertext that it did not get from its encryption oracle.
- Compare to "no verify oracle" version of SUF-CMA for MACs.

INT-CTXT security in a picture



Adversary wins if $c^* \notin C$ and $m^* \neq \perp$

INT-CTXT security

- We define:

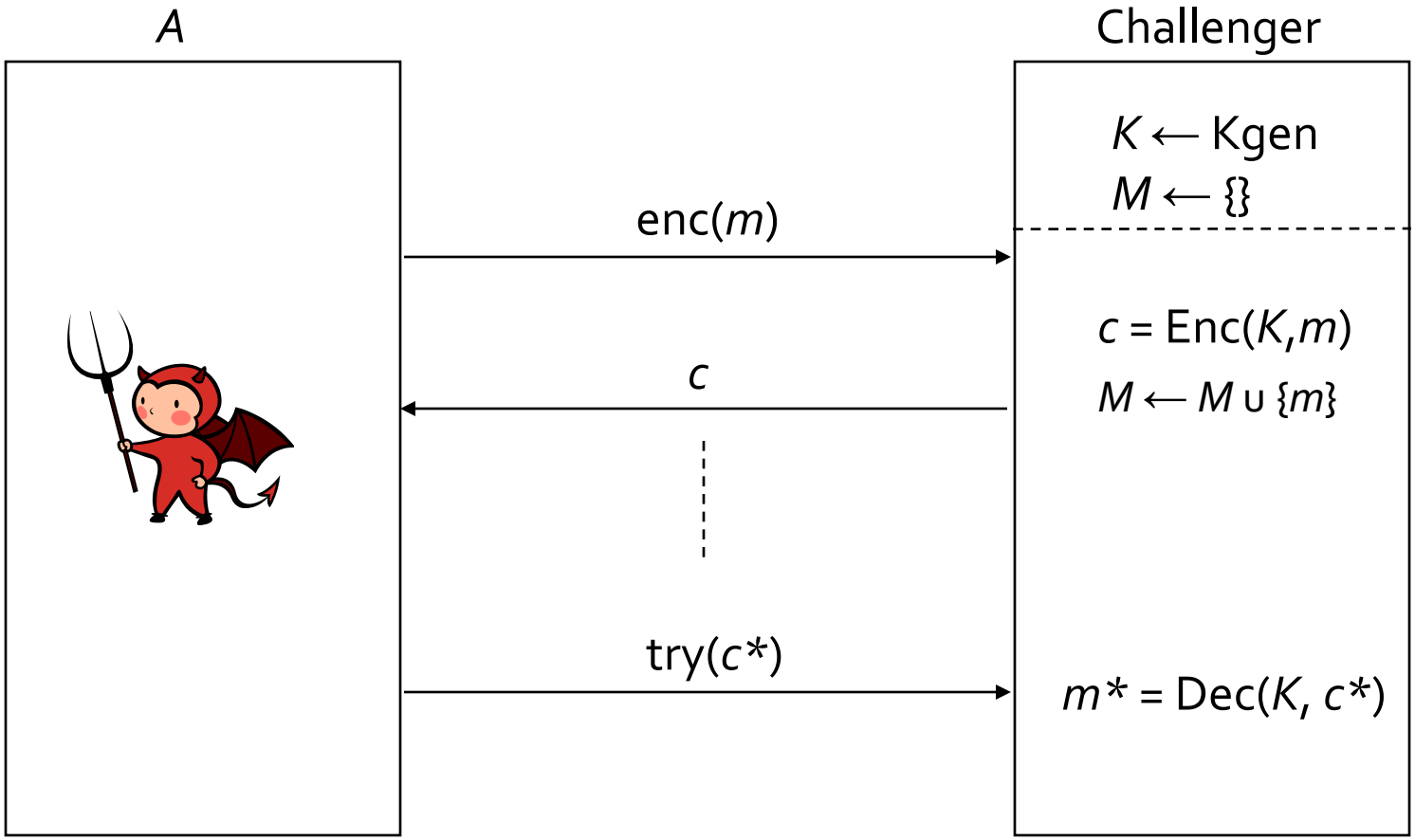
$$\mathrm{Adv}^{\text{INT-CTXT}}_{\mathrm{SE}}(A) := \Pr[A \text{ wins in INT-CTXT game vs. SE}].$$

- A symmetric encryption scheme SE is said to provide INT-CTXT security if the advantage of any adversary using *reasonable* resources is *small.*
- We can quantify in the usual way in terms of q_e, t, ε.
- A symmetric encryption scheme SE is said to be (q_e,t,ε)-INT-CTXT-secure if the advantage of any adversary running in time t and making at most q_e encryption queries is bounded by ε .
- We could also remove the Try(·) oracle and just allow A to output c^* at the end of its game, cf. SUF-CMA security for MACs.
- We can also define a multi-try version of the security notion; it is equivalent to this one up to a factor q_{try} in the advantage.

INT-PTXT security

- INT-PTXT: same as INT-CTXT, but now adversary needs to come up with a ciphertext c^* that decrypts to a message m^* such that m^* was never queried to the encryption oracle.
- Informally, INT-PTXT security means that the adversary can't force a new *plaintext* to be accepted by the receiver.
- Note that an INT-PTXT adversary need not know the value of m^*.
- Formally, a symmetric encryption scheme SE is said to be (q_e, t, ε)-INT-PTXT-secure if the advantage of any adversary running in time t and making at most q_e encryption queries is bounded by ε .

INT-PTXT security in a picture



Adversary wins if $m^* \notin M$ and $m^* \neq \perp$

INT-PTXT security

- If a scheme SE is INT-CTXT secure, then it is also INT-PTXT secure.
  - To show this, we start by assuming scheme SE is not INT-PTXT secure, and show that this implies it is not INT-CTXT secure.
  - Let A be any (q_e, t, ε)-INT-PTXT adversary against SE.
  - Let B be the INT-CTXT adversary that runs A and relays A's queries to its own enc(·) and try(·) oracles.
  - Then B is a (q_e, t, ε)-INT-CTXT adversary against SE!
  - To see why:
    - Suppose A wins, and let $m^* = \mathrm{Dec}(K, c^*)$ where A (and B) makes query try(c^*).
    - Since A wins, we have $m^* \notin M$.
    - Suppose $c^* \in C$. Then c^* would decrypt to some $m \in M$, contradicting $m^* \notin M$ (here we rely on correctness of SE).
    - Hence we can deduce that $c^* \notin C$, and we already know that c^* decrypts correctly (to m^*).
    - So we see that B wins whenever A wins.

Definition of AE Security

AE Security

Recall that a symmetric encryption scheme SE is said to offer Authenticated Encryption security, or be AE-secure if:

It is IND-CPA secure
AND
An attacker with access to an encryption oracle cannot *forge* any new ciphertexts.

i.e.:

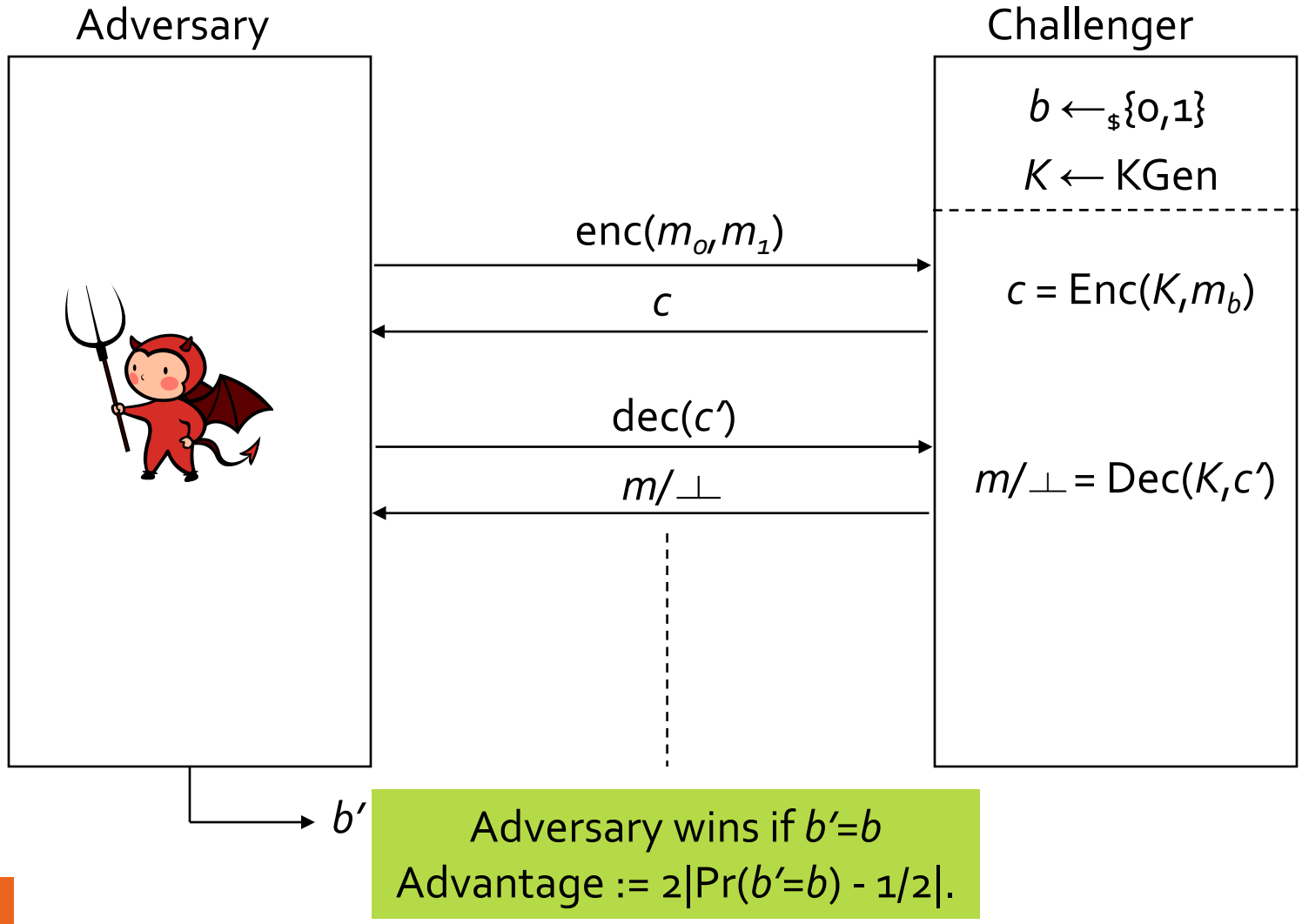
AE := IND-CPA + INT-CTXT

It's common to refer to an SE scheme offering AE security as being an **Authenticated Encryption (AE) scheme**.

What about chosen ciphertext attacks?

- We are also interested in security against chosen ciphertext attacks.
  - Here the adversary has access to both an LoR encryption oracle *and* a decryption oracle.
  - Cannot submit any output from encryption oracle to decryption oracle (otherwise trivial wins for the adversary).
  - Leading to the IND-CCA security notion.
- In practice, the adversary may only have access to something weaker than a full decryption oracle.
  - Partial information about the decryption process might leak, for example, error messages, timing information, etc.
  - cf. padding oracle attacks,…
- But, as usual, we are conservative in our approach:
  - We give the adversary as much power as we can
  - Subject to us being able to efficiently achieve our target security notions.

IND-CCA security in a picture



AE Security implies IND-CCA security

Informal reasoning:

- Suppose we have a successful IND-CCA adversary against an AE-secure scheme SE.
- **Either** at some point this adversary queries the dec(·) oracle on a new and valid ciphertext *c** (not output by the encryption oracle).
  - Then the adversary has broken INT-CTXT security.
  - This is a contradiction, since AE security implies INT-CTXT security.
- **Or** the adversary never queries the dec(·) oracle on a new, valid *c**.
  - This means a challenger can always reply with "$\perp$" to any dec(·) query.
  - So the dec (·) oracle is useless, and the IND-CCA adversary is effectively reduced to being an IND-CPA one.
  - This is also a contradiction, since AE security implies IND-CPA security.

AE Security implies IND-CCA security

Theorem:

Let SE = (KGen, Enc, Dec) be AE-secure (i.e. offering IND-CPA and INT-CTXT security). Then SE is also IND-CCA secure.

More specifically, for any $(q_e, q_d, t, \varepsilon)$-IND-CCA adversary A against SE, there exists a (q_e, t', ε')-IND-CPA adversary B against SE and a $(q_e, t'', \varepsilon'')$-INT-CTXT adversary C against SE such that $t' \approx t'' \approx t$ and:

$$\mathrm{Adv}_{SE}^{\text{IND-CCA}}(A) \leq \mathrm{Adv}_{SE}^{\text{IND-CPA}}(B) + 2q_d \cdot \mathrm{Adv}_{SE}^{\text{INT-CTXT}}(C)$$

(i.e. $\mathrm{Adv}_{SE}^{\text{IND-CCA}}(A) \leq \varepsilon' + 2q_d \cdot \varepsilon''$.)

AE Security implies IND-CCA security

Proof:

Let X be the event that adversary *A* wins the IND-CCA security game against SE (i.e. *b'*=*b* in *A*'s execution).

Let Y be the event that in *A*'s execution in the IND-CCA security game, *A* makes a dec(·) query on some *c** such that Dec(*K*,*c**) ≠ ⊥, and where *c** is not output from any prior encryption oracle query. (We say that *c** is fresh in this case.)

Let Z denote the event that *A* wins the IND-CCA security game against SE but Y does **not** occur, i.e. $Z = X \wedge \neg Y$.

We have:

$$\Pr[X] = \Pr[X \wedge \neg Y] + \Pr[X \wedge Y]$$

$$\leq \Pr[Z] + \Pr[Y].$$

AE Security implies IND-CCA security

Proof (ctd):

We now show that we can construct from A:

1. an IND-CPA adversary B against SE whose winning probability is at least Pr[Z], and
2. an INT-CTXT adversary C against SE whose advantage is at least $\Pr[Y] / q_d$.

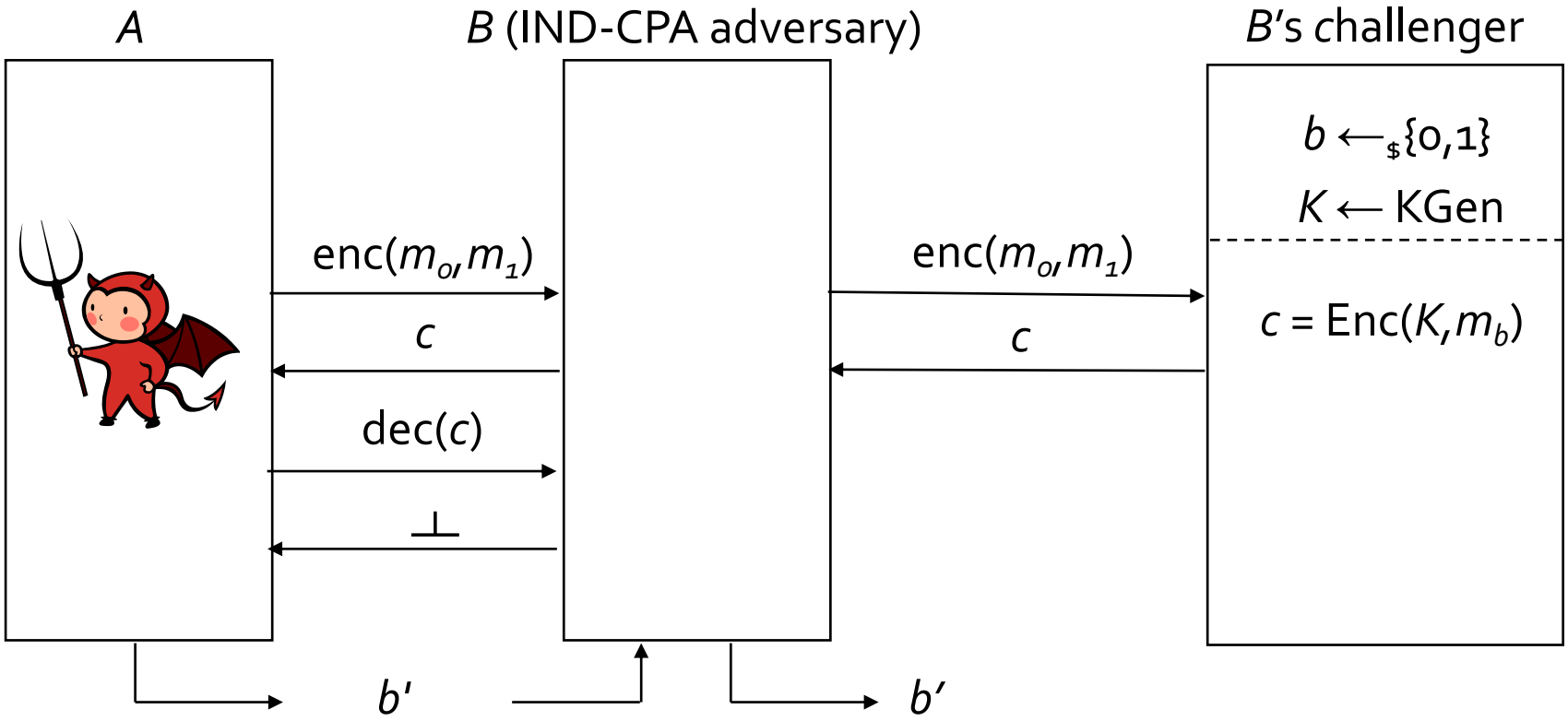
Then to complete the proof, let's assume (wlog) that $\Pr[b'=b] \geq 1/2$ in A's execution. So:

$$0 \leq \Pr[b'=b]-1/2 = \Pr[X] - 1/2 \leq \Pr[Z] + \Pr[Y] - ½$$

Then

$$\begin{aligned} \mathrm{Adv}_{SE}^{\text{IND-CCA}}(A) \leq 2|\Pr[b'=b]-1/2| &\leq 2|\Pr[Z] + \Pr[Y] - ½| \\ &\leq 2|\Pr[Z] - 1/2| + 2|\Pr[Y]| && \text{(triangle inequality)} \\ &= 2|\Pr[Z] - 1/2| + 2\Pr[Y] && \text{(Pr[Y] is non-negative)} \\ &\leq \mathrm{Adv}_{SE}^{\text{IND-CPA}}(B) + 2q_d\,\mathrm{Adv}_{SE}^{\text{INT-CTXT}}(C) \end{aligned}$$

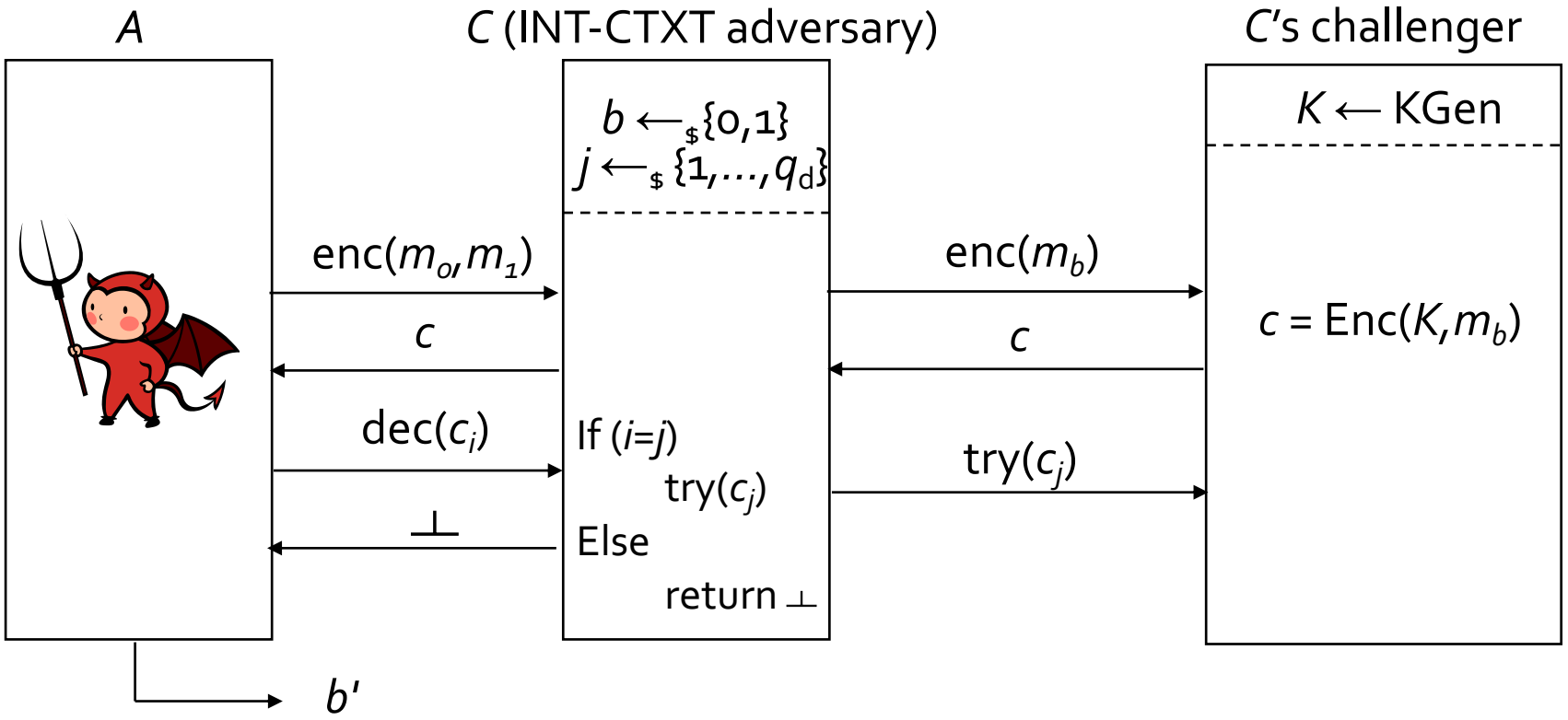
Construction of *B*



Key observations: suppose Z = X ∧ ¬Y occurs.

- Event ¬Y implies *B*'s responses to dec(·) queries are all correct.
- *B*'s responses to enc() queries are correct by construction (for hidden bit *b*).
- Event X implies *A*'s output *b'* is correct, hence so is *B*'s.
- So *B* wins (breaks SE in the IND-CPA sense) whenever Z occurs.

Construction of *C*

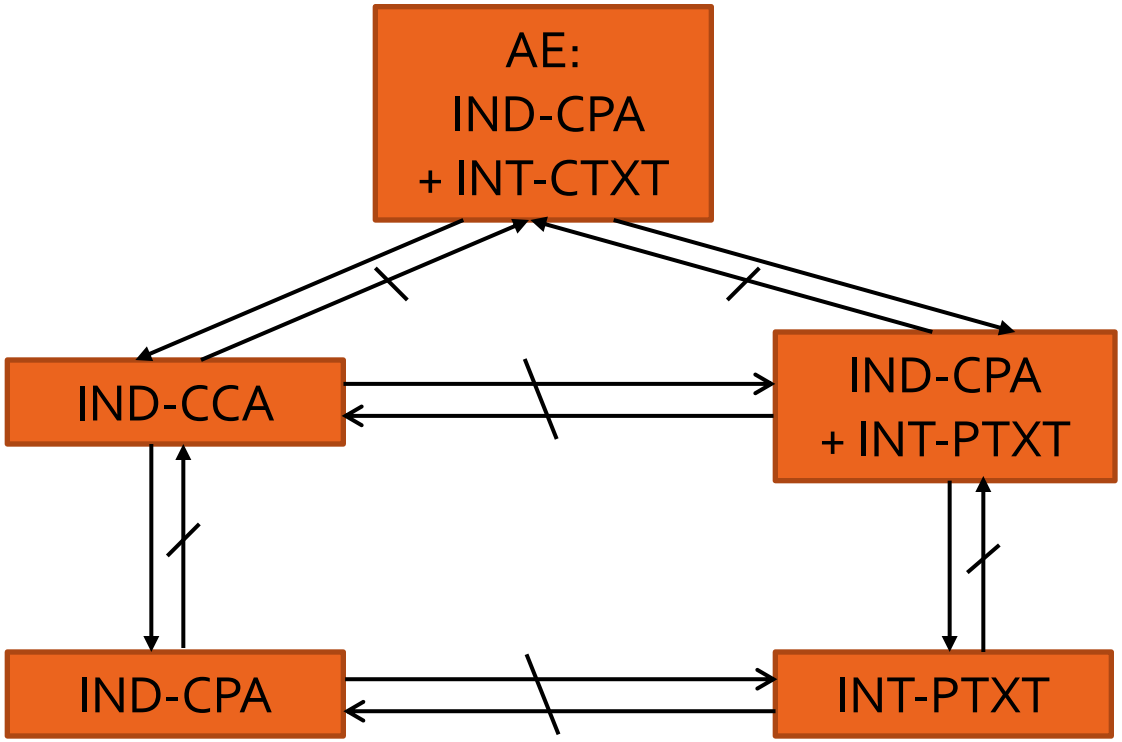


AE Security implies IND-CCA security

Proof (ctd) – analysis of Y

- Suppose event Y occurs.
- *C*'s responses to enc(·) queries are correct (for hidden bit *b* chosen by *C*).
- Event Y implies that at least one of *A*'s q_d dec(·) queries is on a fresh ciphertext that decrypts correctly.
- *Up to the point when this occurs*, *C*'s responses to *A*'s dec(·) queries are all correct (namely, $\perp$).
- There is no guarantee of correct simulation after this point.
- However, the probability that Y occurs in *C*'s simulation for *A* is the same as it is when *A* is playing the actual IND-CCA game.
- If Y occurs, then *C* selects a valid ciphertext to query to its try(·) oracle with probability at least $1/q_d$.
- So *C* breaks SE in the INT-CTXT sense with probability at least $1/q_d$.

Relations between security notions for symmetric encryption



And: IND-CCA + INT-PTXT implies AE!
(Jost et al., https://eprint.iacr.org/2018/135)

AE security and beyond

- Because AE security implies IND-CCA security and INT-PTXT security, AE security has emerged as the natural target security notion for symmetric encryption.
- However it's not the end of the story…
  - In many applications we want to integrity protect some data and provide confidentiality for the remainder – AE with Associated Data, AEAD.
  - We want to work in a nonce-based setting for SE (and AEAD) instead of the randomised setting.
  - AE security does not protect against attacks on secure channels based on reordering or deletion of ciphertexts.
  - The "AE implies IND-CCA" proof only works under the assumption that there is a single possible error message.
  - We want to guarantee some level of security even if nonces are repeated in the nonce-based setting: nonce misuse resistance.
  - We might want to ensure that an AEAD ciphertext is *key-committing.*

AE from generic composition

Generic composition for AE

- We have IND-CPA secure encryption schemes (e.g. CBC mode, CTR mode) and we have SUF-CMA secure MAC schemes (e.g. HMAC).
- Can we combine these to obtain AE security for symmetric encryption?
- Generic options: E&M, MtE, EtM.

- (In what follows, *KM* denotes a MAC key, and *KE* an encryption key.)

Generic composition for AE

Encrypt-and-MAC (E&M)

- To encrypt, compute $c \leftarrow \mathrm{Enc}_{KE}(m)$, $\tau \leftarrow \mathrm{Tag}_{KM}(m)$ and output $c' = c \,||\, \tau$.
- A stateful variant is used in Telegram and was the default in SSH.

MAC-then-Encrypt (MtE)

- To encrypt, compute $\tau \leftarrow \mathrm{Tag}_{KM}(m)$ and output $c = \mathrm{Enc}_{KE}(m \,||\, \tau)$.
- Used in SSL and TLS prior to TLS 1.3.

Encrypt-then-MAC (EtM)

- To encrypt, compute $c \leftarrow \mathrm{Enc}_{KE}(m)$ and $\tau \leftarrow \mathrm{Tag}_{KM}(c)$ and output $c' = c \,||\, \tau$.
- Used in IPsec ESP "enc + auth" and as core of AES-GCM, ChaCha20-Poly1305,…

Security of generic composition for AE: EtM

EtM gives AE security.

- Assuming SE = ($KGEN_{SE}$, Enc, Dec) is IND-CPA secure and MAC = ($KGen_{MAC}$, Tag, Vfy) is SUF-CMA secure.
- Intuition: Applying a MAC to the ciphertext *c* provides ciphertext integrity; IND-CPA security of encryption carries over to the composition $c \| \tau$.
- Sketch proof: Treat IND-CPA and INT-CTXT security notions separately, reduce security of EtM(SE,MAC) to that of either SE or MAC.
- Formal proof: see exercises.
- SUF-CMA-security is essential for the proof (WUF-CMA-security is not enough – why?)
- Plus point: during decryption, we check the MAC on ciphertext, don't even decrypt if it fails; reduced temptation for programmer to "use the plaintext anyway" if MAC fails.
- Implementation pitfall: the MAC needs to include the **whole** ciphertext, including any IV.

Security of generic composition for AE: E&M

To see why E&M fails to be secure in general, consider the natural case when the MAC scheme is instantiated using a PRF F.

- Consider pair of Enc queries (m_0,m_0) and (m_0,m_1) in the IND-CPA security game.
- Ciphertexts are: $c_1 = \text{SE.Enc}(m_0) \,||\, F(K,m_0)$ and then $c_2 = \text{SE.Enc}(m_b) \,||\, F(K,m_b)$.
- If MAC tag changes: must be the case that m_0 and then m_1 was encrypted, so output $b'=1$.
- If MAC tag stays the same, then w.h.p. this is because m_0 was encrypted both times, so output $b'=0$.
  - It's possible but unlikely that $F(K,m_0) = F(K,m_1)$.
- So E&M is not even IND-CPA secure!
- What about INT-CTXT security?
- The analysis here extends to any deterministic MAC scheme.
- This attack can be avoided in the nonce-based and stateful settings.

Security of generic composition for AE: E&M

E&M Implementation pitfall:

During decryption, the natural sequence of operations is:

1. Parse c' as $c \| \tau$.
2. Decrypt c to recover m.
3. Check MAC on m.

- It is tempting to add a step 2.5: sanity check m, e.g. to check a length field.
- This means performing operations on m before it has been integrity checked.
- This can lead to an error side-channel, enabling (partial) plaintext recovery.
- SSH and Telegram both fell into this trap.
  - SSH: https://ieeexplore.ieee.org/document/5207634
  - Telegram: https://mtpsym.github.io/

Security of generic composition for AE: MtE

To see why MtE can fail to be secure is more subtle.

Example

Consider MtE in which the MAC is provided by HMAC and the encryption scheme is provided by CBC-mode using simplified TLS padding.

So we start with an IND-CPA secure encryption scheme and a SUF-CMA secure MAC.

KGen: select at random two keys, *KM*, *KE*.

Encryption:

1. $\tau = \mathrm{Tag}_{KM}(m)$;
2. $p = \text{TLS-PAD}(m \,||\, \tau)$; TLS-PAD: add "00", or "01 01", or "02 02 02", etc.
3. $c \leftarrow\$ \text{CBC-Enc}_{KE}(p)$.

Decryption: ???

Security of MtE generic composition for AE

Decryption:

1. Perform CBC-mode decryption.
2. Check and remove padding – possibility of padding error.
3. Perform MAC verification – possibility of MAC verification error.

If the errors at steps 2 and 3 are distinguishable, then we can carry out a padding oracle attack and recover the plaintext!

- Padding error --> padding was bad.
- MAC verification error --> padding was good.

This attack is a special case of a chosen-ciphertext attack, which would be prevented by AE security (and recall AE security implies IND-CCA security).

Security of MtE generic composition for AE

- We've just seen an example of a scheme constructed from components that are both good (IND-CPA secure encryption scheme, SUF-CMA secure MAC) but for which the MtE composition fails to be secure.
- The example is closely related to the construction that is used in TLS prior to TLS 1.3.
- That construction was subject to a sequence of attacks due to the MtE choice: padding oracles, Lucky 13, POODLE: see extra slides.
- **Specific ways of instantiating MtE can be made secure, but it's unsafe in general and should be avoided wherever possible.**

Summary of generic composition for AE

Encrypt-and-MAC (E&M)

- Compute $c \leftarrow \mathrm{Enc}_{KE}(m)$ and $\tau \leftarrow \mathrm{Tag}_{KM}(m)$ and output $c' = c \,||\, \tau$.
- **INSECURE IN GENERAL, DO NOT USE**

MAC-then-Encrypt (MtE)

- Compute $\tau \leftarrow \mathrm{Tag}_{KM}(m)$ and output $c = \mathrm{Enc}_{KE}(m \,||\, \tau)$.
- **INSECURE IN GENERAL, DO NOT USE**

Encrypt-then-MAC (EtM)

- Compute $c \leftarrow \mathrm{Enc}_{KE}(m)$ and $\tau \leftarrow \mathrm{Tag}_{KM}(c)$ and output $c' = c \,||\, \tau$.
- **SECURE, DO USE, BUT SEE ALSO FOLLOWING TOPICS**

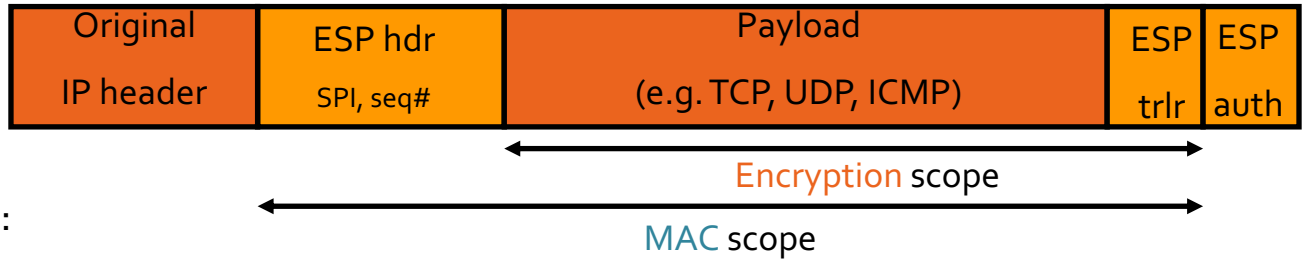
Nonce-based AEAD

Authenticated Encryption with Associated Data (AEAD)

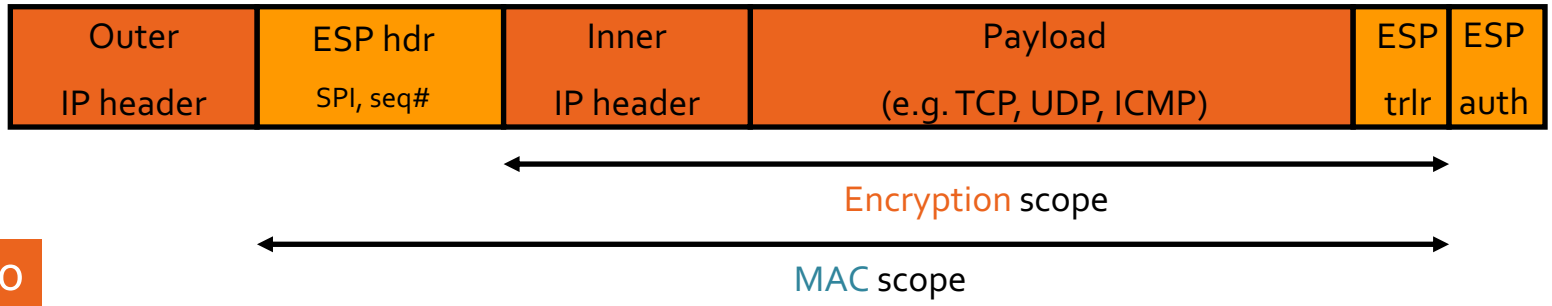
In practical applications, we often require confidentiality and integrity for some data fields and only integrity for others.

Example: ESP in transport and tunnel modes in IPsec, using generic EtM

Transport mode:



Tunnel mode:



Nonce-based AEAD

Nonce-based AEAD = AE with associated data and nonces!

Motivation:

- SE schemes as we have described them so far must consume randomness in Enc algorithm to achieve AE security (IND-CPA security requires randomised encryption).
- Guaranteeing good sources of randomness is hard.
- It is arguably easier to trust the programmer to always pass a fresh nonce value as one of the inputs to the Enc and Dec algorithm.
- For example, may secure communications protocols already use a sequence number (counter) which is incremented for every message sent.
- This makes it an obligation on the calling application to ensure unique nonces.

Nonce-based AEAD

A nonce-based AEAD scheme consists of a triple of algorithms (KGen, Enc, Dec) such that:

KGen: key generation, selects a key K uniformly at random from $\{0,1\}^k$.

Enc: encryption, takes as input key K, **nonce $N \in \mathcal{N}$**, associated data $AD \in \mathcal{AD} \subseteq \{0,1\}^*$, plaintext $m \in \mathcal{M} \subseteq \{0,1\}^*$, and produces output $c \in \mathcal{C} \subseteq \{0,1\}^*$.

Dec: decryption, takes as input key K, **nonce $N \in \mathcal{N}$**, associated data $AD \in \{0,1\}^*$, ciphertext $c \in \{0,1\}^*$, and produces output $m \in \mathcal{M}$ or an error message, denoted $\perp$.

Correctness: we require that for all keys K, for all nonces N, for all associated data strings AD, and for all plaintexts m:

$$\text{Dec}(K, N, AD, \text{Enc}(K, N, AD, m)) = m.$$

Using nonce-based AEAD

Notes:

- For decryption to "undo" encryption, the same value of the associated data *AD* needs to be used. Same for the nonce *N*.
- But the ciphertext *c* does **not** include *AD*; decryption does not recover it.
- Ciphertext *c* also does **not** include the nonce.
- In applications, *AD* and *N* may need to be sent along with *c* or be reconstructed somehow by the decrypting party.
- In applications, sender and receiver typically maintain a synchronized counter to ensure they both use the same *N* when encrypting and decrypting.
- Example: TLS sequence numbers.
- See TLS 1.3 Record Protocol description in Boneh-Shoup, Section 9.8.

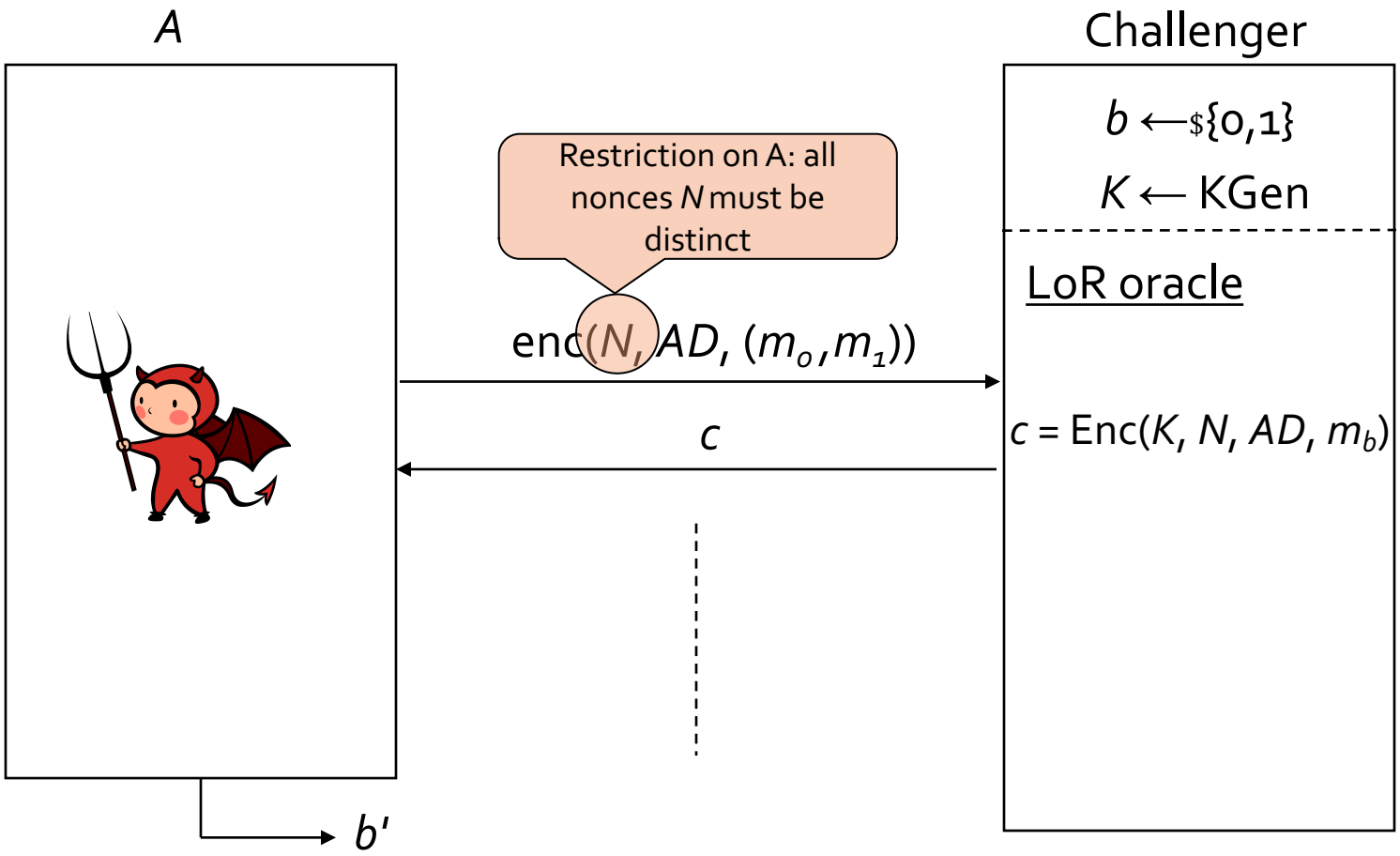
Security for nonce-based AEAD (informal)

- IND-CPA security for messages *m.*
- Integrity for combination of associated data *AD* and ciphertext *c.*
  - *AD* and *c* are both integrity protected and are bound together.
  - So should not be able to "cut and paste" *AD* and *c* together from different contexts.
- For adversaries that never repeat a nonce in their encryption queries (but have no restriction on repeated nonces in try(·) queries).

Security for nonce-based AEAD – IND-CPA

- In the IND-CPA security game, the adversary now gets to specify a pair (m_0,m_1), along with *N* and *AD* in LoR encryption queries.
- Adversary never repeats *N* in its LoR queries (but is otherwise free to set the values of *N*).
  - Formally, we will define AEAD security by quantifying over all adversaries that meet this requirement.
  - Such an adversary is called *nonce-respecting.*
- This very conservatively reflects the idea that an encrypting application must always make sure to use a fresh value of the nonce *N* each time Enc algorithm is called.

IND-CPA security for nonce-based AEAD



$$\mathrm{Adv}^{\mathit{IND\text{-}CPA}}_{SE}(A) := 2|\Pr[b'=b] - 1/2|$$

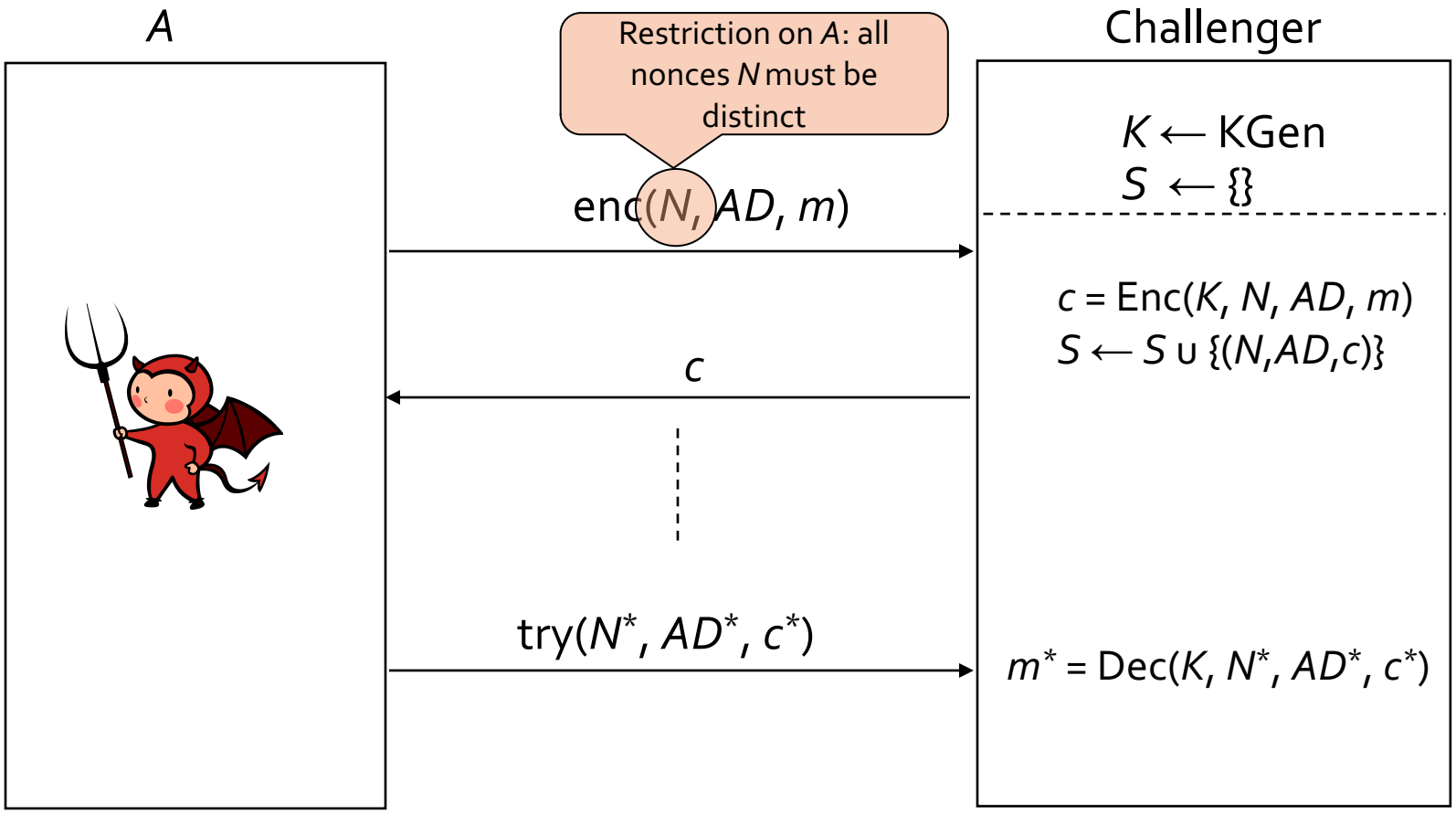
Security for nonce-based AEAD – INT-CTXT

- In the INT-CTXT game for AEAD, adversary now gets to specify m, N and AD in queries to an encryption oracle, and has access to a try(·) oracle.
- Requirement for win is to submit a fresh triple (N^*, AD^*, c^*) to try(·) oracle that decrypts to some $m^* \neq \perp$.
- Freshness of a triple (N^*, AD^*, c^*) means: no response c^* from enc(·) oracle on some input (N^*, AD^*, m^*)
- Adversary can now repeat nonce values in queries to its try(·) oracle (but not in queries to the encryption oracle).
- This reflects the idea that in some AEAD applications, the adversary can set N to any value it pleases when sending data to the decrypting party (including with repeated values).

Security for nonce-based AEAD – INT-CTXT

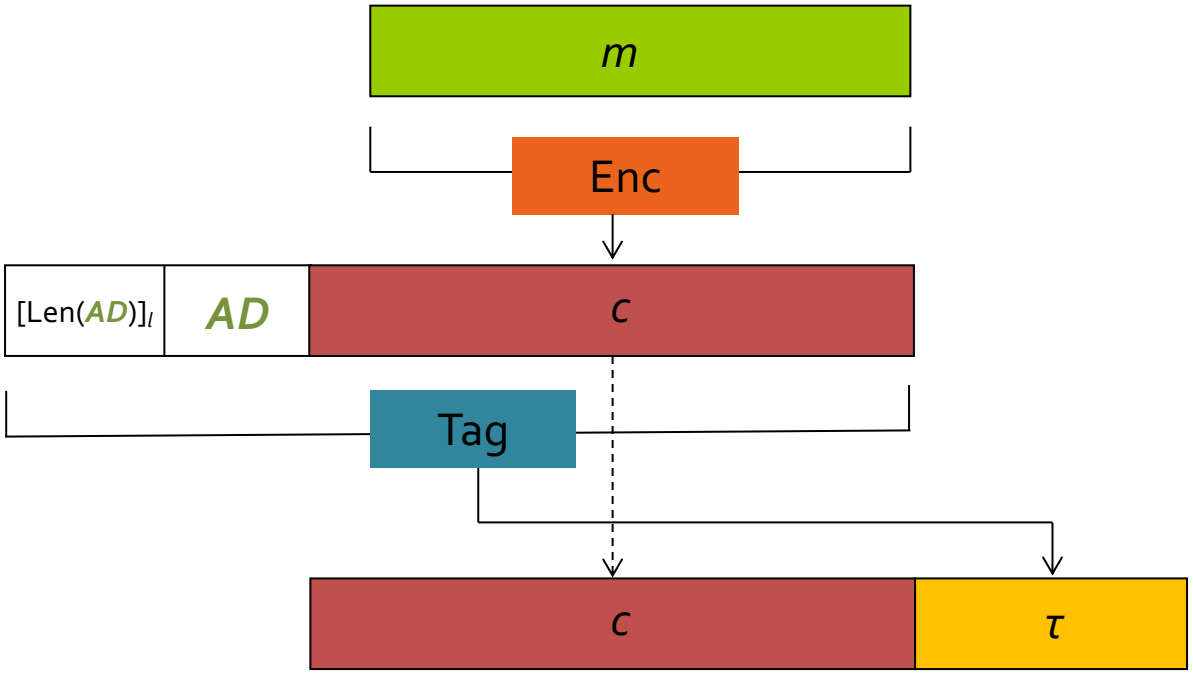
- We will assume that INT-CTXT adversaries make only a single call to the try(·) oracle.
- So the "repeated nonce in try(·) queries" point does not arise in our definitions.
- It can be shown that single and multiple try(·) versions of the INT-CTXT game for nonce-based AEAD are equivalent (up to factor of q_{try}).
  - Similar analysis to that in the SUF-CMA verify/no-verify equivalence.
  - See extra slides at end of Lecture 17 for proof in MAC case.

INT-CTXT security for AEAD



A wins if $(N^*, AD^*, c^*) \notin S$ and $m^* \neq \perp$

Generic construction of nonce-based AEAD from EtM



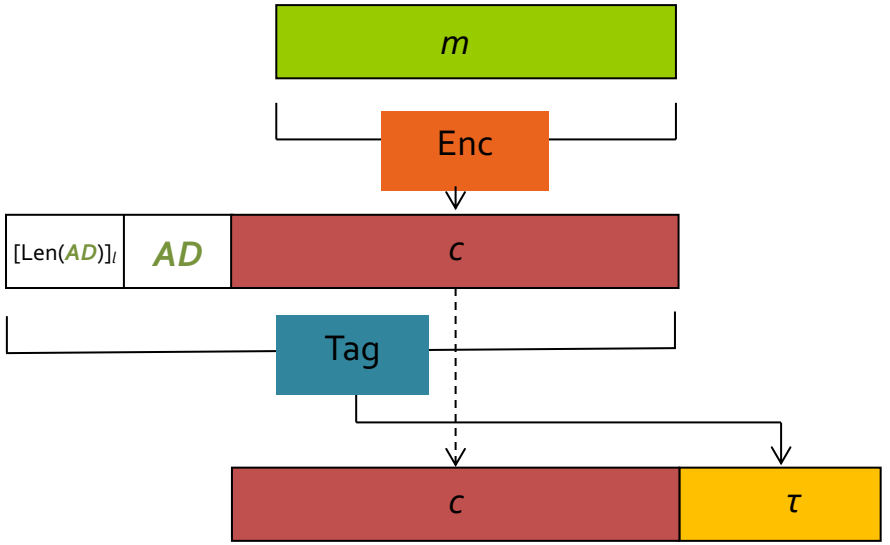
- Requires IND-CPA secure encryption scheme and SUF-CMA MAC scheme, both nonce-based.
- Nonces and keys omitted from diagram for simplicity.
- Encrypt plaintext *m* using key *KE* and nonce value *N*.
- Include ***AD*** in scope of MAC, created using key *KM* and same nonce value *N*.
- Also include fixed length (*l* bits) encoding of length of AD in scope of MAC to prevent mis-parsing attacks (moving part of ***AD*** into *c* or vice-versa).

Informal Security Analysis

- Nonce-based IND-CPA security of the EtM construction follows easily from IND-CPA security of the SE scheme used in the construction.
- **Integrity proof idea**: show that any INT-CTXT adversary *A* against the scheme can be used to create a SUF-CMA adversary *B* against the MAC scheme (in nonce-based setting).
- *B* chooses *KE* and simulates all SE encryptions involved in *A*'s enc(·) queries; *B* uses its own tag(·) oracle to generate the needed MAC tags.
- Three cases to consider:

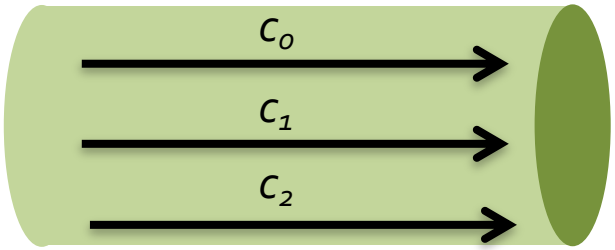
  1. *A*'s output ($N^*, AD^*, c^* || \tau^*$) is valid and has a nonce N^* not used in any of *A's* enc(·) queries: *B* breaks SUF-CMA with output (N^*, p^*, τ^*) where N^* is new and $p^* = [\text{Len}(AD^*)]_l \,||\, AD^* \,||\, c^*$.

  2. *A*'s output is valid, repeats N^* from an enc(·) query, but the string $AD^* || c^*$ arising from that enc(·) query is not repeated: *B* breaks SUF-CMA with output (N^*, p^*, τ^*) where N^* is "old" and "message" $p^* = [\text{Len}(AD^*)]_l \,||\, AD^* \,||\, c^*$ is new.



3. *A*'s output is valid and repeats both N^* and the string $AD^* || c^*$ arising from some enc(·) query: then the only way *A* can win is if $AD^* || c^* = AD_i \,||\, c_i$ for some *i*, but $AD^* \neq AD_i$ (and so also $c^* \neq c_i$). But this implies $[\text{Len}(AD^*)]_l \neq [\text{Len}(AD_i)]_l$; so $p^* = [\text{Len}(AD^*)]_l \,||\, AD^* \,||\, c^*$ is also a new message for *B*, allowing *B* to break SUF-CMA security with output (N^*, p^*, τ^*).

Using nonce-based AEAD to build a basic secure channel



$c_0 = \text{Enc}(K, N{=}0, AD_0, m_0)$

$c_1 = \text{Enc}(K, N{=}1, AD_1, m_1)$

$c_2 = \text{Enc}(K, N{=}2, AD_2, m_2)$

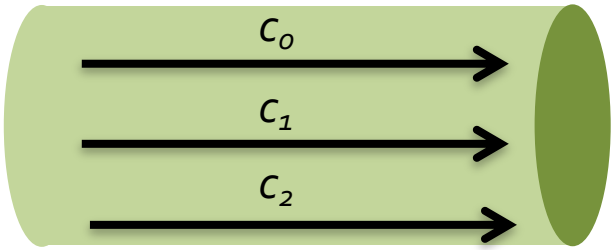
$m_0 = \text{Dec}(K, N{=}0, AD_0, c_0)$

$m_1 = \text{Dec}(K, N{=}1, AD_1, c_1)$

$m_2 = \text{Dec}(K, N{=}2, AD_2, c_2)$

- A sends B a sequence of messages $m_0, m_1, m_2, \ldots$ using nonce-based AEAD.
- A uses an incrementing counter for the nonce values; B uses the same counter values when decrypting.
- For simplicity, we assume *AD* values are not sent on channel, but instead reconstructed at receiver.

Using nonce-based AEAD to build a basic secure channel



$c_0 = \text{Enc}(K, N{=}0, AD_0, m_0)$

$c_1 = \text{Enc}(K, N{=}1, AD_1, m_1)$

$c_2 = \text{Enc}(K, N{=}2, AD_2, m_2)$

$m_0 = \text{Dec}(K, N{=}0, AD_0, c_0)$

$m_1 = \text{Dec}(K, N{=}1, AD_1, c_1)$

$m_2 = \text{Dec}(K, N{=}2, AD_2, c_2)$

- What happens if *A* deletes a ciphertext?
- What happens if *A* reorders the ciphertexts, delivering c_2 before c_1, say?
- In both cases, receiver will use the wrong counter during decryption.
- By INT-CTXT security, decryption will fail and the attack can be detected thanks to the failure.
- So *A* can't achieve undetectable deletion or force a message to be delivered "out of order".
- Nonce-based AEAD integrity also ensures adversary cannot inject new ciphertexts.

Further Constructions

Further constructions

So far we have only seen *generic constructions* for AE and nonce-based AEAD.

- EtM is the only one that is safe to use in general.
- As we've just seen, EtM extends to the nonce-based AEAD setting: encryption is done via:

$$C \leftarrow \mathrm{Enc}_{K_E}(N, M);\ \tau \leftarrow \mathrm{Tag}_{K_M}(N, \mathrm{len}(AD) \,\|\, AD \,\|\, C);\ \text{output } C' = C \,\|\, \tau\ .$$

- We can use the same nonce in both "E" and "M" components.
- Many other AEAD schemes are available; we will look at just one, GCM.
- Some details for CCM in extra slides.

AES-GCM

GCM = Galois Counter Mode

- Basically, an instantiation of nonce-based AEAD using EtM with E = nonce-based CTR mode based on AES, and M = a Carter-Wegman MAC.
- AES-GCM is a single key construction: the key K used in CTR mode is also used to *derive* the MAC key:

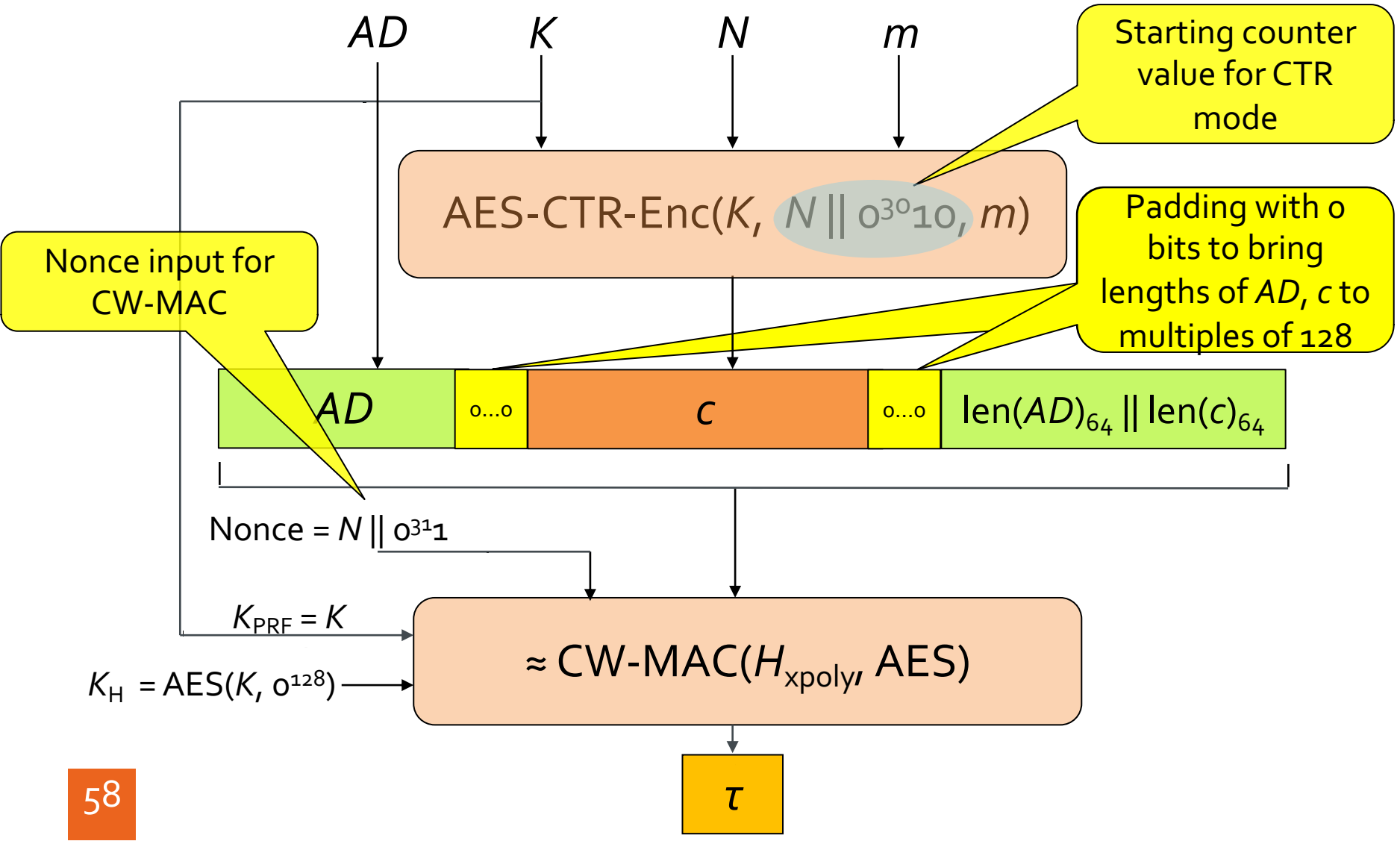
$$K_H = \mathrm{AES}(K, 0^{128}).$$

  - So the scheme has to make sure not to use ctr = 0^{128} anywhere in the CTR-mode encryption.
- MAC is then (essentially) CW-MAC based on the H_{xpoly} ε-DUHF hash over GF(2^{128}) and using AES with key K as the PRF.
  - In reality, H_{poly} is used, with length encodings as part of input.

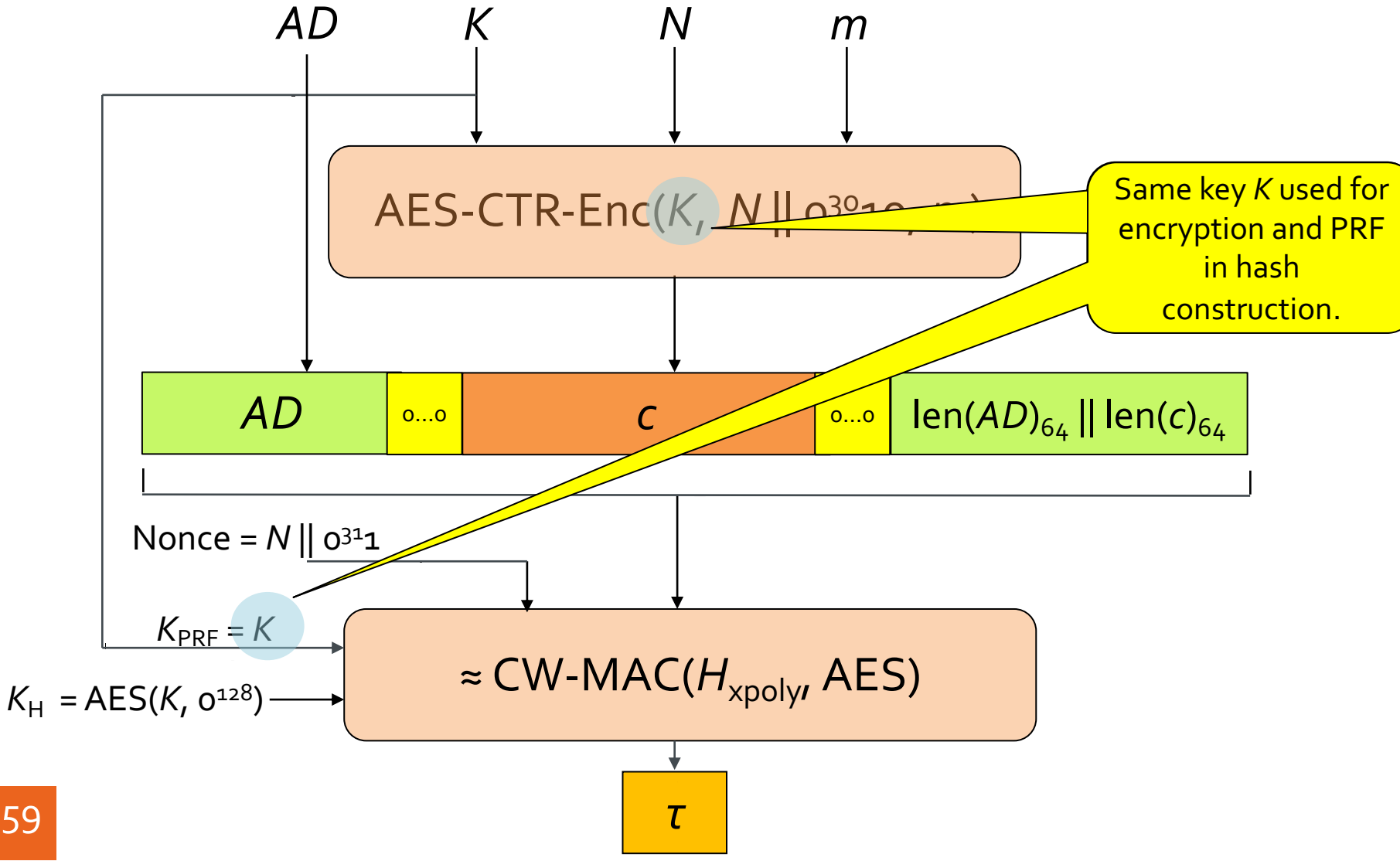
AES-GCM

- Nonces *N* can be of arbitrary length.
- But simpler processing for 96-bit case for speed, and 96-bit nonces are almost universally used in practice.
- Counters used in CTR mode are (roughly) of the form *N* || *ctr* where *ctr* is a 32-bit counter.
- Hence maximum message length is about 2^{32} AES blocks, i.e. 2^{36} bytes.
- Inputs *N* || *ctr* to AES in CTR mode are *separated* from input 0^{128} to AES used to make hash key and from nonce input for CW-MAC.

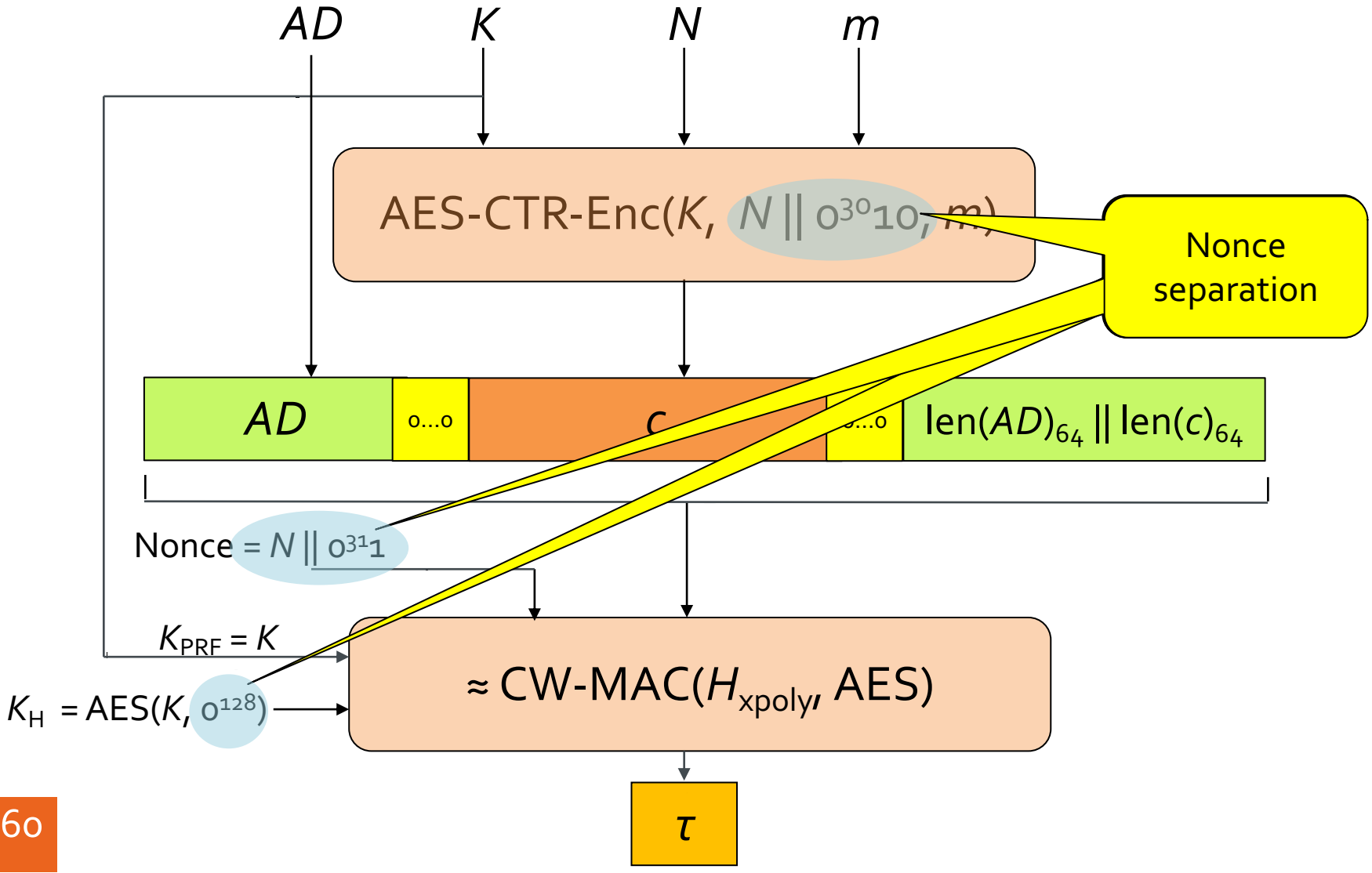
AES-GCM encryption (for 96-bit nonces)



AES-GCM encryption (for 96-bit nonces)



AES-GCM encryption (for 96-bit nonces)



AES-GCM Implementation

- Almost as fast as raw CTR mode with no MAC: speed comes from use of fast MAC algorithm built from universal hash function.
- Special purpose instructions on Intel and AMD CPUs for speeding up field operations underlying the UHF computation.
- AES-GCM only uses block cipher in "forward direction", i.e. only "E" and no "D".
- c and τ can be computed in streaming fashion, no buffering required.
- GCM is patent-free.
- GCM is specified in full in NIST Special Publication SP800-38D (2007).
- GCM is standardised for use in IPsec , TLS 1.2, TLS 1.3 and has become widely used in TLS.

AES-GCM Security

- AES-GCM has a security proof based only on the assumption that AES is a pseudo-random permutation.
- Proof follows from:
  - Generic nonce-based AEAD security of EtM +
  - SUF-CMA security of CW-MAC +
  - IND-CPA security of AES-CTR +
  - Careful domain separation of AES inputs.
- Concrete security bounds are not very good, e.g. TLS 1.3 has a rekeying option to ensure not too much data is encrypted under a single key.
- No known attacks of significance (when implemented properly and when data limits are respected).
- Beware: some allowed parameter sets of GCM have "short" tags, leading to reduced security.

Insecurity of AES-GCM under nonce reuse

- Security of AES-GCM fails very badly if nonces are ever reused with same key K.
- This is a consequence of using a universal MAC scheme.
- In the nonce reuse scenario, an attacker can (w.h.p.) recover the MAC key K_H.
- Nonce reuse also causes CTR mode failure: plaintext and keystream recovery (XOR of ctxts = XOR of ptxts).
- If attacker has the CTR mode keystream for a given nonce, plus the MAC key, it can forge arbitrary packets for that nonce.
- Just do "CTR mode" encryption using the known keystream, and forge MAC using the known MAC key!
- So AES-GCM is **not** nonce-misuse resistant.
- See Böck *et al.: Nonce-Disrespecting Adversaries: Practical Forgery Attacks on GCM in TLS.* https://eprint.iacr.org/2016/475.pdf
- NB this does not violate nonce-based AEAD security definitions because of restrictions on nonces in adversary's enc(·) queries.

Other things you should know about AE(AD)

- Other nonce-based AEAD schemes are seeing growing adoption, e.g. ChaCha20-Poly1305, OCB.
- ChaCha20-Poly1305 is the default algorithm in all versions of OpenSSH since version 6.9 (2015) and is available as an option in TLS 1.3.
- The SHA-3 winner KECCAK can be adapted to produce an AEAD scheme.
- The CAESAR competition recently completed and generated lots of new research activity:
    - https://competitions.cr.yp.to/caesar-submissions.html
    - See also the AE zoo: https://aezoo.compute.dtu.dk/doku.php
- Mis-use resistant AEAD: AEAD that leaks as little as possible if nonces are reused in error – e.g. AES-GCM-SIV in RFC 8452, AEGIS-128 and OCB in CAESAR final portfolio.
- Multi-user/multi-key security: bounds for AES-GCM, ChaCh20-Poly1305.
    - See for example https://eprint.iacr.org/2023/085

Homework

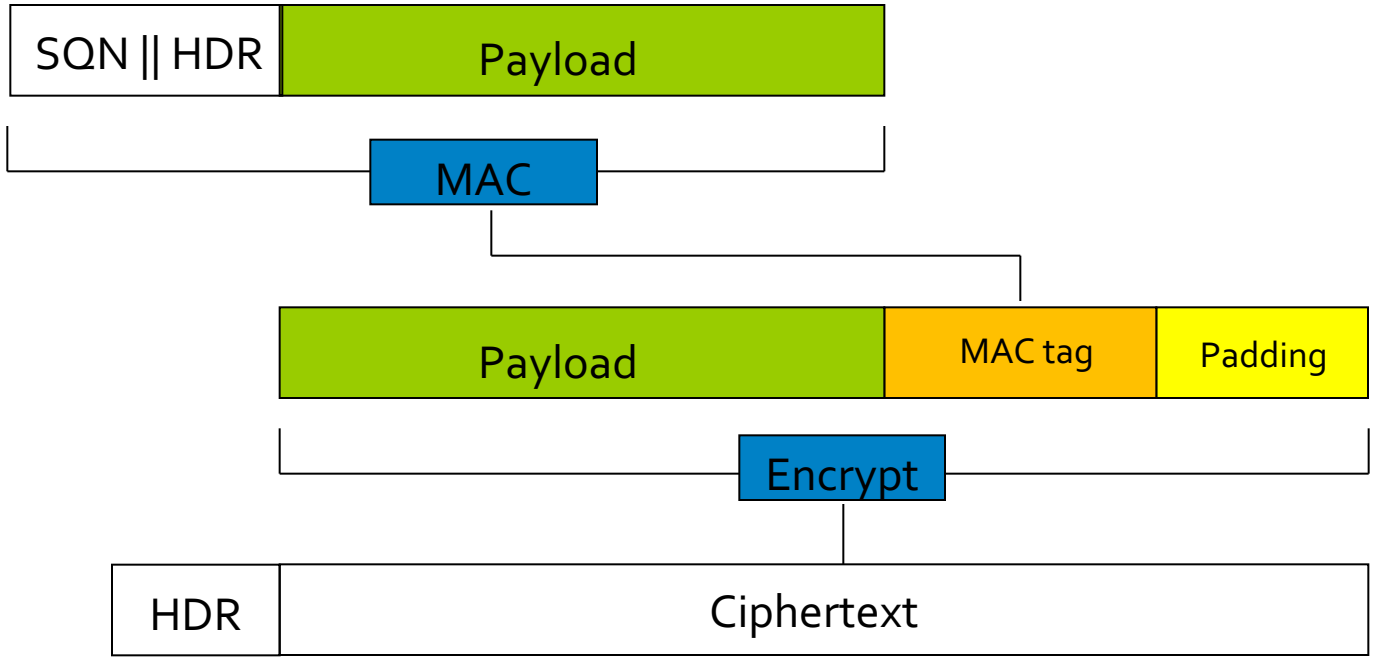
- **Action 1:** start working on exercise sheet.
- **Action 2: r**ead Boneh-Shoup, Chapter 9.

- The mid-term is coming!

Extra slides

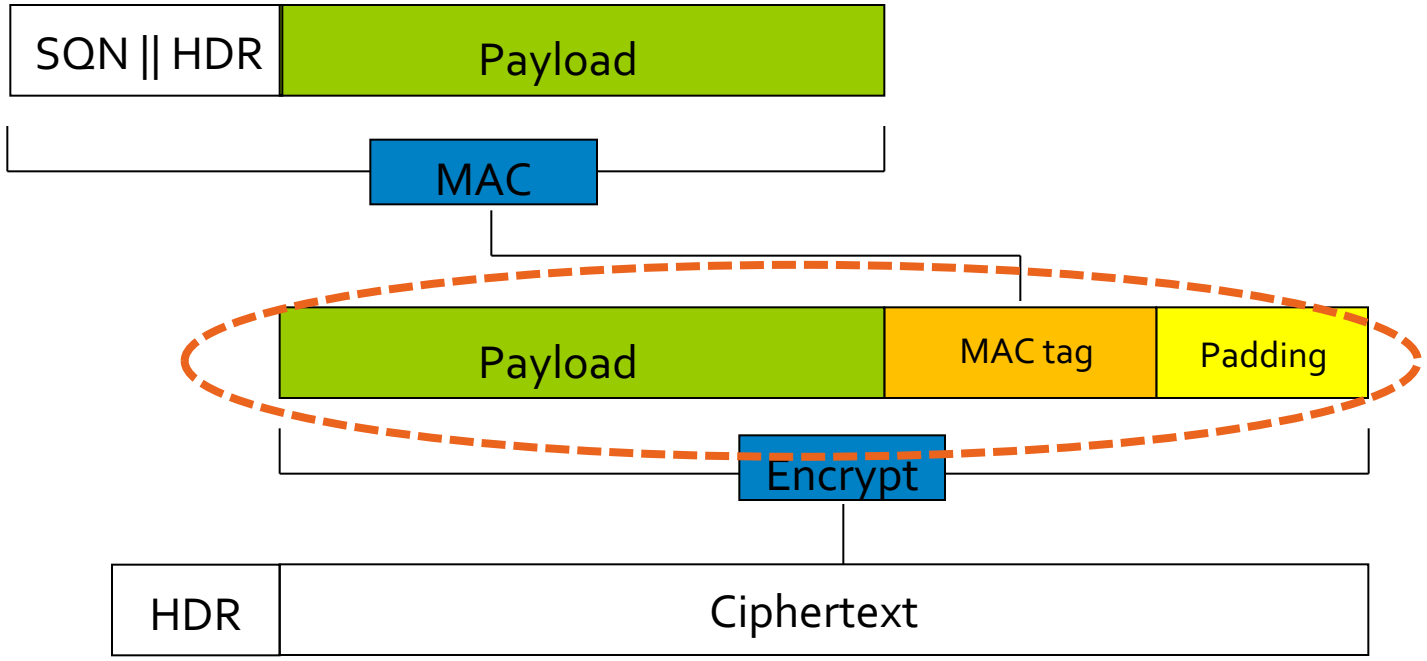
Extra slides: Breaking MtE in SSL/TLS

TLS Record Protocol: MAC-Encode-Encrypt



MAC	HMAC-MD5, HMAC-SHA1, HMAC-SHA256
Encrypt	CBC-AES128, CBC-AES256, CBC-3DES, RC4-128
Padding	"0x00" or "0x01 0x01" or …. or "0xFF 0xFF….0xFF"

TLS Record Protocol: MAC-Encode-Encrypt



Problem is: how to parse plaintext as payload, padding and MAC fields when the padding is not one of the expected patterns 0x00, 0x01 0x01,… ?

TLS Record Protocol padding

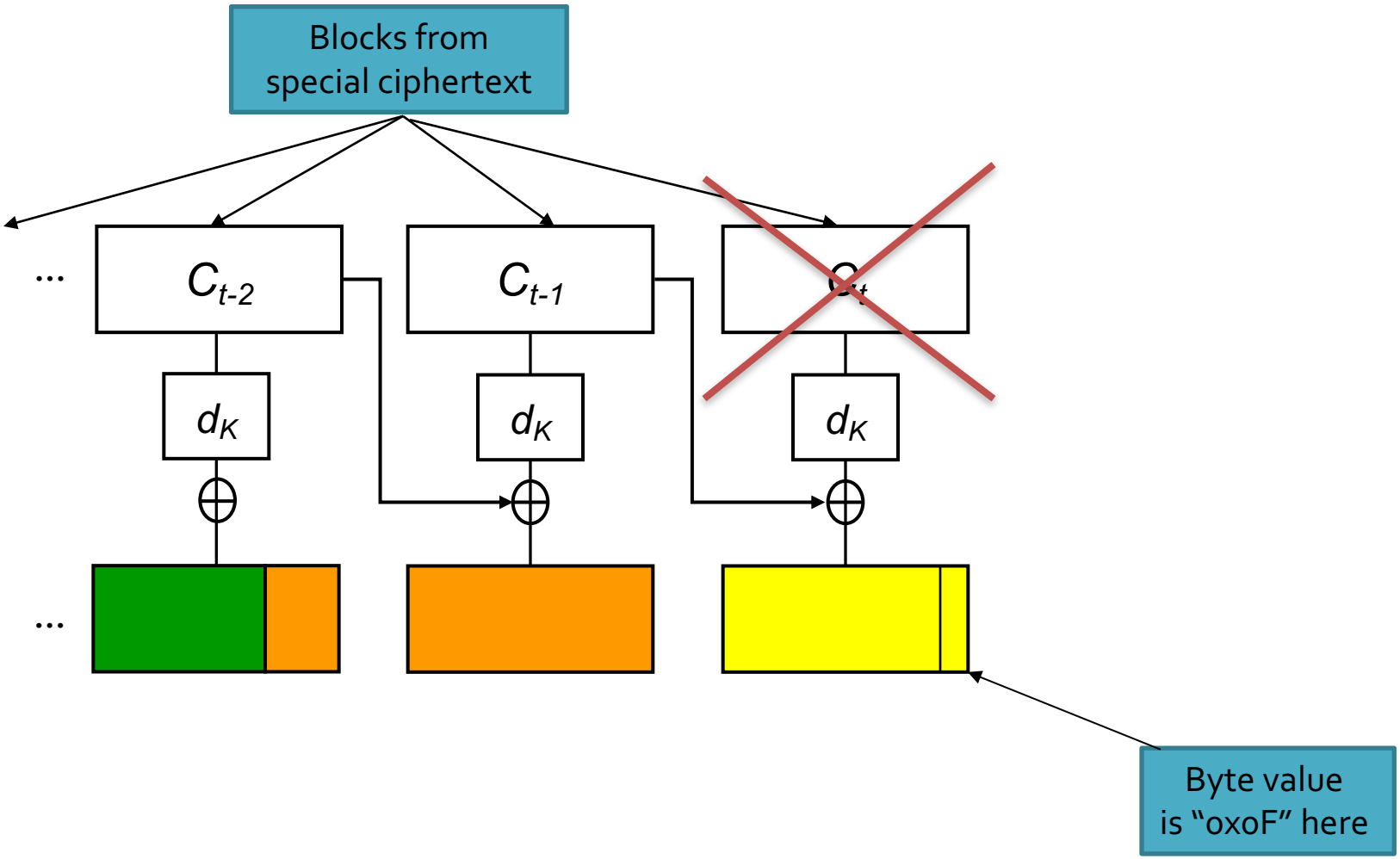
- No padding check (SSLv3) → Moeller's attack (2004) → POODLE.

- Padding check → long series of *padding oracle* attacks, starting with Vaudenay's work in 2002 and ending with Lucky 13, Lucky Microseconds, etc.

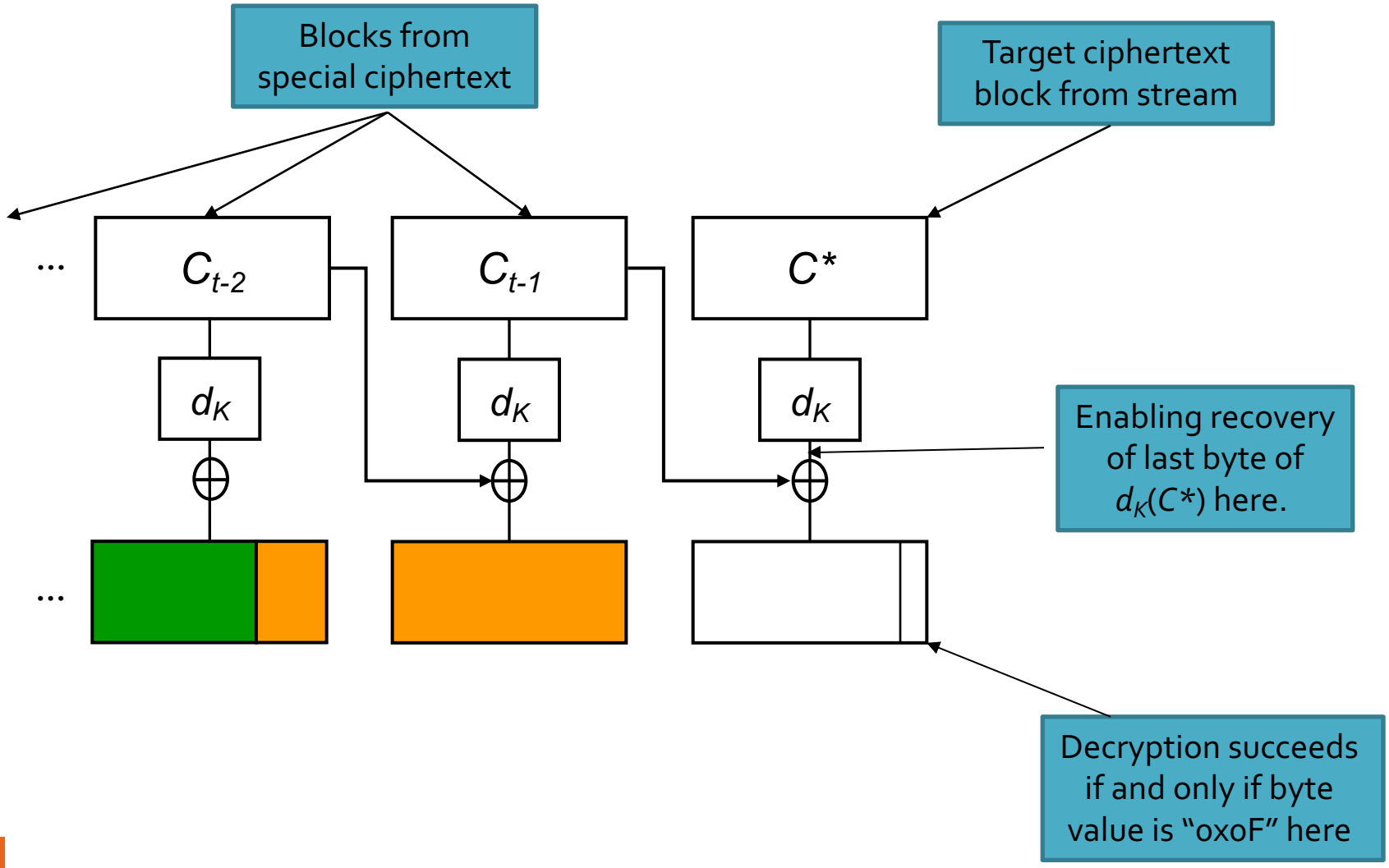
Exploiting weak padding checks – Moeller attack

- Recall SSL/TLS decryption sequence:
    - CBC mode decrypt, remove padding, check MAC.
- SSL padding: add *t*-1 bytes and then byte value *t*-1.
- [Moeller, 2002 & 2004]: inability to check padding format leads to a simple attack recovering the last byte of plaintext from any block.
- Assumptions for Moeller's attack:
    - Attacker has a special TLS ciphertext containing a complete block of padding.
    - So MAC ends on block boundary for this ciphertext.
    - Padding is removed by inspecting last byte only and removing that many bytes plus 1.

Moeller attack



Moeller attack



Moeller attack

- Decryption succeeds if and only if:

$$C_{t-1} \oplus d_K(C^*) = (??,??,....??, \text{oxoF})$$

- Failure of this condition is indicated by a MAC error (visible on network).
- Hence attacker can recover last byte of $d_K(C^*)$ with probability 1/256.
- This enables recovery of last byte of original plaintext P^* corresponding to C^* in the CBC stream, by solving system of eqns:

$$C_{t-1} \oplus d_K(C^*) = (??,??,....??,\text{oxoF})$$

$$C^*_{-1} \oplus d_K(C^*) = P^*$$

where C^*_{-1} is the block preceding C^* in the stream.

- Hence, in TLS 1.1 and up:

Each uint8 in the padding data vector MUST be filled with the padding length value. The receiver MUST check this padding….

SECURITY

Truly scary SSL 3.0 vuln to be revealed soon: sources

So worrying, no one's breathing a word until patch is out

By Darren Pauli, 14 Oct 2014

Gird your loins, sysadmins: *The Register* has learned that news of yet another major security vulnerability - this time in SSL 3.0 - is probably imminent.

Maintainers have kept quiet about the vulnerability in the lead-up to a patch release expected in in the late European evening, or not far from high noon Pacific Time.

Details of the problem are under wraps due to the severity of the vulnerability.

RELATED STORIES

OpenVPN open to pre-auth Bash

POODLE = BEAST techniques + Moeller attack

Repeat

1. Use Javascript in the browser to pad HTTP GET requests (as in BEAST), ensuring that the target cookie byte is placed as last byte of block and that the MAC field aligns on a block boundary.
2. Do Moeller's attack with that block to recover the cookie byte with probability 1/256.

Until (all cookie bytes are recovered).

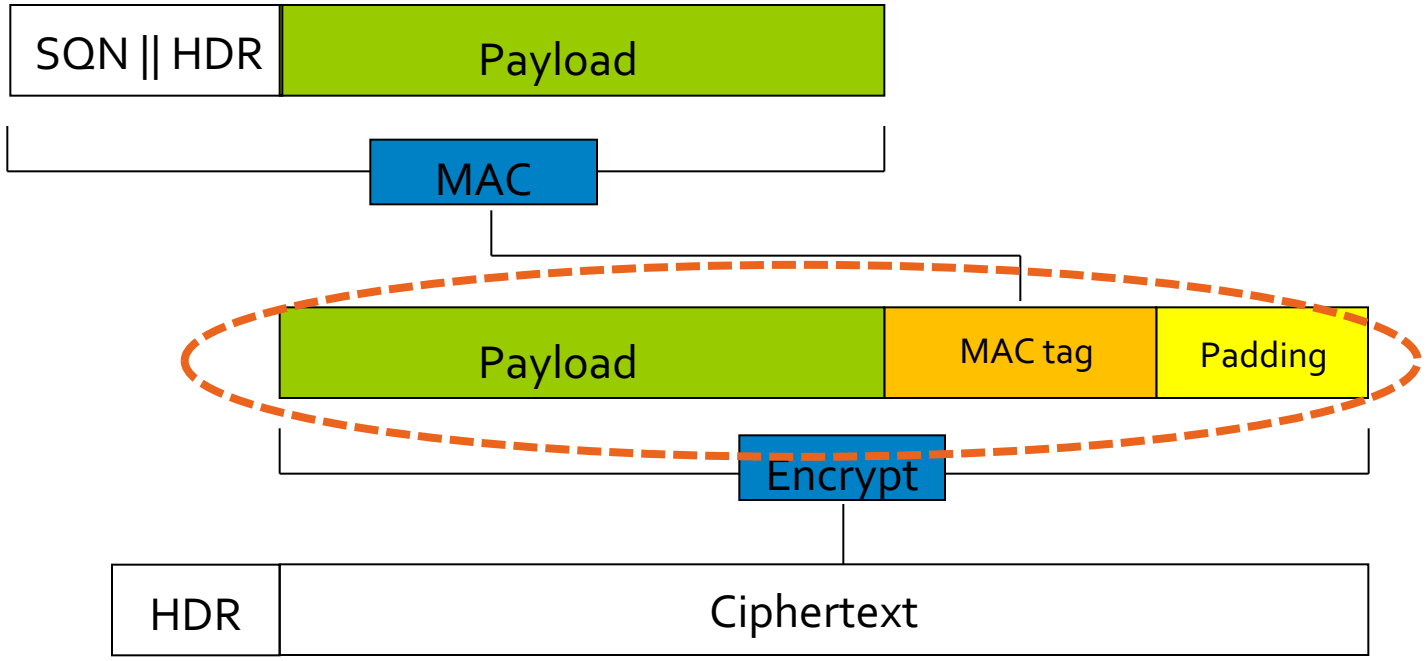
- Details at: https://www.openssl.org/~bodo/ssl-poodle.pdf

POODLE and SSL/TLS fallback

The attack is made worse because of an active version downgrade attack against SSL/TLS, aka SSL/TLS fallback.

- An active MITM attacker could always force client and server to downgrade to SSLv3.
- Because the SSL/TLS version negotiation process is not stateful and was not cryptographically protected across fallbacks.
- RC4 in SSL was also broken by this time (Autumn 2014).
- No more cipher suites left for SSLv3!
- Various countermeasures now exist, but the safest option was to stop supporting SSLv3 in browsers and servers.

Reminder: TLS Record Protocol: MAC-Encode-Encrypt



Padding check in TLS

- We now suppose that TLS does a *full* padding check.
- So decryption checks that bytes at the end of the plaintext have one of the following formats:

00;

01, 01;

02, 02, 02;

....

FF, FF,.....FF;

and outputs an error if *none* of these formats is found.

- NB Other "sanity" checks may also be needed during decryption.

TLS padding oracles in practice?

- In TLS, an error message during decryption can arise from either a failure of the padding check or a MAC failure.
- A padding oracle attack will produce an error of one type or the other.
  - Padding failure indicates *incorrect* padding.
  - MAC failure indicates *correct* padding.
- If these errors are *distinguishable*, then a padding oracle attack should be possible.

TLS padding oracles in practice?

Good news (for the attacker):

- The error messages arising in TLS 1.0 *are* different:
  - `bad_record_mac`
  - `decryption_failed`

Bad news (for the attacker):

- The error messages are encrypted, so cannot be seen by the attacker.
- And an error of either type is *fatal*, leading to immediate termination of the TLS session.

TLS padding oracles in practice?

Canvel *et al.* (Crypto 2003):

- A MAC failure error message will appear on the network **later** than a padding failure error message.
- Because an implementation would only bother to check the MAC if the padding is good.
- So *timing* the appearance of error messages might give us the required padding oracle.
  - Even if the error messages are encrypted!
  - Amplify the timing difference by using long messages – slow MAC verification.
- But the errors are fatal, so it seems the attacker can still only learn one byte of plaintext, and then with probability only 1/256.

OpenSSL and padding oracles

Canvel *et al.* (Crypto 2003):

- The attacker can still decrypt reliably if a *fixed* plaintext is repeated in a *fixed* location across many TLS sessions.
  - e.g. password in login protocol or an HTTP session cookie.
  - Modern viewpoint: use BEAST-style Javascript in the browser to generate the required encryptions.
- The OpenSSL implementation had a detectable timing difference.
  - Roughly 2ms difference for long messages (close to 2^{14} byte maximum).
  - Enabling recovery of TLS-protected Outlook passwords in about 3 hours.

Padding oracle attack countermeasures?

- Redesign TLS:
  - Pad-MAC-Encrypt or Pad-Encrypt-MAC.
  - Too invasive, did not happen, but see RFC 7366.
- Switch to RC4?
- Or add a fix to ensure uniform errors:
  - Check the MAC anyway, even if the padding is bad.
  - If attacker can't tell difference between MAC and pad errors, then maybe TLS's MEE construction is secure?
  - Fix included in TLS 1.1 and 1.2 specifications.

Padding oracle countermeasures, revisited

From the TLS 1.1 and 1.2 specifications (RFCs 4346 and 5246):

…implementations MUST ensure that record processing time is essentially the same whether or not the padding is correct.

In general, the best way to do this is to compute the MAC even if the padding is incorrect, and only then reject the packet.

Compute the MAC on what though?

Issue here is that if padding is bad, we don't know where the MAC tag is located in the byte sequence making up the plaintext!

Ensuring uniform errors

From the TLS 1.1 and 1.2 specifications (RFCs 4346 and 5246):

For instance, if the pad appears to be incorrect, the implementation might assume a zero-length pad and then compute the MAC.

- This approach was adopted in many implementations, including OpenSSL, NSS (Chrome, Firefox), BouncyCastle, OpenJDK, …

Ensuring uniform errors

… This leaves a small timing channel, since MAC performance depends to some extent on the size of the data fragment, but it is not believed to be large enough to be exploitable, due to the large block size of existing MACs and the small size of the timing signal.

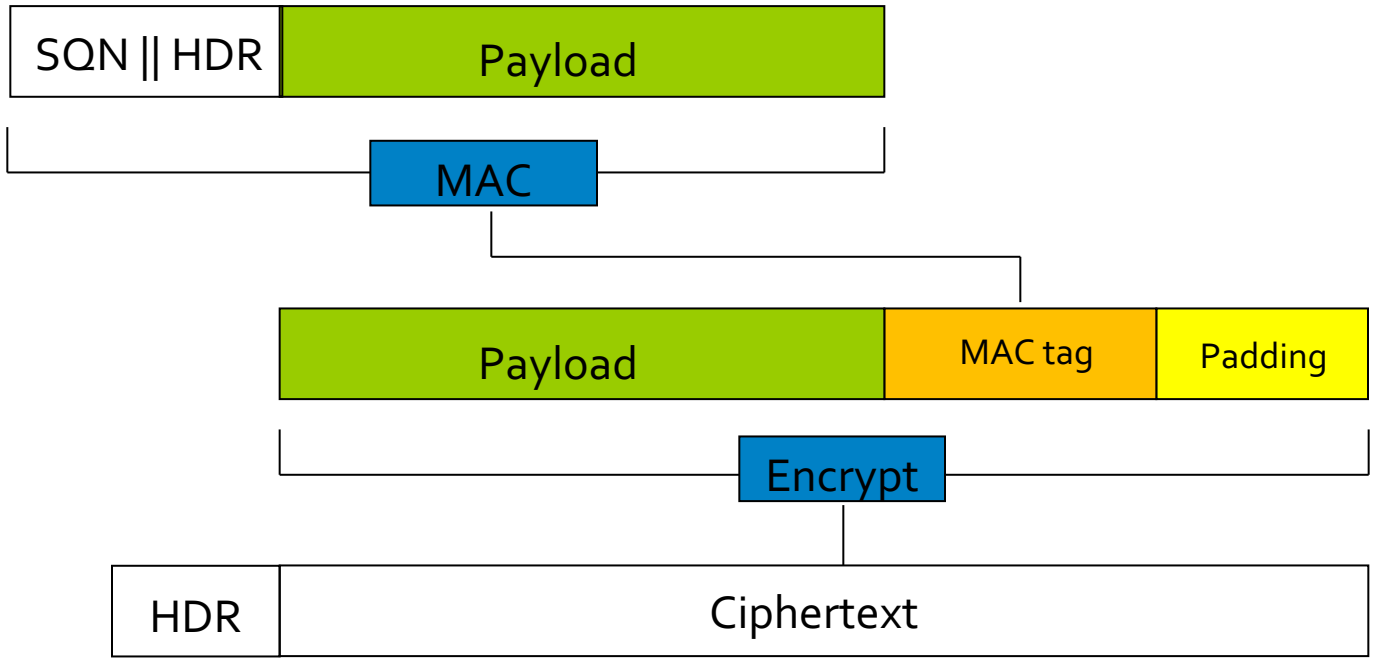
Ensuring uniform errors

… This leaves a small timing channel, since MAC performance depends to some extent on the size of the data fragment, but it is not believed to be large enough to be exploitable, due to the large block size of existing MACs and the small size of the timing signal.

Lucky 13 (AlFardan-Paterson, 2013)

- Full plaintext recovery attack against TLS-CBC implementations following the advice in the TLS 1.1 and 1.2 specs.
- Variants apply to all versions of SSL and TLS (including SSLv3 – cf POODLE).
- Demonstrated in the lab against OpenSSL and GnuTLS.
- Affected 50% of all TLS traffic at the time of publication.
- Paper: Nadhem J. AlFardan, Kenneth G. Paterson: *Lucky Thirteen: Breaking the TLS and DTLS Record Protocols*. IEEE Symposium on Security and Privacy 2013:
- Details at: www.isg.rhul.ac.uk/tls/Lucky13.html

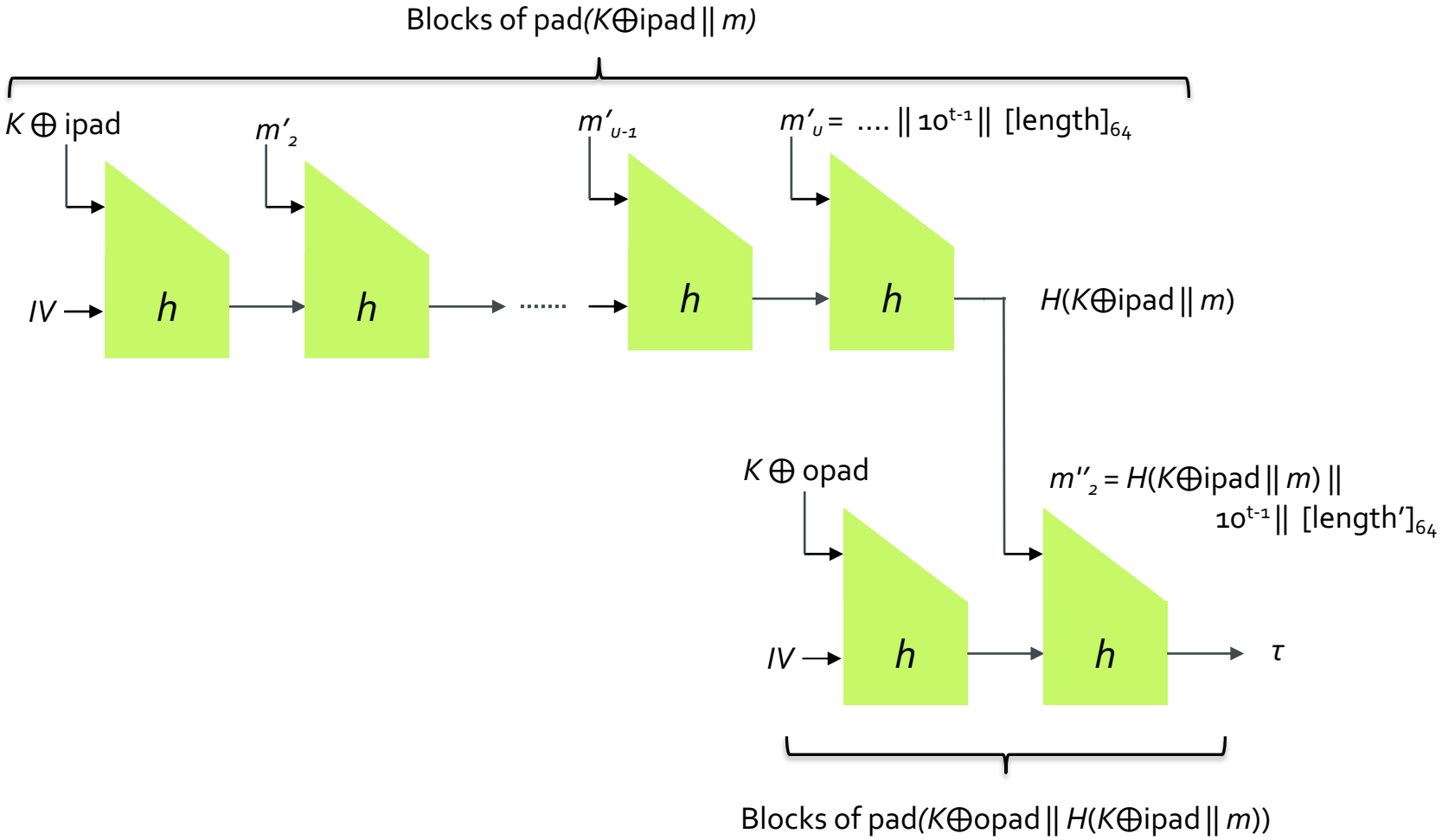
Recall: TLS Record Protocol: MAC-Encode-Encrypt



MAC

HMAC-MD5, HMAC-SHA1, HMAC-SHA256

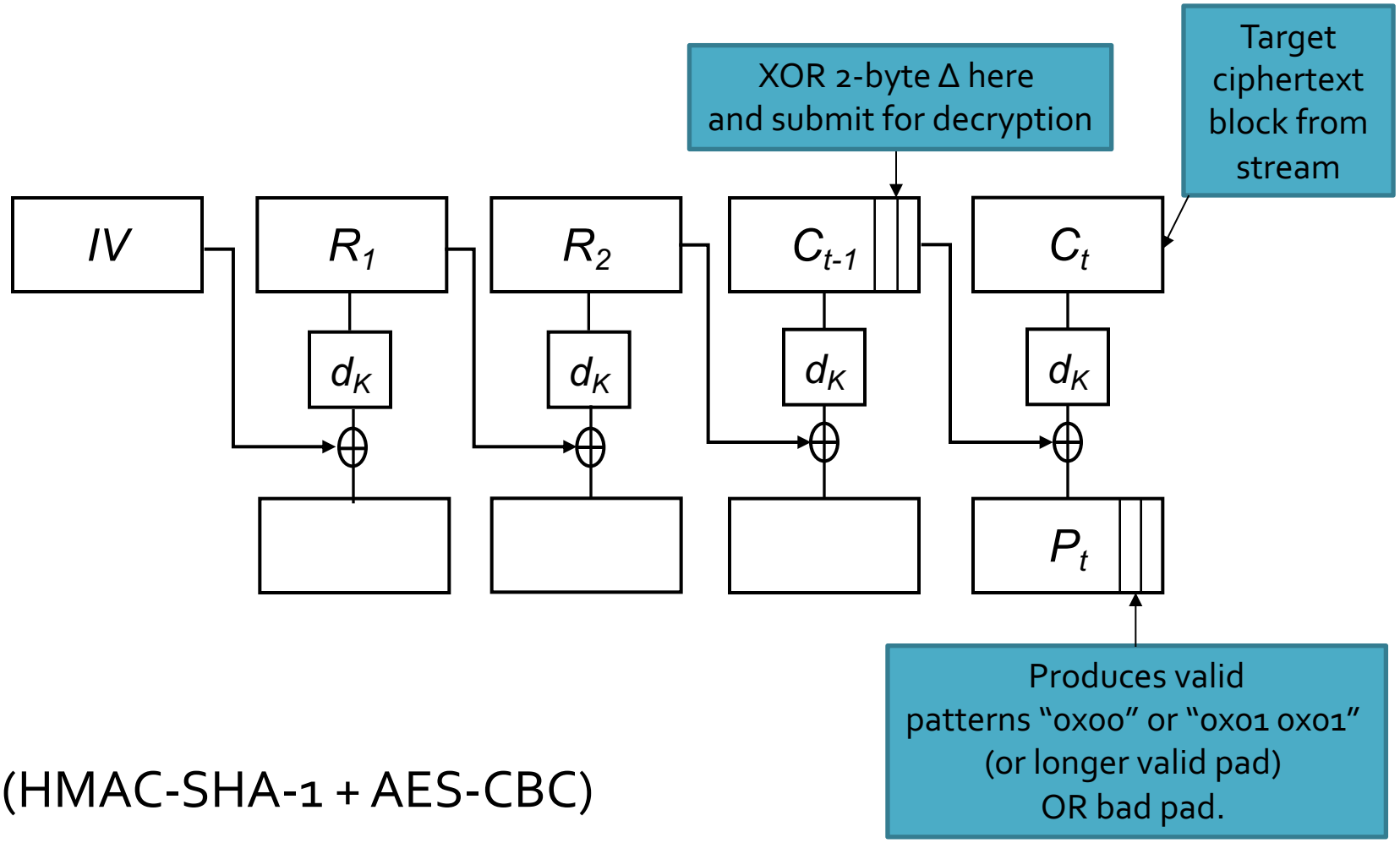
Reminder: HMAC



Lucky 13: Main idea

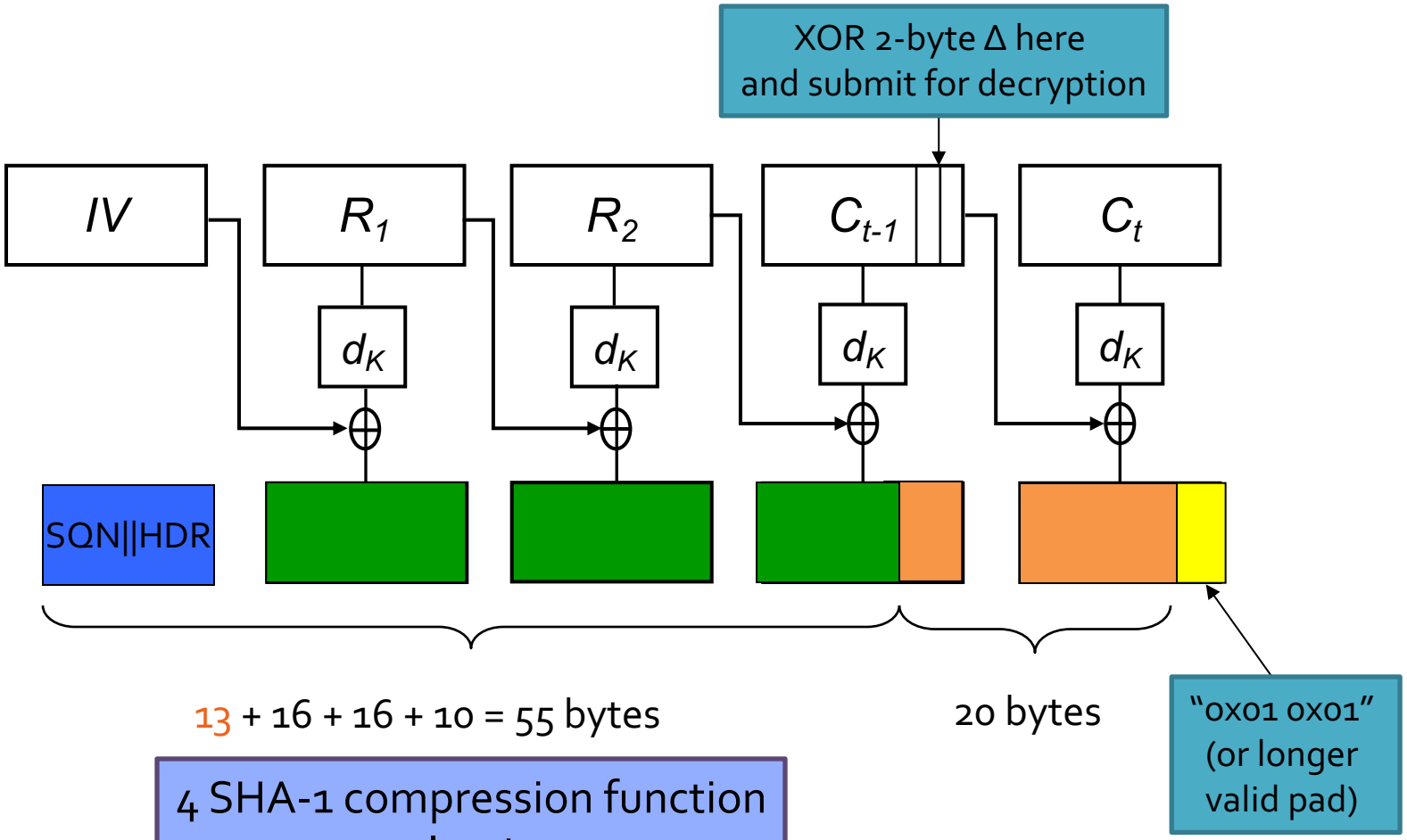
- MAC is HMAC and is computed on SQN || HDR || PAYLOAD.
- HMAC hash computation itself involves:
  - Adding ≥ 1 byte of padding + 8 bytes of length encoding.
  - Iteration of inner hash compression function, e.g. MD5, SHA-1, SHA-256.
  - Outer hash function computation.
- Running time of HMAC depends on L, the exact byte length of the protected data SQN || HDR || PAYLOAD:
  - $L \leq 55$ bytes: 4 compression function calls;
  - $56 \leq L \leq 119$: 5 compression function calls;
  - $120 \leq L \leq 183$: 6 compression function calls; etc.
  - These numbers come from interaction between compression function input size (k) and HMAC padding scheme.

Lucky 13

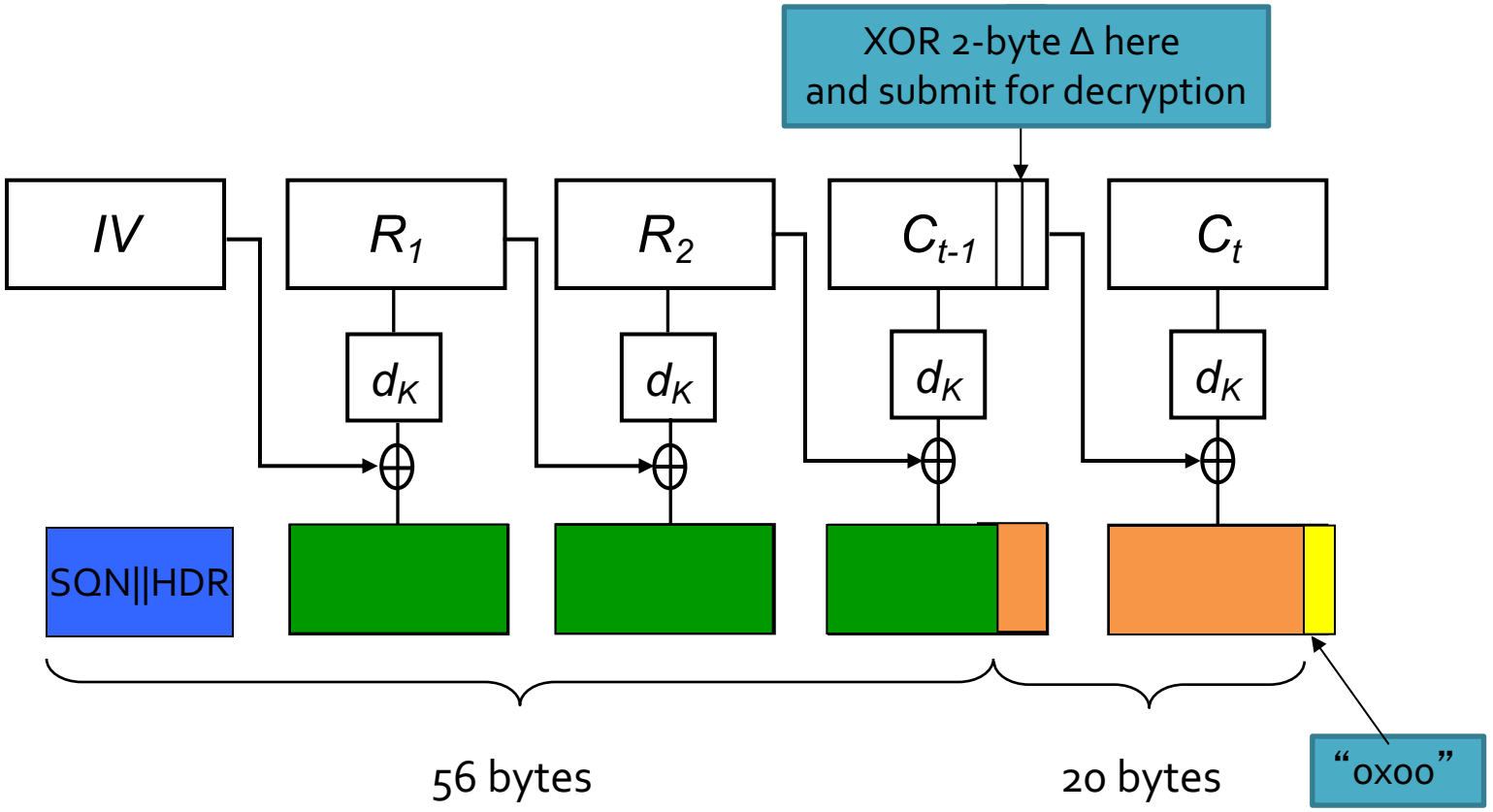


(HMAC-SHA-1 + AES-CBC)

Case 1: "0x01 0x01" (or longer valid pad)

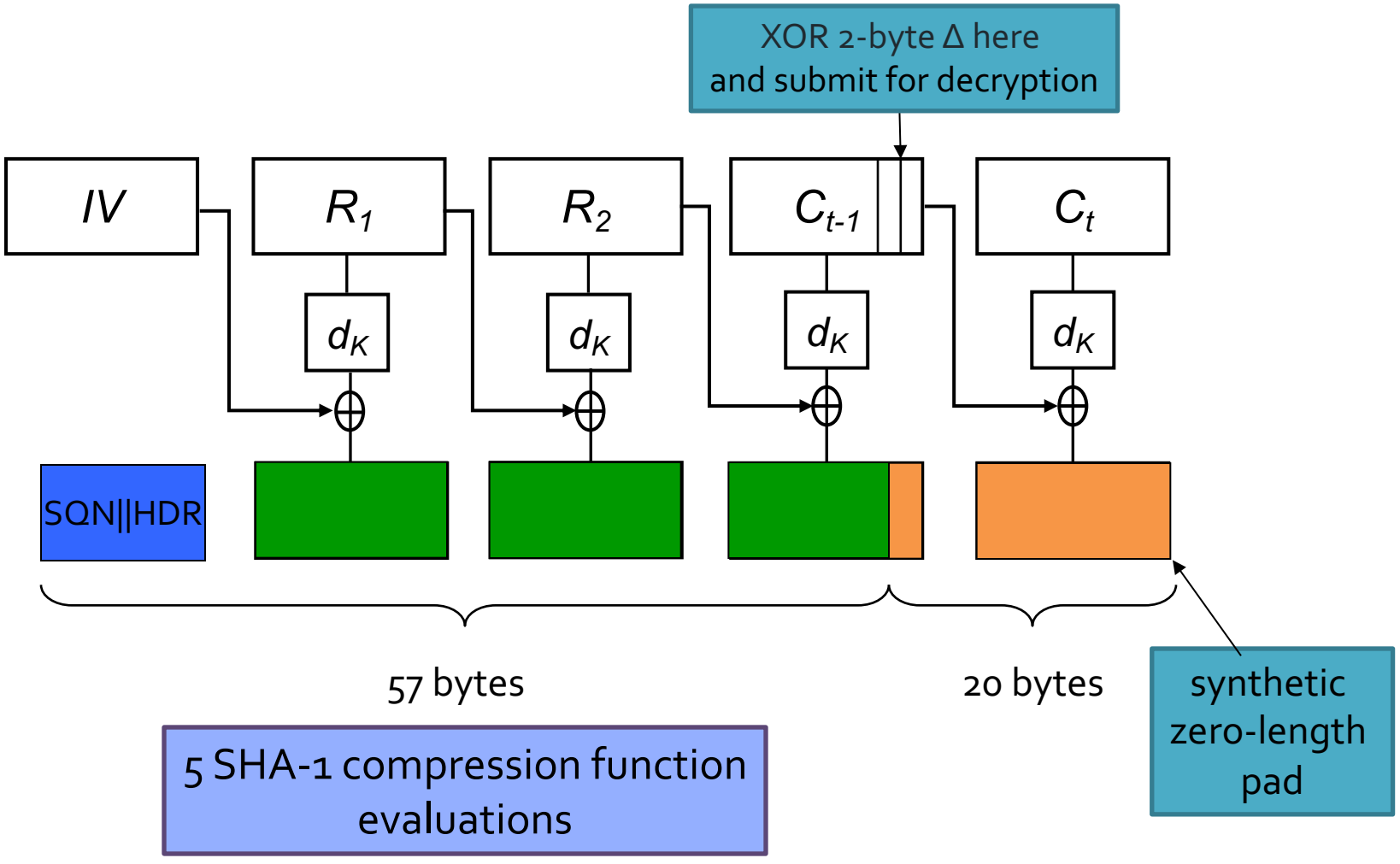


Case 2: "0x00"



5 SHA-1 compression function evaluations

Case 3: bad padding



Lucky 13 – plaintext recovery

- The injected ciphertext causes bad padding and/or a bad MAC.
  - This leads to a TLS error message, which the attacker times.
- There is a timing difference between "0x01 0x01" case and the other 2 cases.
  - Time taken for a single SHA-1 compression function evaluation.
  - Roughly 500-1000 clock cycles.
  - Measurable difference on same host, LAN, or a few hops away.
- Detecting the "0x01 0x01" case allows the last 2 plaintext bytes in the target block C_t to be recovered.
  - Using standard CBC algebra.
  - Attack then extends to all bytes as in a standard padding oracle attack.

Lucky 13 – improvements (attacks get better!)

- If 1-out-of-2 of the initial target bytes is already known, then complexity can be reduced from 2^{16} queries down to 2^{8} queries.
- If the plaintext is base64 encoded, then we only need 2^{6} attempts per byte.
- Need to multiply total number of queries by some factor to deal with noisy timing measurements.
  - Noise comes from other processes on same CPU + network jitter.
- BEAST-style attack targeting HTTP cookies.
  - Malicious client-side Javascript makes HTTP GET requests.
  - TLS sessions are automatically generated and HTTP cookies attached.
  - GET requests are "padded" so that 1-out-of-2 condition always holds.
  - Final cost of attack is about 2^{13} GET requests per byte of cookie.

Lucky 13 – countermeasures

- We really need constant-time decryption for TLS-CBC.
- Add dummy hash compression function computations when padding is good to ensure total is the same as when padding is bad.
- Add dummy padding checks to ensure number of iterations done is independent of padding length and/or correctness of padding.
- Watch out for length sanity checks too!

Constant time decryption for MEE

- Proper constant-time, constant-memory access implementation is really hard.
  - No branching on secret data.
- See Adam Langley's blog:

  https://www.imperialviolet.org/2013/02/04/luckythirteen.html

  for full details on how Lucky 13 was fixed in OpenSSL and NSS, requiring about 500 new line of 'C'.
- Other implementations adopted "approximate constant time" patches, e.g. GnuTLS or "add random noise", e.g. AWS.
  - See "Lucky Microseconds" paper for breakage of AWS s2n approach.
  - For breakage of all the rest, see : Ronen, Paterson, Shamir: *Pseudo Constant Time Implementations of TLS Are Only Pseudo Secure*. ACM CCS 2018: 1397-1414.

Extra slides: CCM

CCM

CCM = Counter with CBC-MAC.

- Basically, an instantiation of nonce-based AEAD using MtE with M = CBC-MAC and E = CTR mode, using a 128-bit block cipher, e.g. AES.

Tweaks:

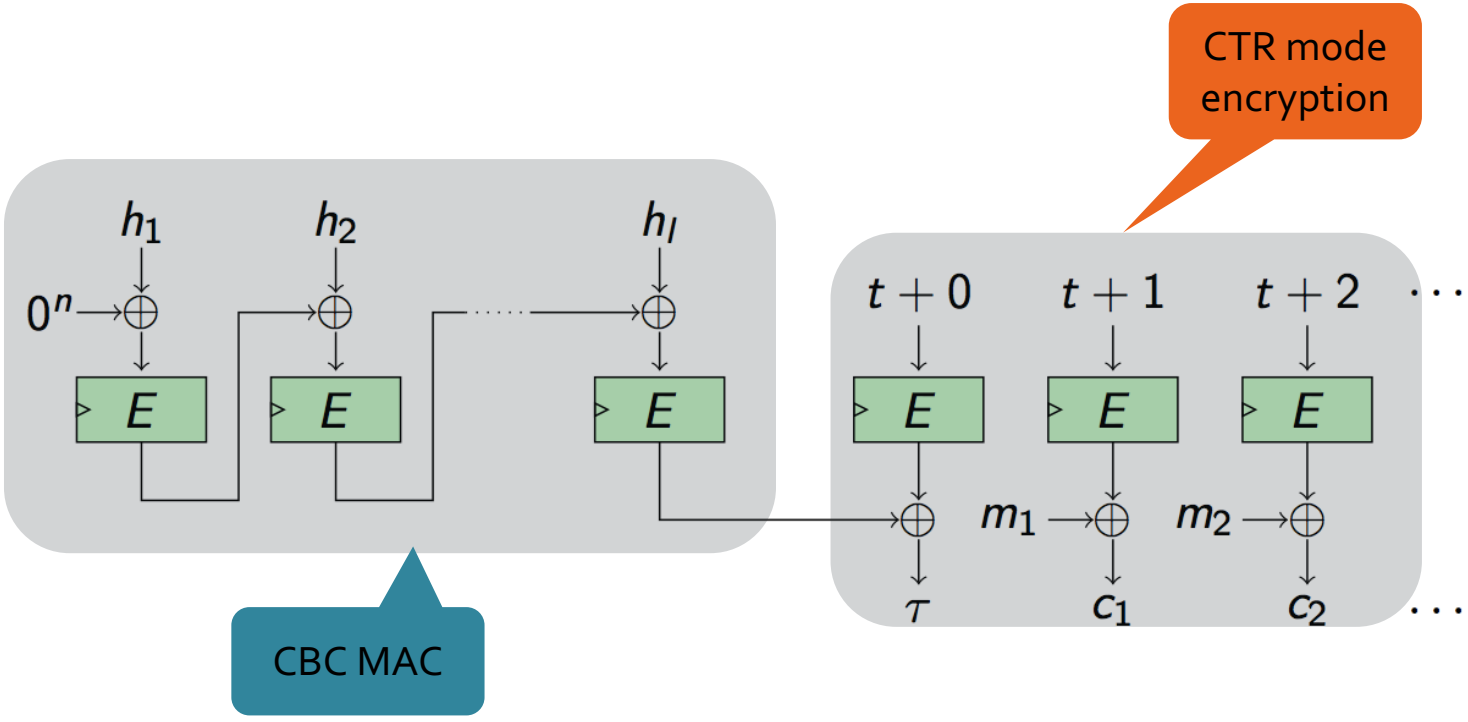
- Uses same key for "M" and "E" components.
- This is a Bad Idea™ in general, provably OK here.
- Apply CBC-MAC to the string:

$$h = N \,||\, \text{len}(m)_{64} \,||\, m \,||\, \text{len}(AD)_{64} \,||\, AD \,||\, \text{padding}$$

(where $\text{len}(X)_{64}$ means the 64-bit encoding of len. of string X) to get a tag τ.

- Tag can be truncated as desired.
- Encrypt $\tau \,||\, m$ using CTR mode.
- Initial counter value for CTR mode is $N \,||\, \text{oxoo}^{16-|N|}$, where N is the nonce.

CCM, pictorially



CCM

CCM = Counter with CBC MAC.

- CCM is quite slow: it needs 1 pass over associated data *AD* in CBC-MAC and two passes over the message *m*, one in CBC-MAC and one in CTR-mode encryption.
- CCM only uses block cipher and only in "forward direction", i.e. only "E" and no "D".
- CCM is patent-free.
- CCM is used in WPA2, a successor to WEP and WPA/TKIP.
- CCM is standardised for use in IPsec, TLS 1.2 and TLS 1.3.
- CCM is specified in full in RFC 3610 (https://tools.ietf.org/html/rfc3610).
- CCM has as a security proof based on block cipher being a pseudo-random permutation.
- No known attacks (when implemented properly!)