Cryptography, Part 2

CMSC 23200, Spring 2026, Lecture 6

David Cash and Grant Ho

University of Chicago

Logistics

1. Assignment 2 is due tonight at 11:59pm
2. Assignment 3 will be released tomorrow
3. Discussion sections meet Monday (will cover crypto)

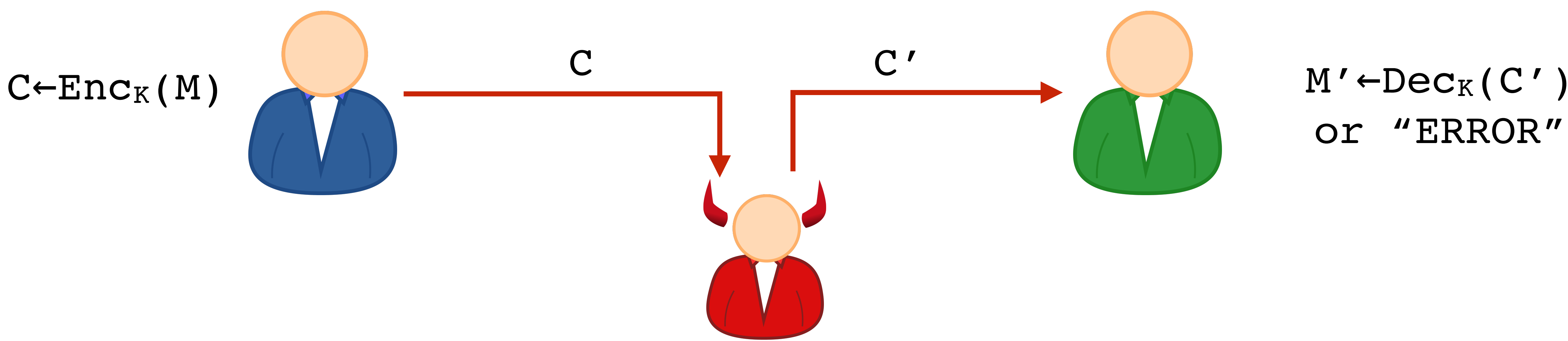
Outline of Lecture 6

1. Message Authentication Codes (MACs)
2. Cryptographic Hash Functions
3. Authenticated Encryption (AE)
4. AE Case Study: Padding Oracle Attacks
5. Public Key Encryption
6. Digital Signatures

Outline of Lecture 6

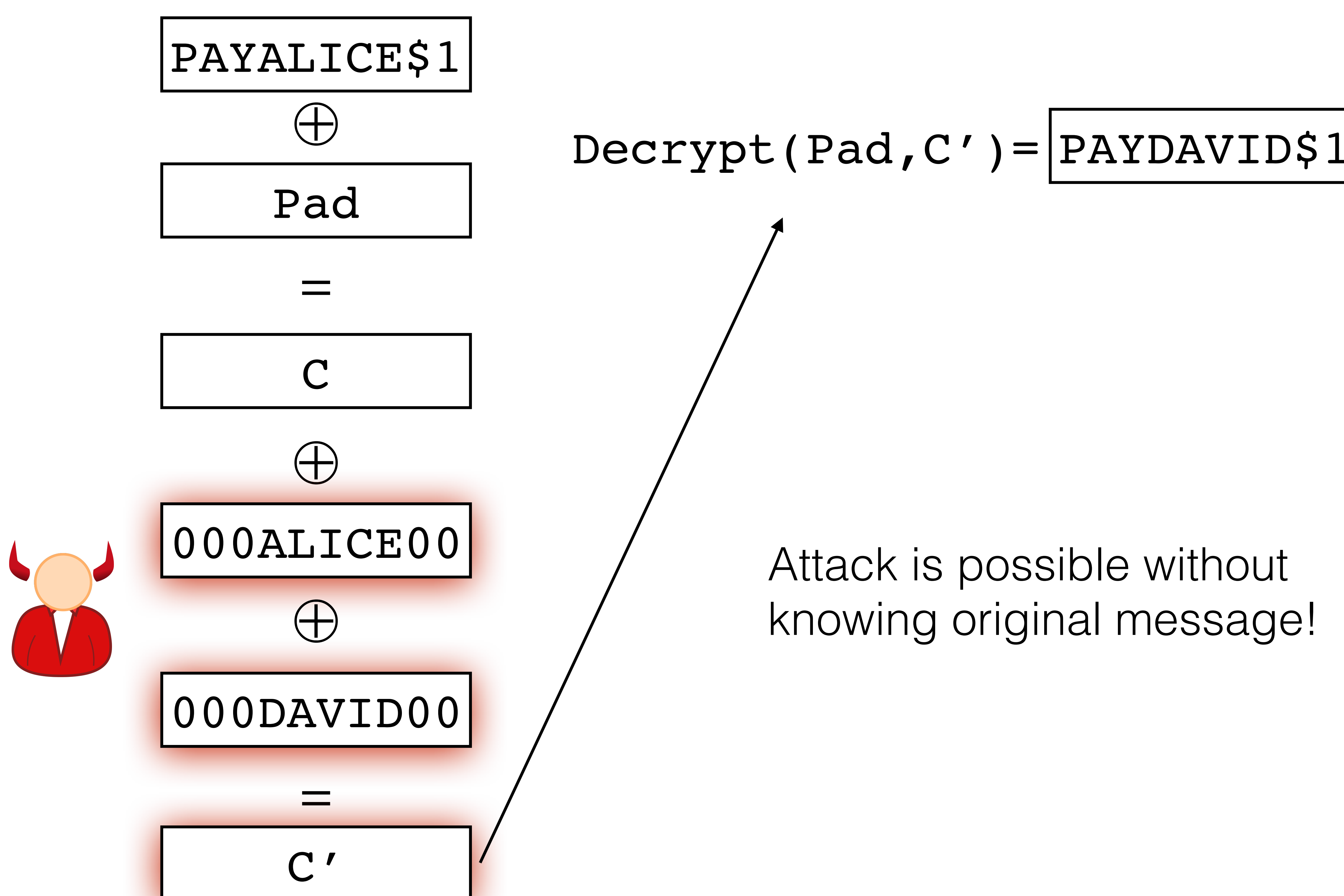
1. **Message Authentication Codes (MACs)**
2. Cryptographic Hash Functions
3. Authenticated Encryption (AE)
4. AE Case Study: Padding Oracle Attacks
5. Public Key Encryption
6. Digital Signatures

Encryption Integrity: An abstract setting



Encryption satisfies **integrity** if it is infeasible for an adversary to send a new `C'` such that `Dec`$_K$`(C')`≠ERROR.

Issue #2: One-Time Pad Does Not Provide Integrity

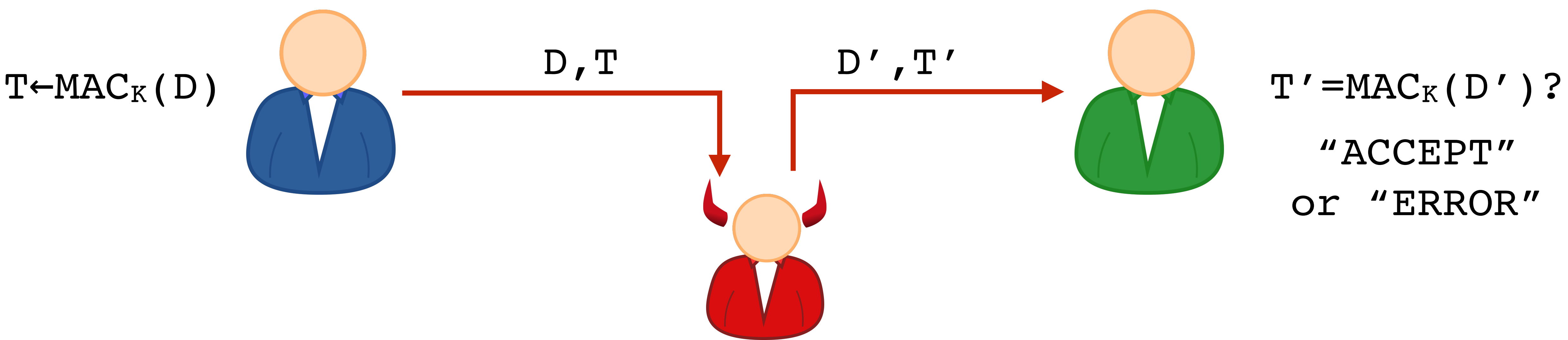


Message Authentication Codes

A **message authentication code (MAC)** is an algorithm that takes as input a key and a message, and outputs an "unpredictable" **tag.**

- D will usually be a ciphertext, but is often called a "message".
- Note: MACs are not block ciphers or encryption

MAC Security Goal: Unforgeability



MAC satisfies **unforgeability** if it is infeasible for Adversary to fool Bob into accepting `D'` not previously sent by Alice.

MAC Security Goal: Unforgeability

Note: No encryption on this slide. MACs do not provide confidentiality

`D = please pay ben 20 bucks`

`T = 827851dc9cf0f92ddcdc552572ffd8bc`

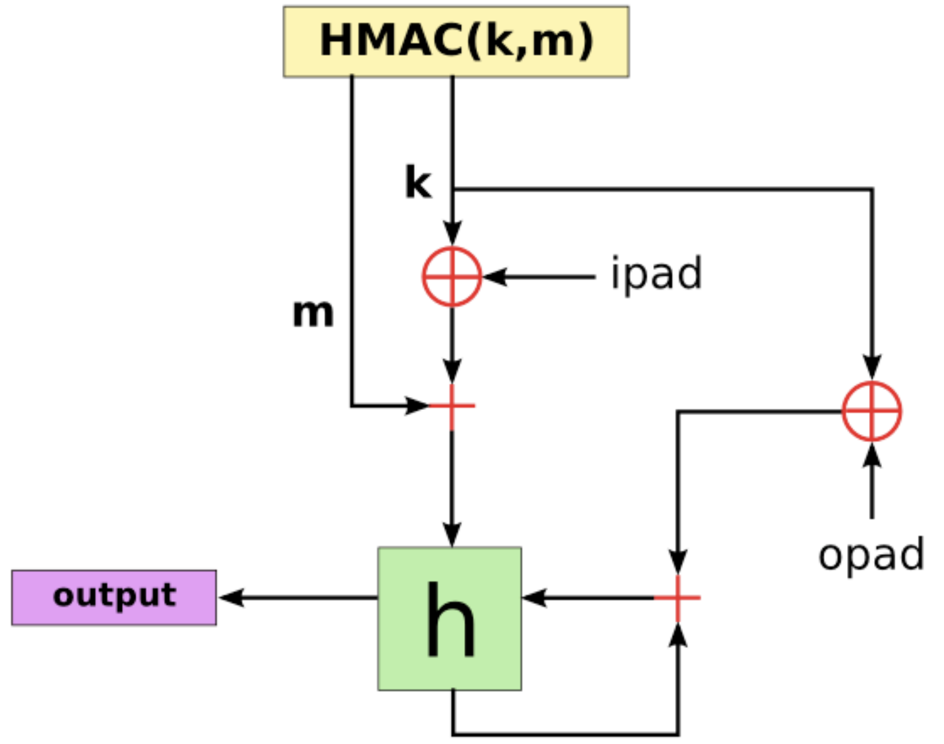
`D’= please pay ben 21 bucks`

`T’= baeaf48a891de588ce588f8535ef58b6`

Should be hard to predict **`T’`** for any new **`D’`**.

MACs In Practice: Use HMAC or Poly1305-AES

- More on how MACs are constructed in a moment.



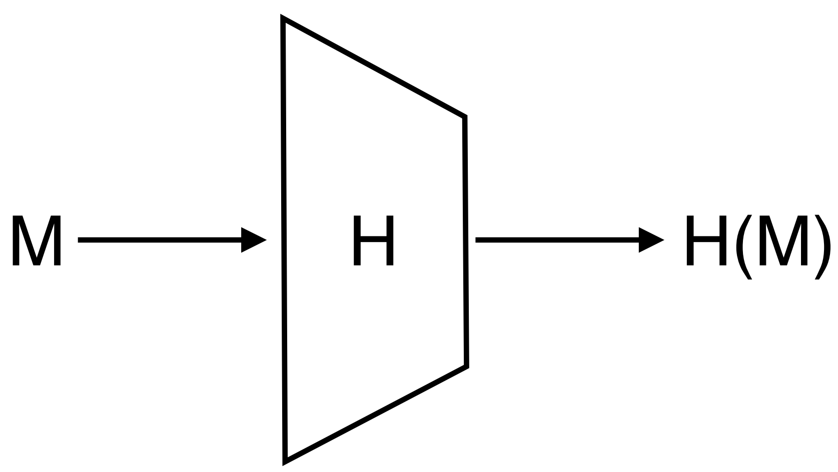
- Other, less-good option: AES-CBC-MAC (bug-prone)

Outline of Lecture 6

1. Message Authentication Codes (MACs)
2. **Cryptographic Hash Functions**
3. Authenticated Encryption (AE)
4. AE Case Study: Padding Oracle Attacks
5. Public Key Encryption
6. Digital Signatures

Next Up: Hash Functions

Definition: A hash function is a deterministic function H that reduces arbitrary strings to fixed-length outputs.

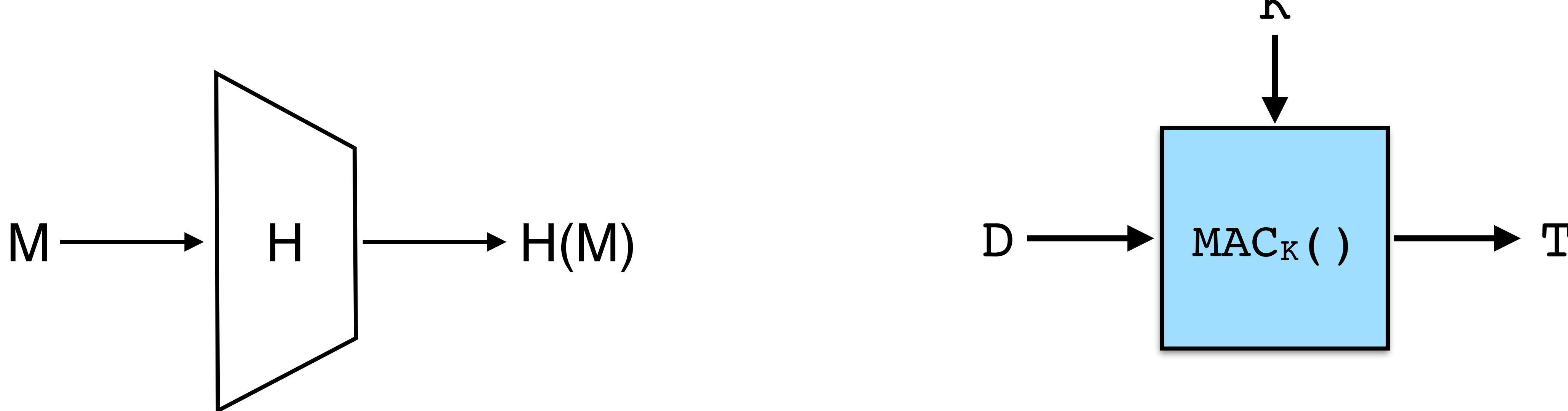


Primary security goal:

- **collision resistance**: can't find `M≠M'` such that `H(M)=H(M')`

Note: Very different from hashes used in data structures!

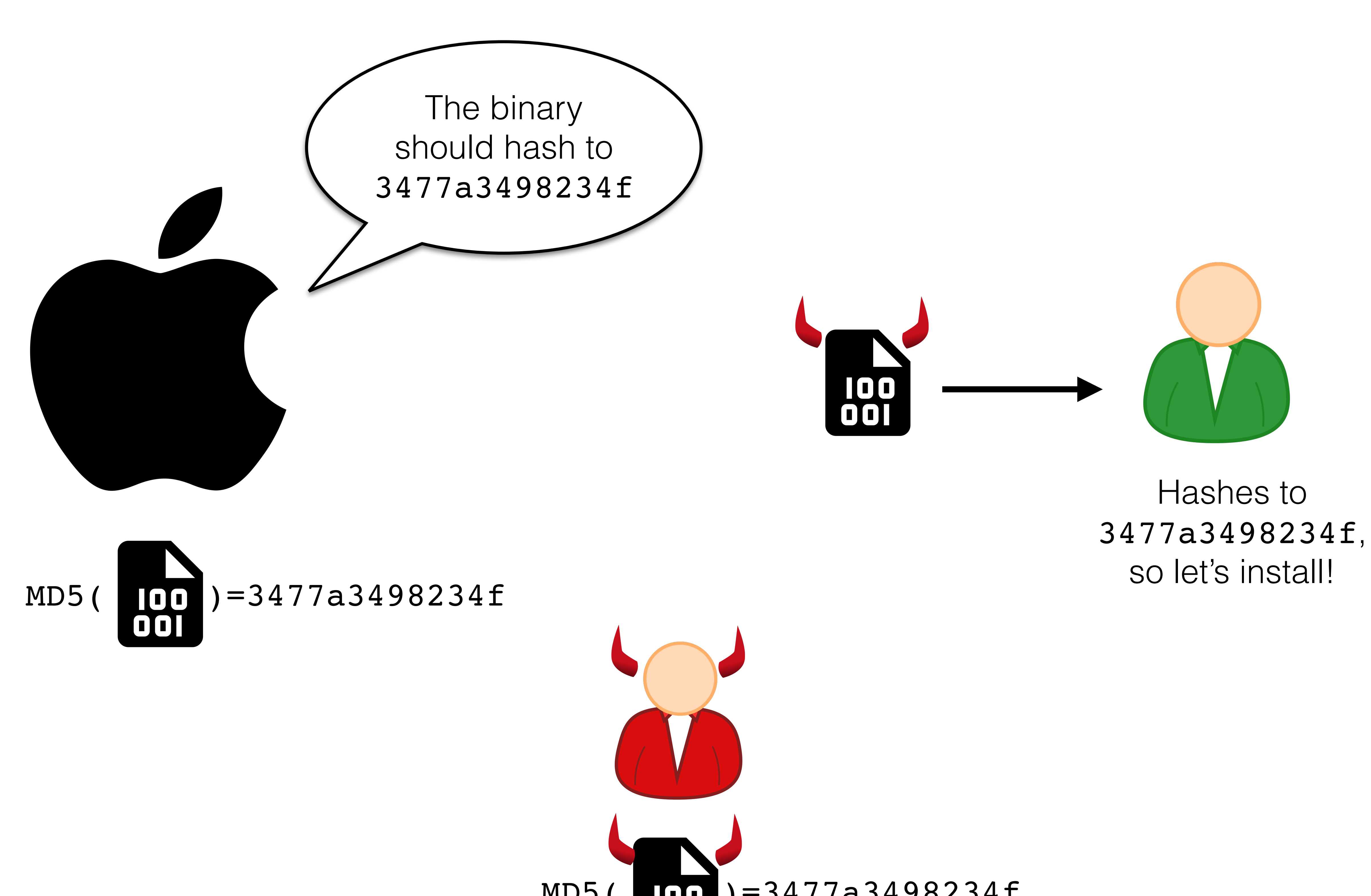
Hash Functions are not MACs



Both map long inputs to short outputs… but a hash function does not take a key.

Intuition: a MAC is like a hash function, that only the holders of key can evaluate.

Why are collisions bad?



Practical Hash Functions

Name	Year	Output Len (bits)	Broken?
MD5	1993	128	Super-duper broken
SHA-1	1994	160	Yes
SHA-2 (SHA-256)	1999	256	No
SHA-2 (SHA-512)	2009	512	No
SHA-3	2019	>=224	No

Confusion over “SHA” names leads to vulnerabilities.

Hash Function Security History

- MD5 (1992) was broken in 2004 - can now find collisions very quickly.
- SHA-1 (1995) was broken in 2017 - A big computer can find collisions
- SHA-256/SHA-512 (2001) are not broken
- SHA-3 (2015) is new and not broken

```
MD5( d131dd02c5e6eec4693d9a0698aff95c 2fcab58712467eab4004583eb8fb7f89
     55ad340609f4b30283e488832571415a 085125e8f7cdc99fd91dbdf280373c5b
     d8823e3156348f5bae6dacd436c919c6 dd53e2b487da03fd02396306d248cda0
     e99f33420f577ee8ce54b67080a80d1e c69821bcb6a8839396f9652b6ff72a70 )
```

```
= MD5( d131dd02c5e6eec4693d9a0698aff95c 2fcab50712467eab4004583eb8fb7f89
       55ad340609f4b30283e4888325f1415a 085125e8f7cdc99fd91dbd7280373c5b
       d8823e3156348f5bae6dacd436c919c6 dd53e23487da03fd02396306d248cda0
       e99f33420f577ee8ce54b67080280d1e c69821bcb6a8839396f965ab6ff72a70 )
```

Xiaoyun Wang (Tsinghua University), 2004

MACs from Hash Functions

Goal: Build a secure MAC out of a good hash function.

Construction: `MAC(K,D) = H(K+D)`

Warning: Broken

- Here, "+" means string concatenation
- Totally insecure if H = MD5, SHA1, SHA-256, SHA-512
- May be secure with SHA-3 (but don't do it)

Construction: `MAC(K,D) = H(D+K)`

Just don't

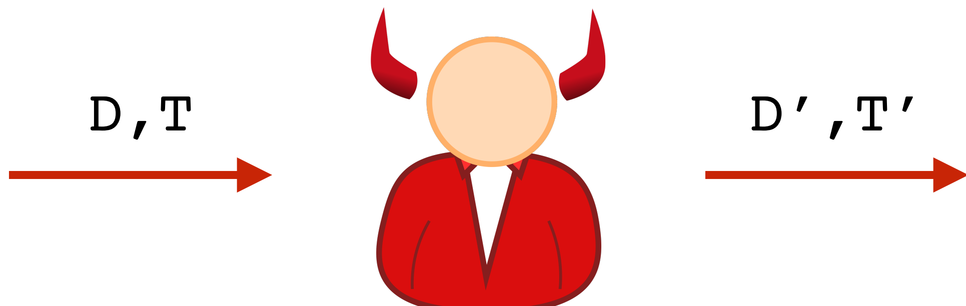
Upshot: Use HMAC; It's designed to avoid this and other issues.

Later: Other applications of hash functions

Length Extension Attack

Construction: `MAC(K,D) = H(K+D)`

Adversary goal: Find new message `D'` and a valid tag `T'` for `D'`



Need to find: Given `T=H(K+D)`, find `T'=H(K+D')` without knowing `K`.

In Assignment 3: Break this construction!

Review of Crypto So Far

- **Stream ciphers** stretch keys for use with the OTP and provide good confidentiality
- **Block ciphers** encrypt small blocks (e.g. 16 bytes) and must be run in a “mode” to encrypt large messages.
  - ECB is a broken mode
  - CTR should be the default, but CBC can be good too
- **MACs** should be used to prevent changing/inserting ciphertexts
  - HMAC and Poly-1305 are both secure
- **Hash Functions** take long inputs and produce short outputs, and should resist collision-finding attacks
  - SHA2 and SHA3 are secure, but MD5 and SHA1 are not

Outline of Lecture 6

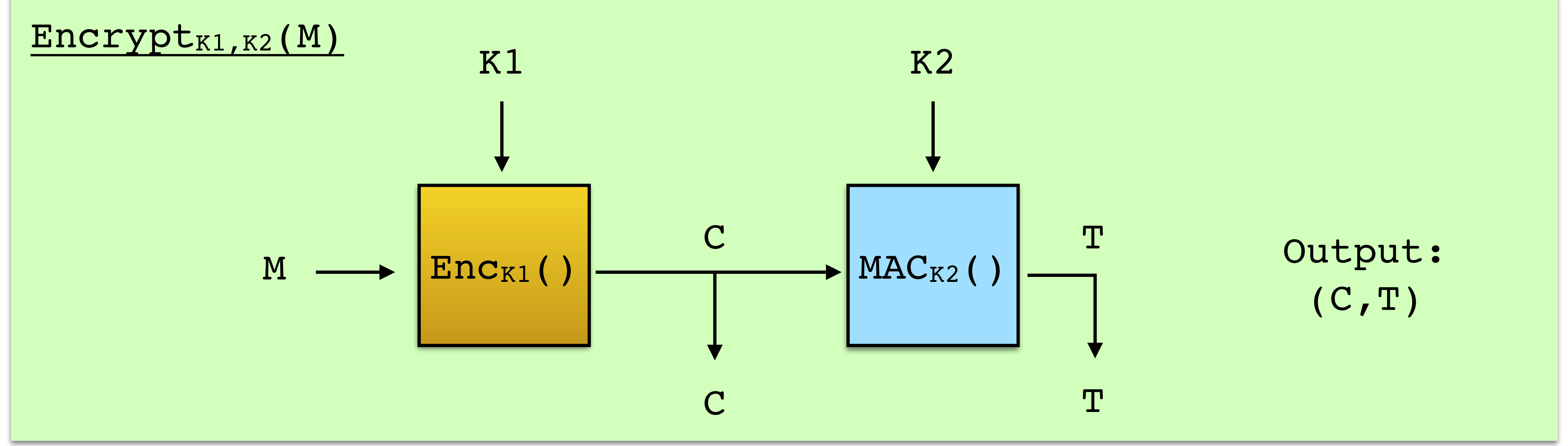
1. Message Authentication Codes (MACs)
2. Cryptographic Hash Functions
3. **Authenticated Encryption (AE)**
4. AE Case Study: Padding Oracle Attacks
5. Public Key Encryption
6. Digital Signatures

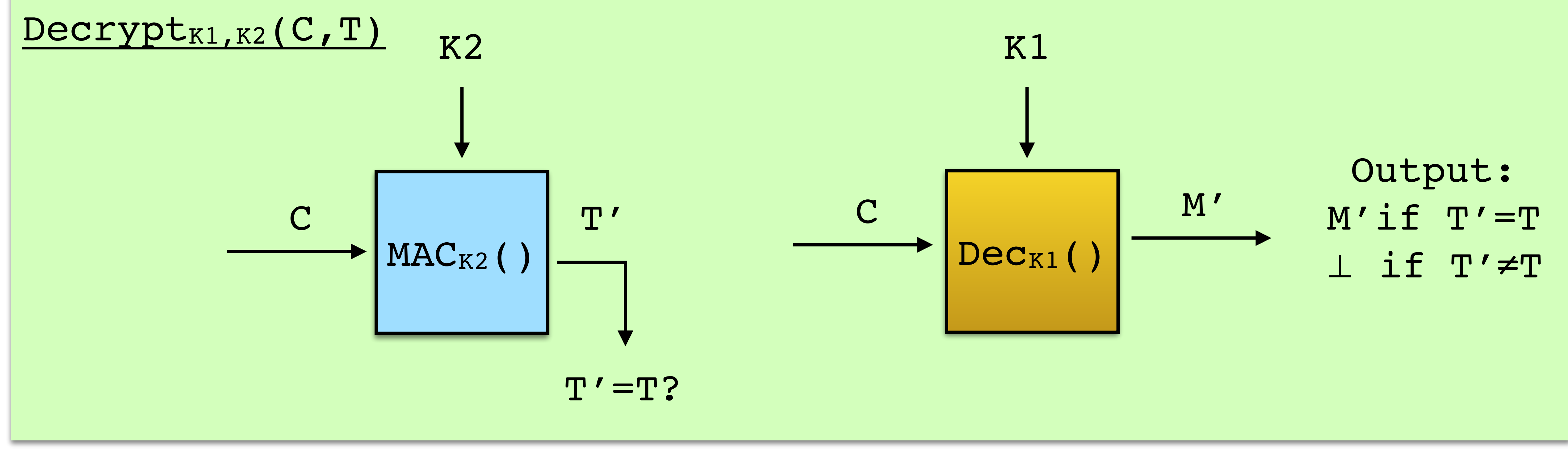
Authenticated Encryption

Encryption that provides **confidentiality** and **integrity** is called **Authenticated Encryption**.

- Built using a good stream cipher and a MAC.
- In practice: Always use ready-made Authenticated Encryption
  - Do not build it yourself!
  - Good AE constructions to use:
    - Chacha20-Poly1305
    - AES-GCM

Building Authenticated Encryption



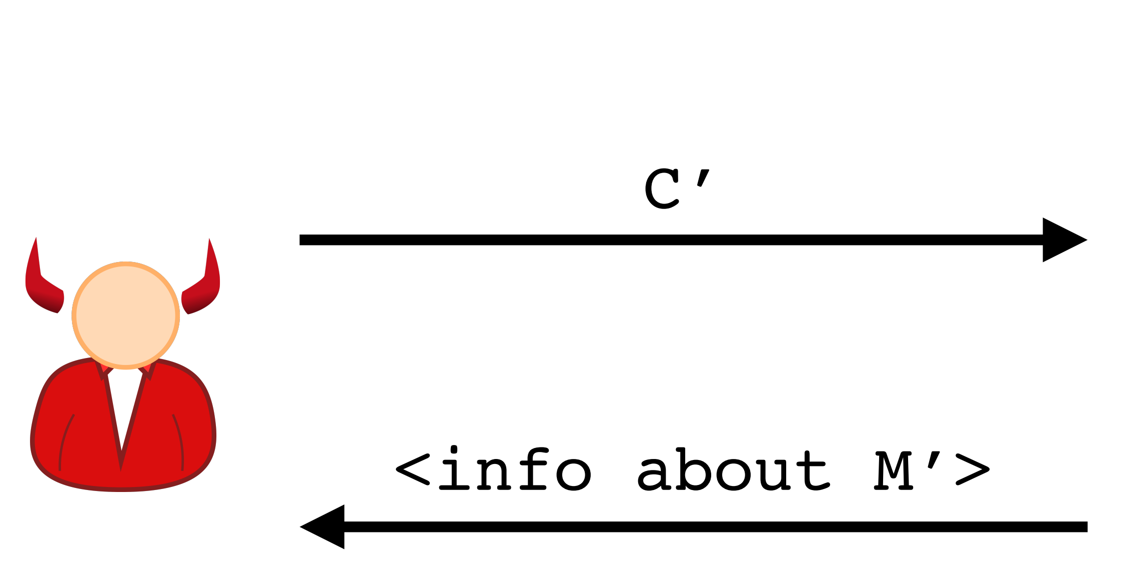


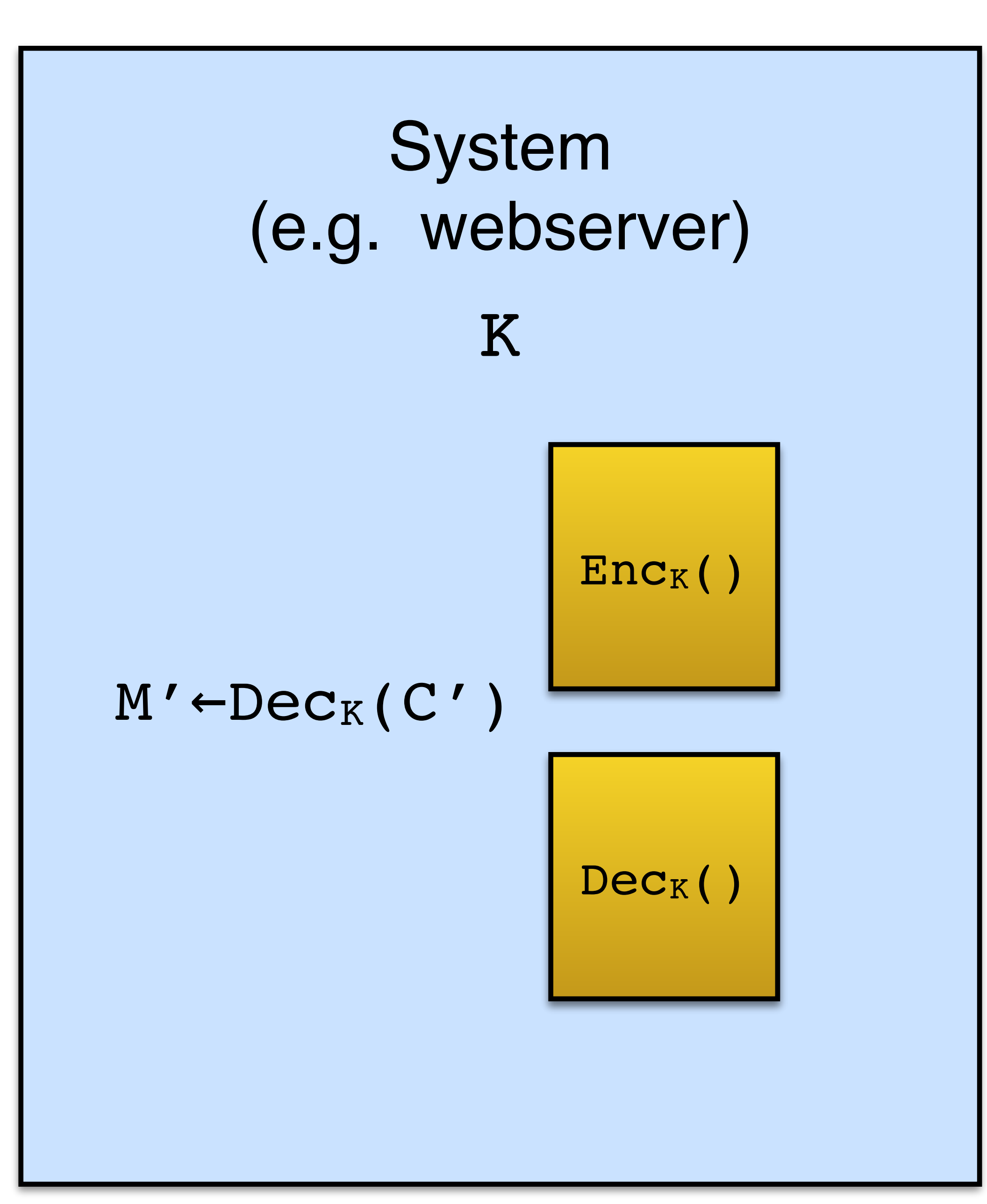
- Summary: Encrypt the message and then MAC the ciphertext

Outline of Lecture 6

1. Message Authentication Codes (MACs)
2. Cryptographic Hash Functions
3. Authenticated Encryption (AE)
4. **AE Case Study: Padding Oracle Attacks**
5. Public Key Encryption
6. Digital Signatures

Chosen-Ciphertext Attacks (CCAs) against Encryption

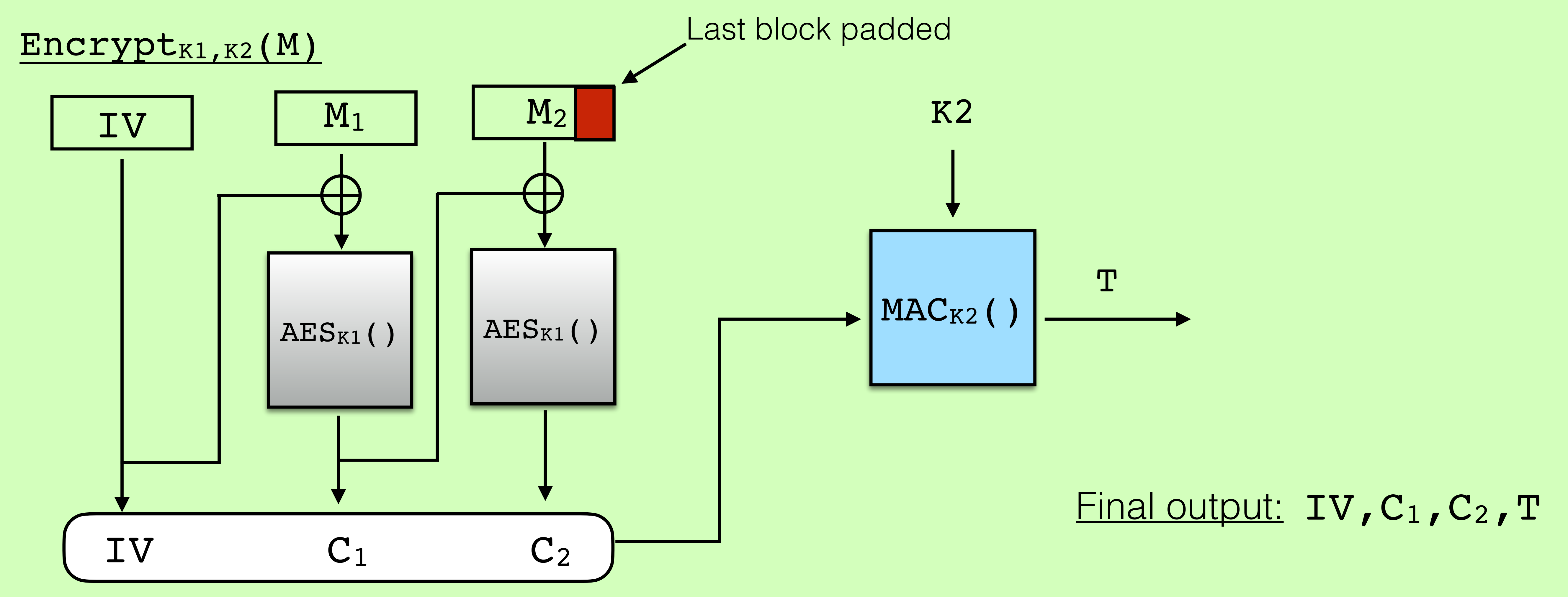




- Adversary provides ciphertext inputs to system
- Obtains info about decryptions of its ciphertexts

- Integrity + Confidentiality = security against CCAs

CBC Padding and MACs: When to reject ciphertexts



Decrypt$_{K1,K2}$(IV,C_1,C_2,T)

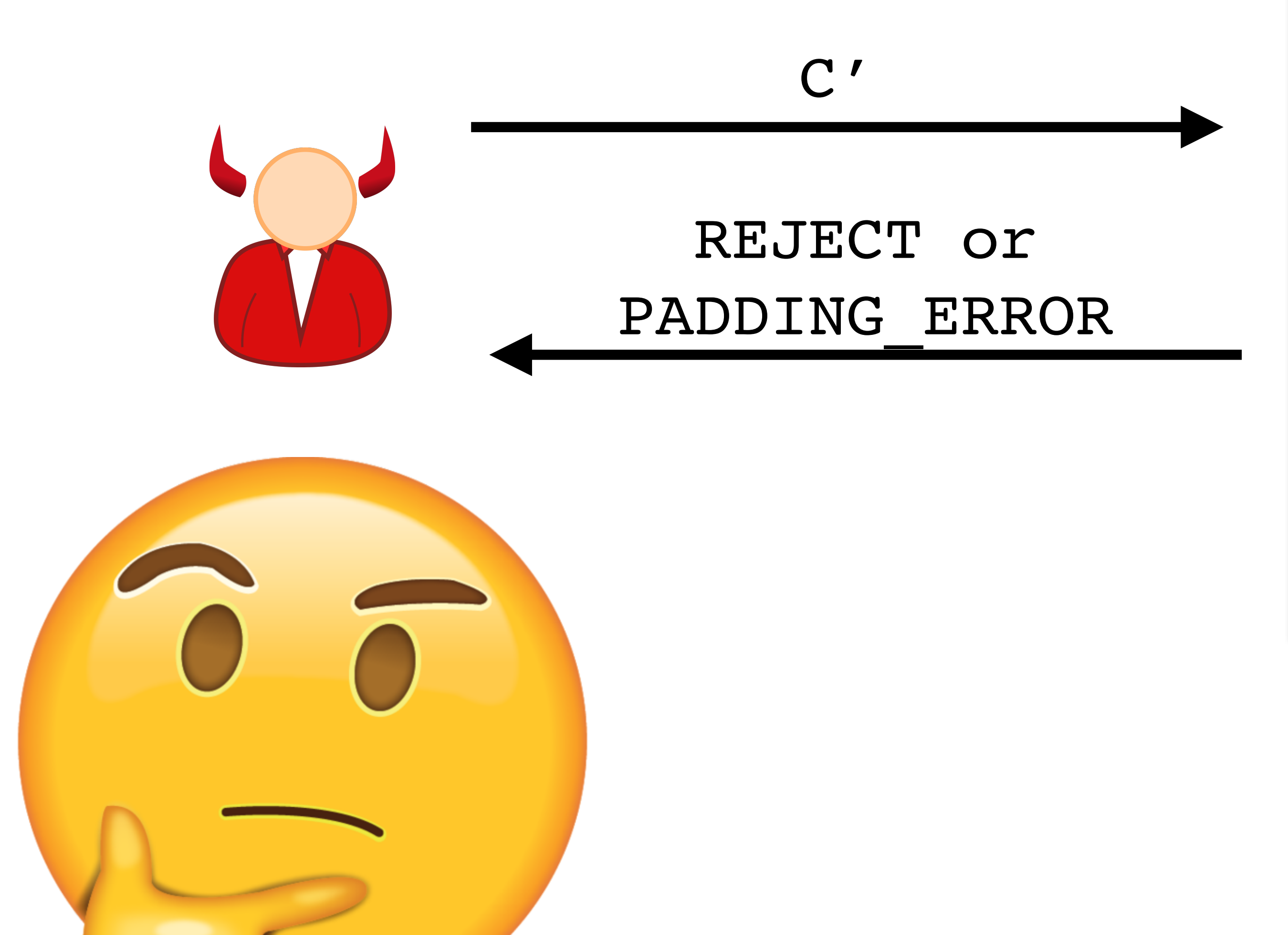
1. If tag T wrong:
   Output REJECT
2. M′←CBC-Decrypt$_{K1}$(IV,C_1,C_2)
3. If padding format wrong:
   Output PADDING_ERROR
4. Output M′

Decrypt$_{K1,K2}$(IV,C_1,C_2,T)

1. M′←CBC-Decrypt$_{K1}$(IV,C_1,C_2)
2. If padding format wrong:
   Output PADDING_ERROR
3. If tag T wrong:
   Output REJECT.
4. Output M′

Broken

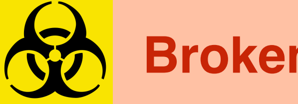
Padding Oracle Attacks



System
(e.g. webserver)
K

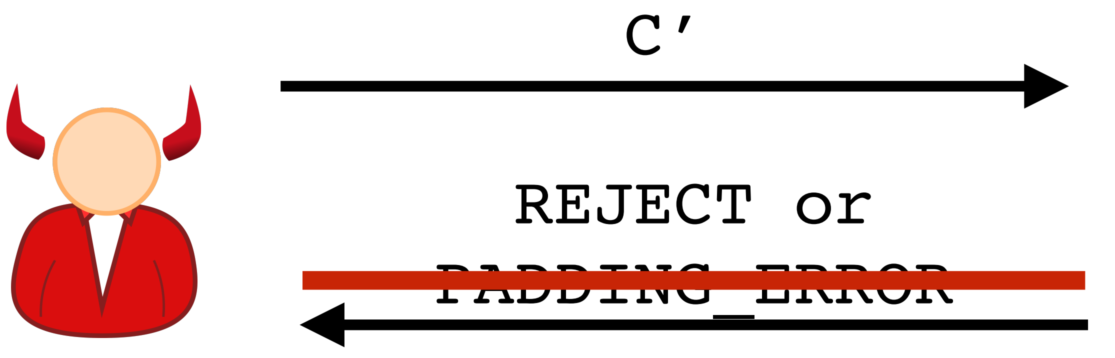
$\underline{\text{Decrypt}_{K1,K2}(IV, C_1, C_2, T)}$

1. $M' \leftarrow \text{CBC-Decrypt}_{K1}(IV, C_1, C_2)$
2. If padding format wrong:
   Output PADDING_ERROR
3. If tag T wrong:
   Output REJECT.
4. Output M′



Allows decryption of arbitrary ciphertexts by adversary!

Padding Oracle Attacks: It gets worse



REJECT in 10ms or
REJECT in 15ms

System
(e.g. webserver)
K

Real-world attacks against:
- TLS (2003)
- IPSec (2007,2010)
- Ruby on Rails (2010)
- ASP.NET (2010)
- SecurID Auth Tokens (2012)
- Steam Client (2016)

Output RE
different ti

Attack stil

Output REJECT.

<u>$\mathtt{Decrypt}_{K1,K2}(IV, C_1, C_2, T)$</u>

1. $M' \leftarrow \mathtt{CBC\text{-}Decrypt}_{K1}(IV, C_1, C_2)$
2. If padding format wrong:
   ~~Output PADDING_ERROR~~
3. If tag T wrong:
   Output REJECT.
4. Output M′

Broken

<u>Solutions:</u>
1. Constant-time code (extremely difficult).
2. Use un-padded encryption like CTR.

CBC Padding: PKCS7

- Need to pad a byte string up to a multiple of 16 bytes
- First look at how many bytes are missing. Here, need 10 bytes
- Fill missing k bytes with value k (k = 10 = 0x0A in example)

`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`

- If data is already a multiple of 16 bytes long, add an entire block of 0x10 bytes

`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`

Can't leave data unchanged;
Bytes might be interpreted as padding.

- Un-padding is easy

Invalid PKCS7 Padding Reminder

Fact: Not every byte string is a “valid padding”. Some strings have to be handled as “malformed”

`48 65 6C 6C 6F 20 43 53 31 37 37 20 73 74 75 64` `48 65 6C 6C 6F 20 43 53 31 37 37 20 73 74 01 02`

Invalid. Why?

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

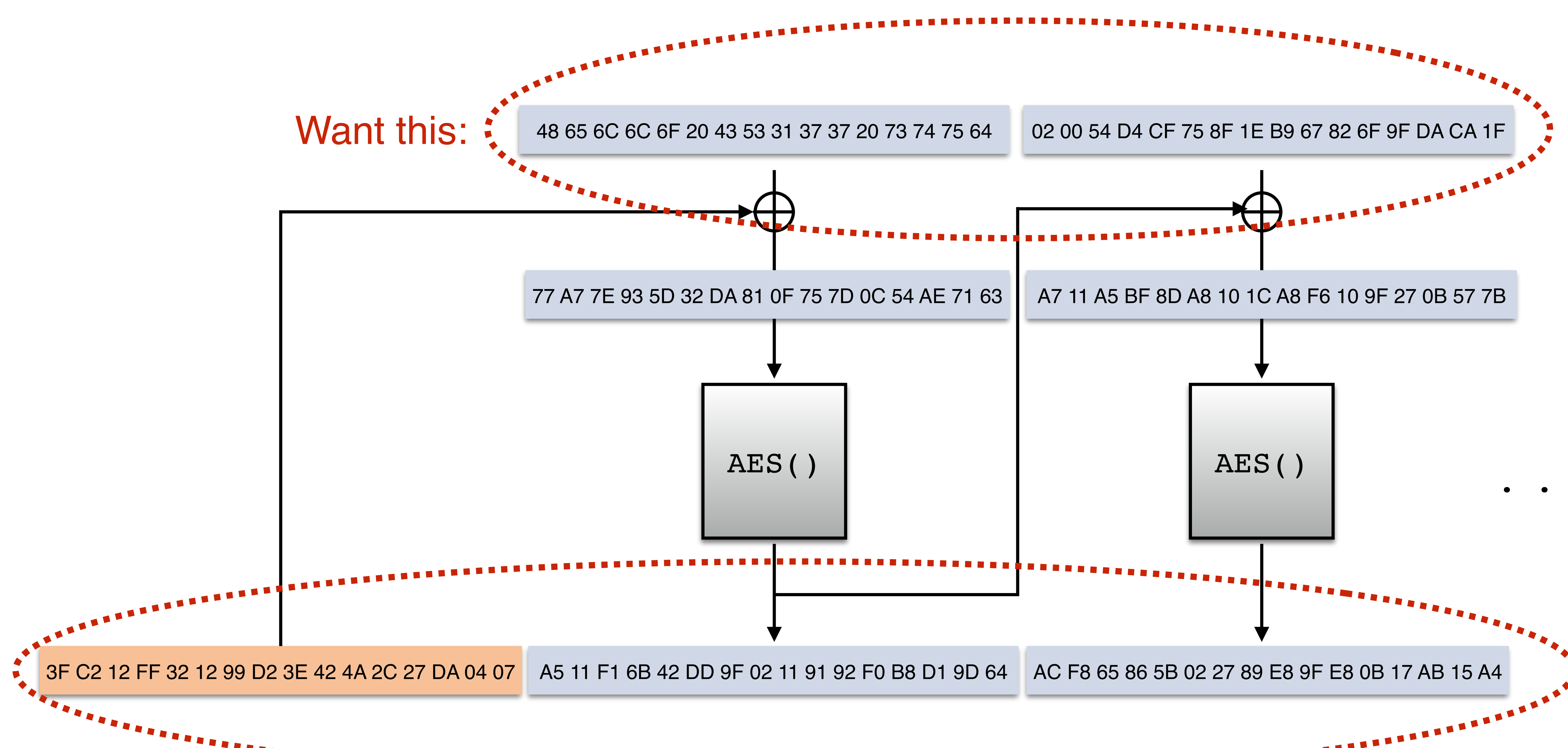
Valid. Why?

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

Valid! All strings that end in 0x01 are valid.

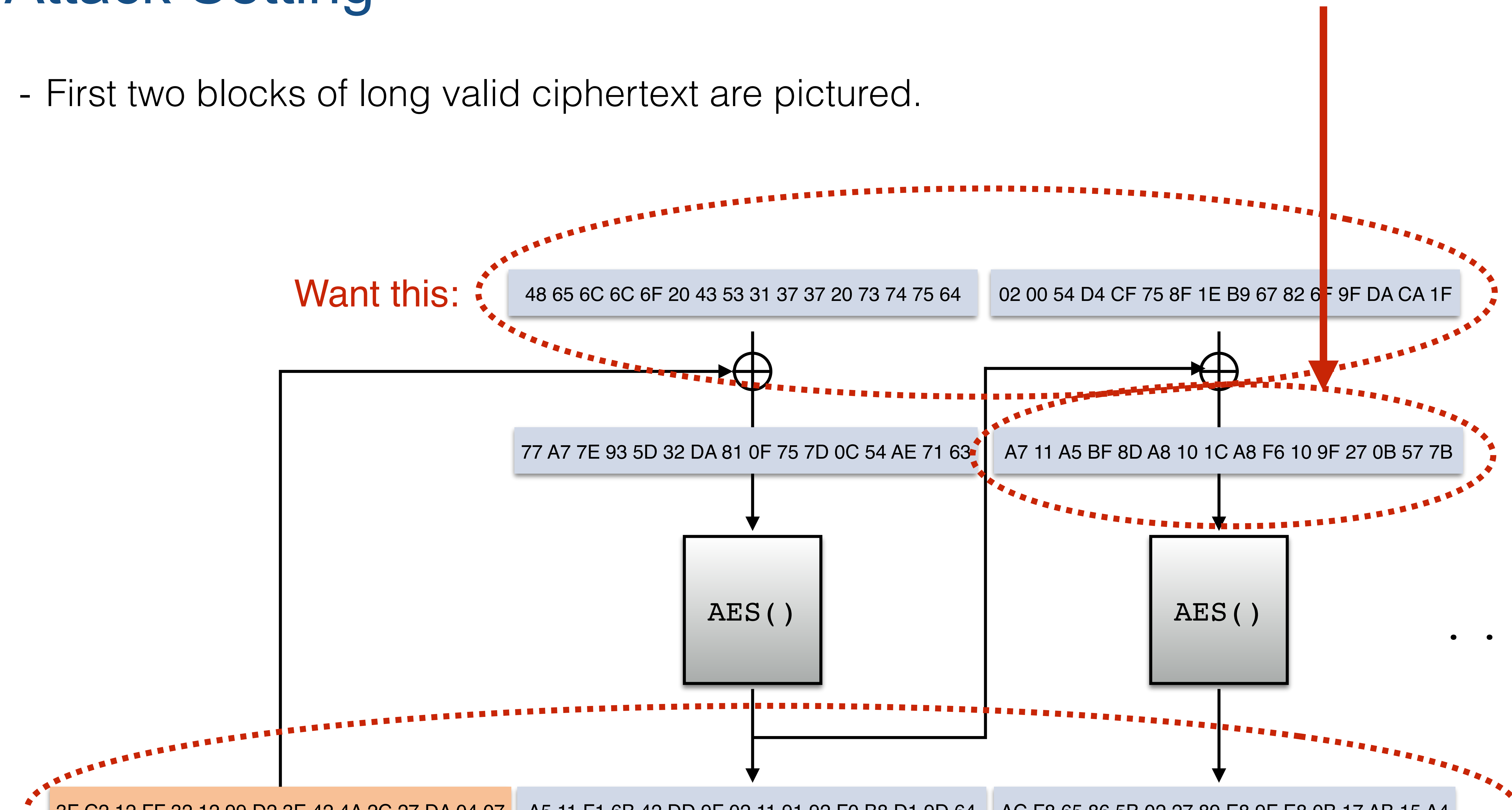
Attack Setting

- First two blocks of long valid ciphertext are pictured.



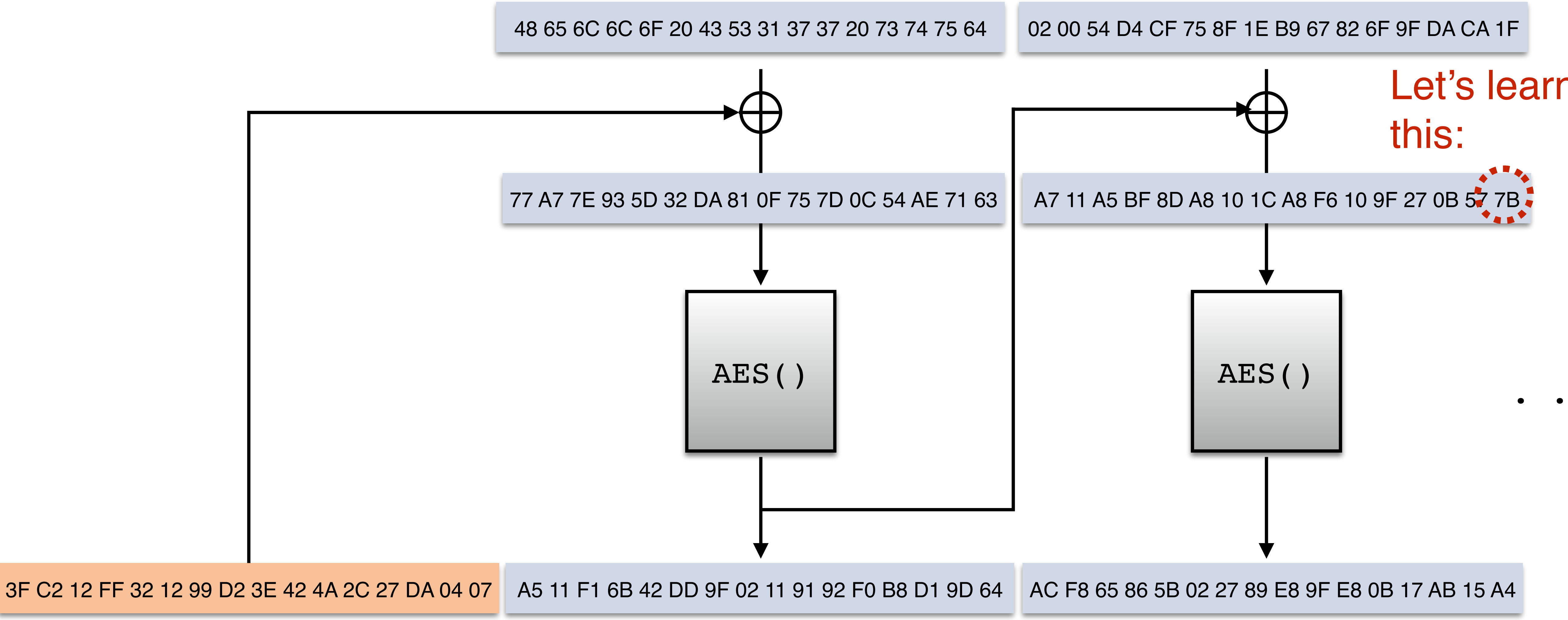
Attack Setting

- First two blocks of long valid ciphertext are pictured.



Initial goal: Learn last byte of a block

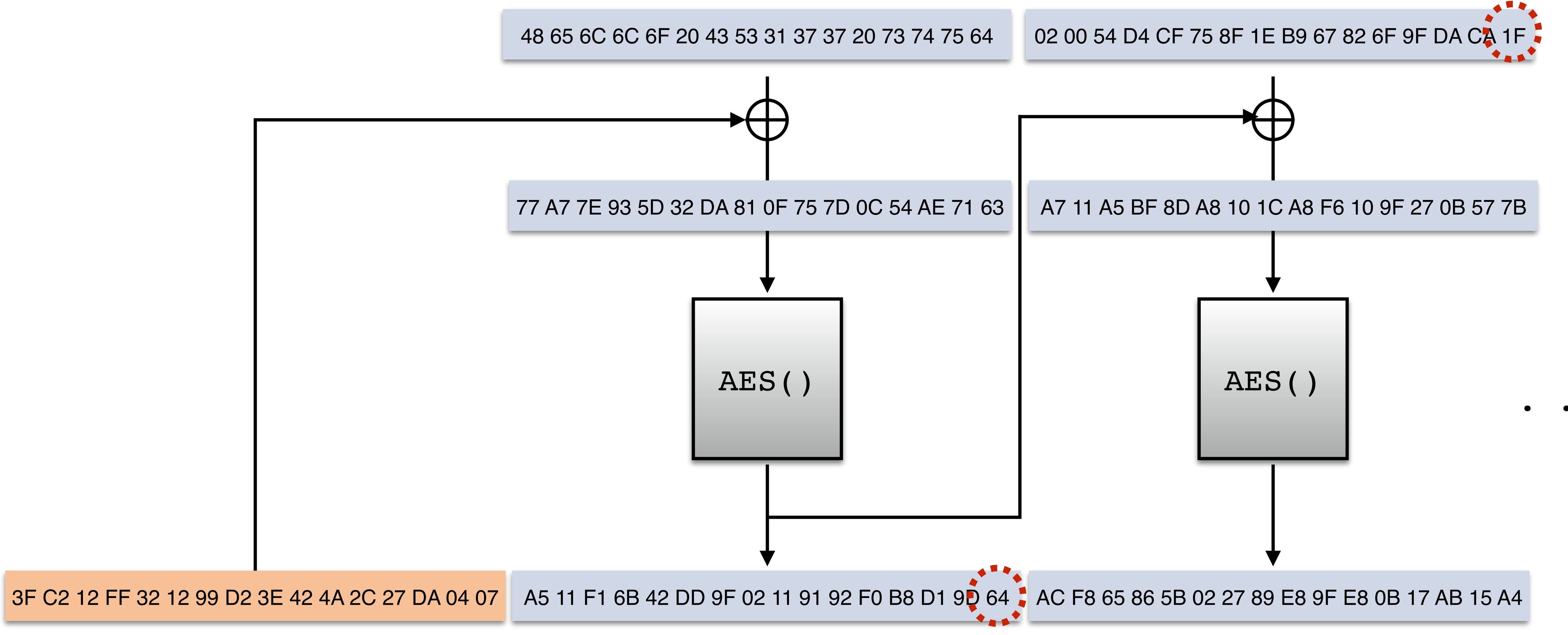
- First observation: If we truncate the long ciphertext down to these blocks, what will padding oracle say?
  - `Padding_Error`!



- What can we infer from the padding error response?
  - Last plaintext byte of that block could not have been 0x01.

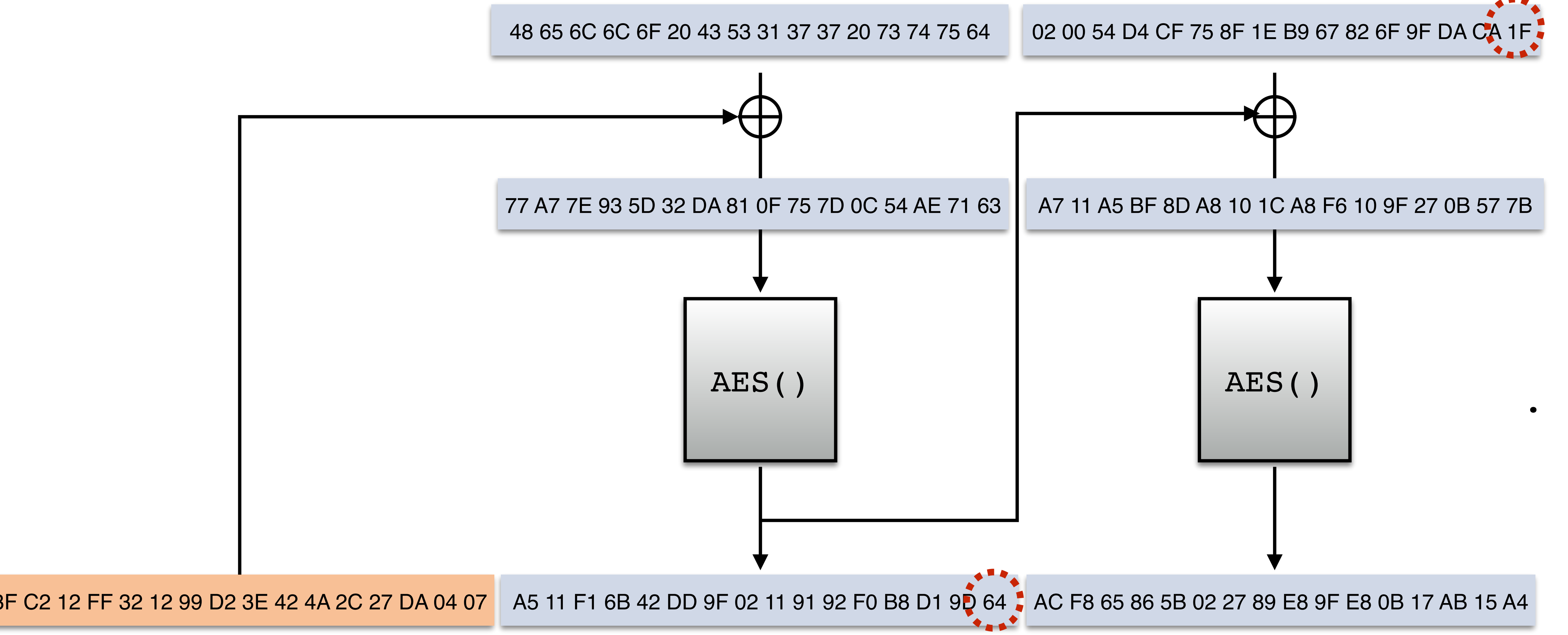
Initial goal: Learn last byte of a block.

- What will padding oracle say?
  - (still) `Padding_Error`!

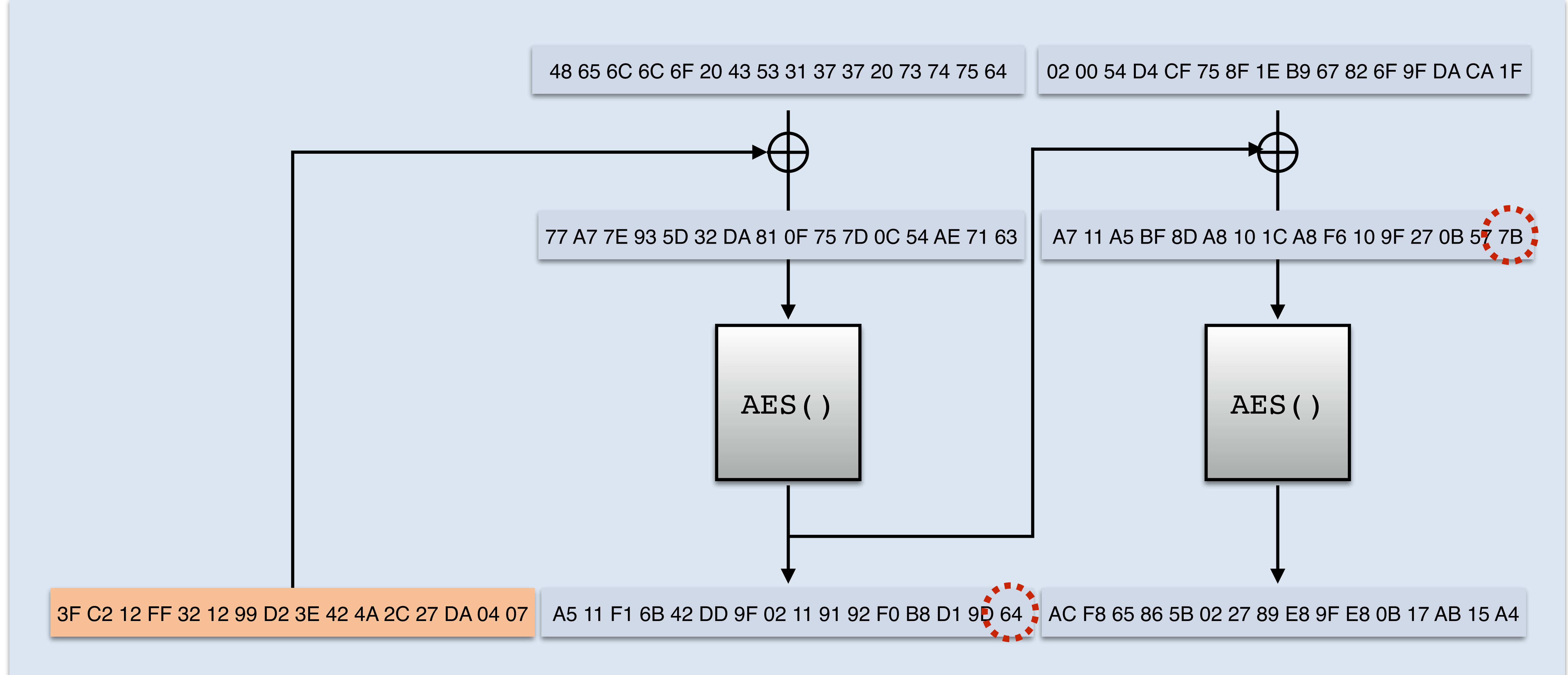


Initial goal: Learn last byte of a block.

- What will padding oracle say?
  - Padding is valid!
  - Infer target byte: 7A⊕(target-byte) = 0x01, so can solve and get target-byte=7B
  - Finally infer last plaintext byte is 7B⊕64=1F



Initial goal: Learn last byte of a block.

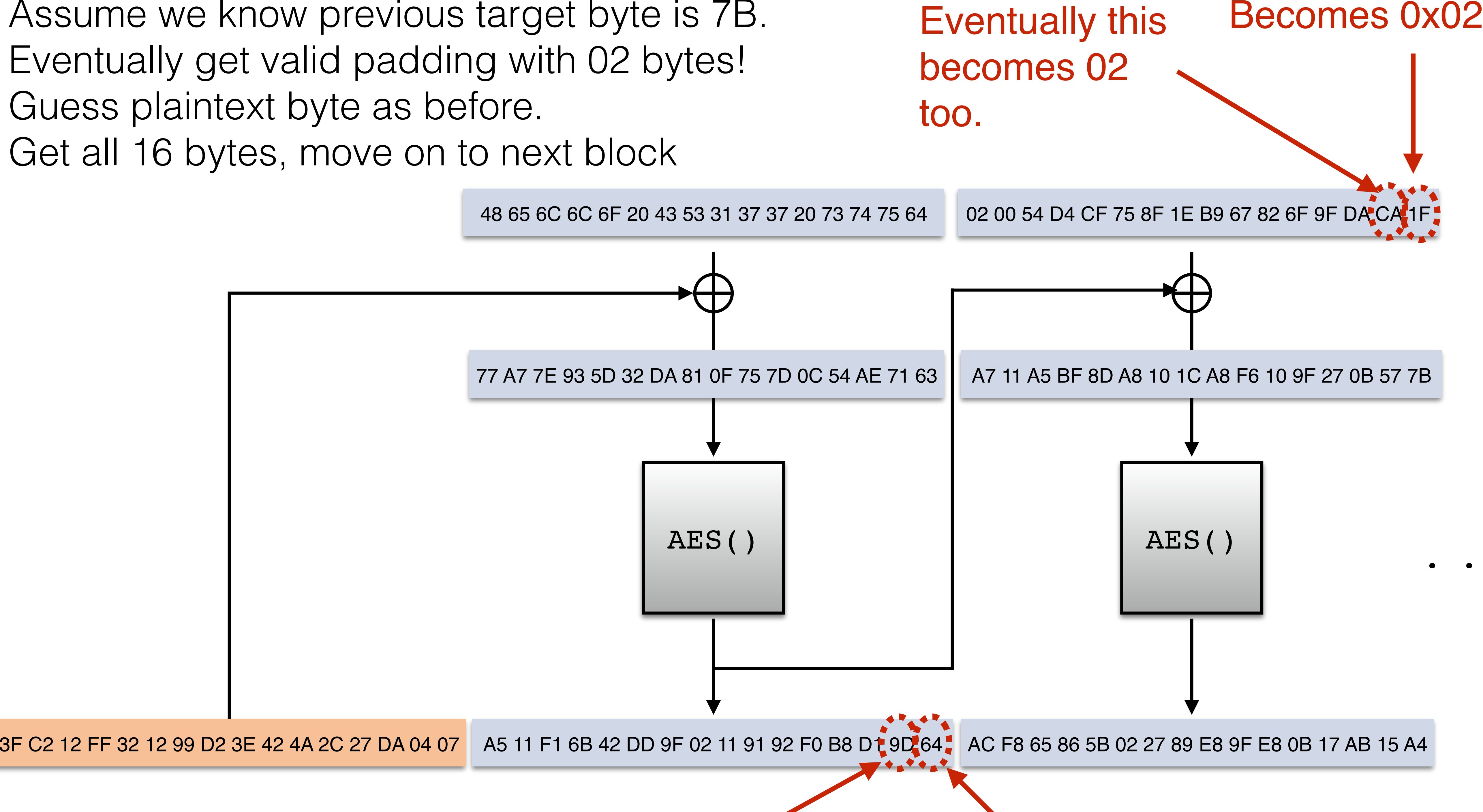


Upshot: Target byte of plaintext is `X`, and we change last byte of prior ciphertext block to $X \oplus 0x01$, then padding will be valid. Changing that byte to anything else will usually result in invalid padding.

Q: But how do we should we set that byte? **A:** Just try all 256 until one works.

Recovering more plaintext bytes

- Assume we know previous target byte is 7B.
- Eventually get valid padding with 02 bytes!
- Guess plaintext byte as before.
- Get all 16 bytes, move on to next block



Outline of Lecture 6

1. Message Authentication Codes (MACs)
2. Cryptographic Hash Functions
3. Authenticated Encryption (AE)
4. AE Case Study: Padding Oracle Attacks
5. **Public Key Encryption**
6. Digital Signatures

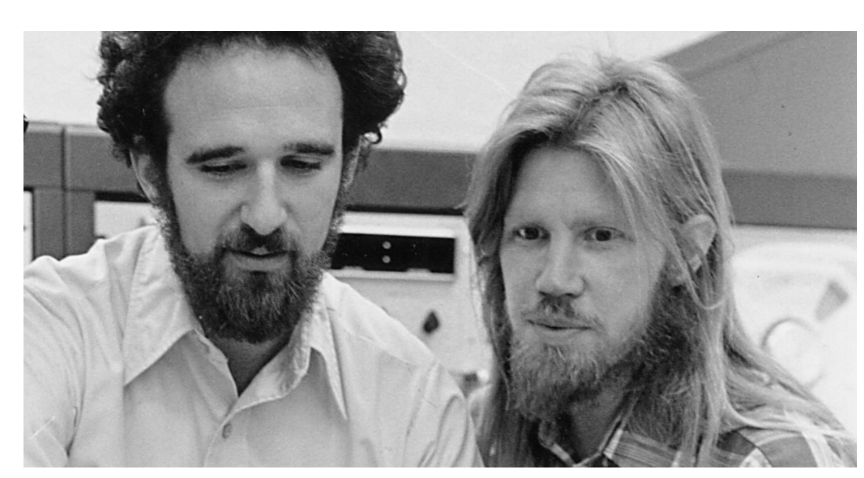
Motivation for Public-Key Cryptography

Basic question: If two people do not have pre-shared a key, is there any way they can send private messages in the presence of an attacker?

Pre-shared key? \ Security Goal	Confidentiality	Authenticity/Integrity
Yes ("Symmetric")	Symmetric Encryption (Stream Ciphers and Block Ciphers)	Message Authentication Codes (MACs)
No ("Asymmetric")	Public-Key Encryption (RSA) and Key Exchange (Diffie-Hellman)	Digital Signatures

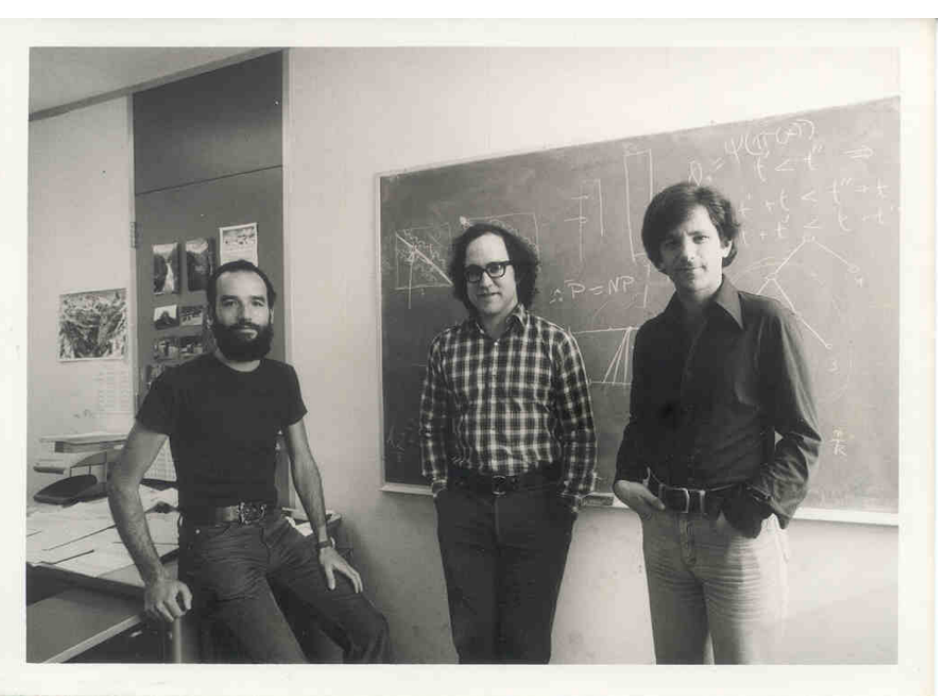
The Seed of Public-Key Cryptography

Basic question: If two people do not have pre-shared a key, is there any way they can send private messages in the presence of an attacker?



Diffie and Hellman
in 1976: **Yes!**

Turing Award, 2015



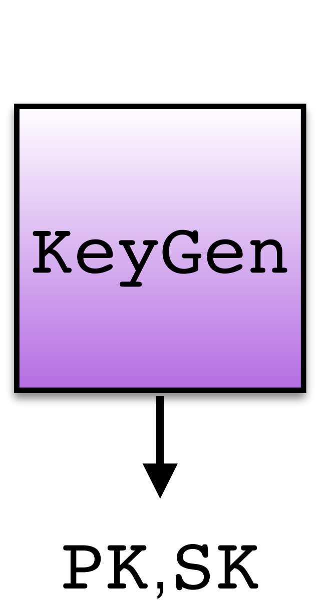
Rivest, Shamir, Adleman
in 1978: **Yes, differently!**

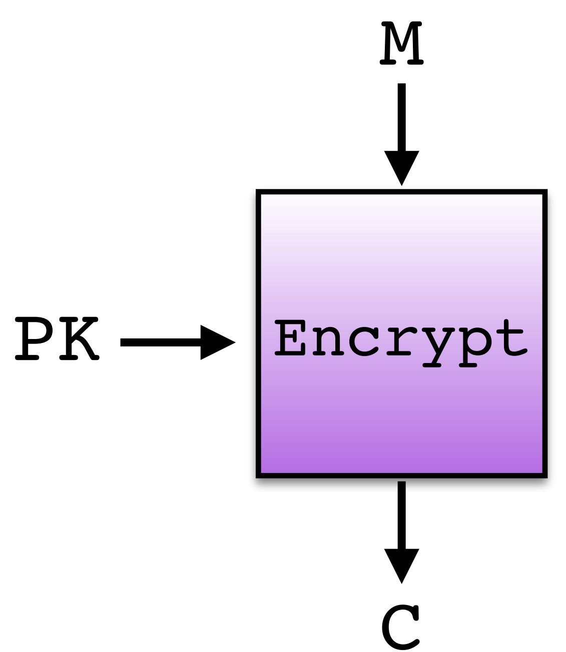
Turing Award, 2002

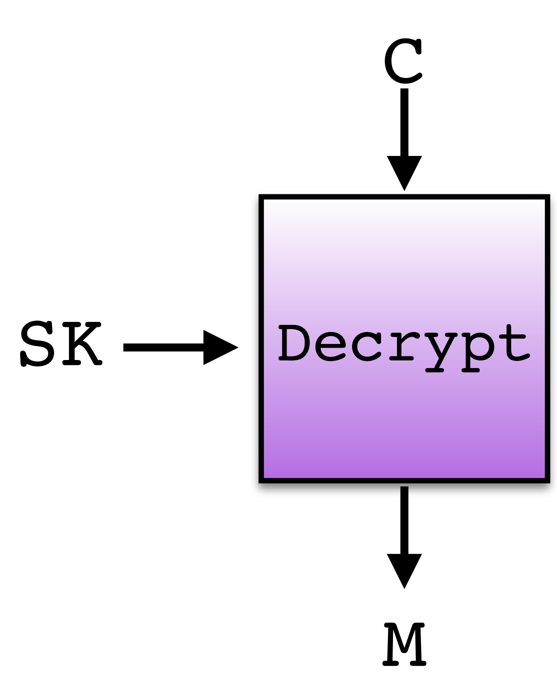
Cocks, Ellis, Williamson
in 1969, at GCHQ:
Yes…

Public-Key Encryption Schemes

A public-key encryption scheme consists of three algorithms **KeyGen**, **Encrypt**, and **Decrypt**







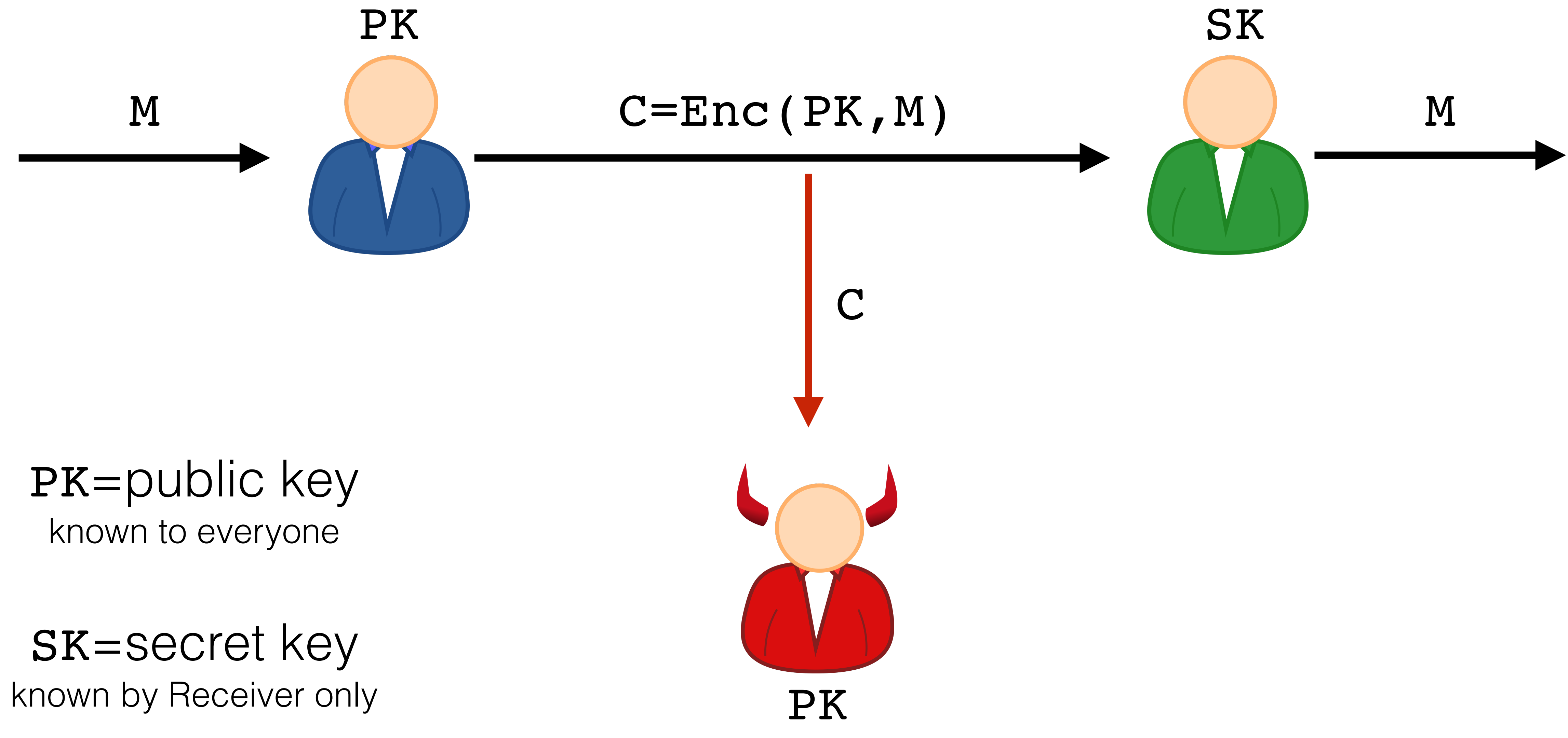
KeyGen: Outputs two keys. PK published openly, and SK kept secret.

Encrypt: Uses PK and M to produce a ciphertext C.

Decrypt: Uses SK and C to recover M.

Public-Key Encryption

- **Goal**: Confidentiality against attacker who knows public key `PK` and ciphertext `C`



Plain RSA Public-Key Encryption: Key Generation

```
RSA-KeyGen(n)
1. Choose random n-bit primes p and q
2. N = pq; phi(N) = (p-1)(q-1)
3. Set e to default value
4. Find d such that ed == 1 (mod phi(N))
5. Output
     PK = (N,e)
     SK = (N,d)
```

- `n` is security level (longer primes => more secure)
- `e` is usually set to 65537 (ask me why later)
- Finding `d` uses extended Euclidean alg (from CS 271)
- Omitted: How primes are chosen
- **Example:**

  `p=5`, `q=11`, `N=55`

  `e=3`, `d=27`

Plain RSA Public-Key Encryption: Encrypt/Decrypt

```
RSA-KeyGen(n)
1. Choose random n-bit primes p and q
2. N = pq; phi(N) = (p-1)(q-1)
3. Set e to default value
4. Find d such that ed == 1 (mod phi(N))
5. Output
     PK = (N,e)
     SK = (N,d)
```

```
RSA-Encrypt(PK, M)
1. Parse PK as (N,e)
2. Compute C = M^e (mod N)
3. Output C
```

```
RSA-Decrypt(SK, C)
1. Parse SK as (N,d)
2. Compute M = C^d (mod N)
3. Output M
```

- In CS 271, it is proved that decryption recovers message correctly
- **Intuition**: Computing "`e`-th roots modulo `N`" is hard if you can't factor `N`
- **Warning**: Not secure as stated; needs further protections

Factoring Records and RSA Key Length

- Factoring N allows recovery of secret key
- Challenges posted publicly by RSA Laboratories

Bit-length of N	Year
400	1993
478	1994
515	1999
768	2009
795	2019

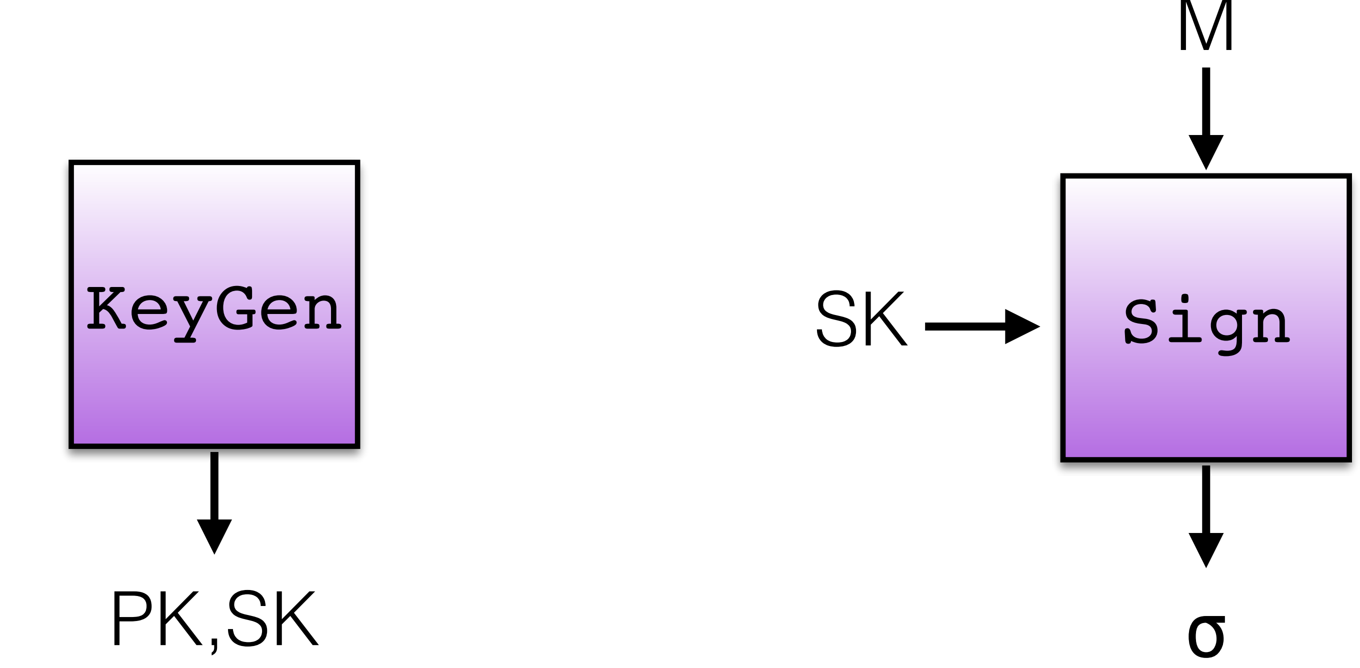
- Recommended bit-length today: 2048
- Note that fast algorithms force such a large key.
  - 512-bit N defeats naive factoring

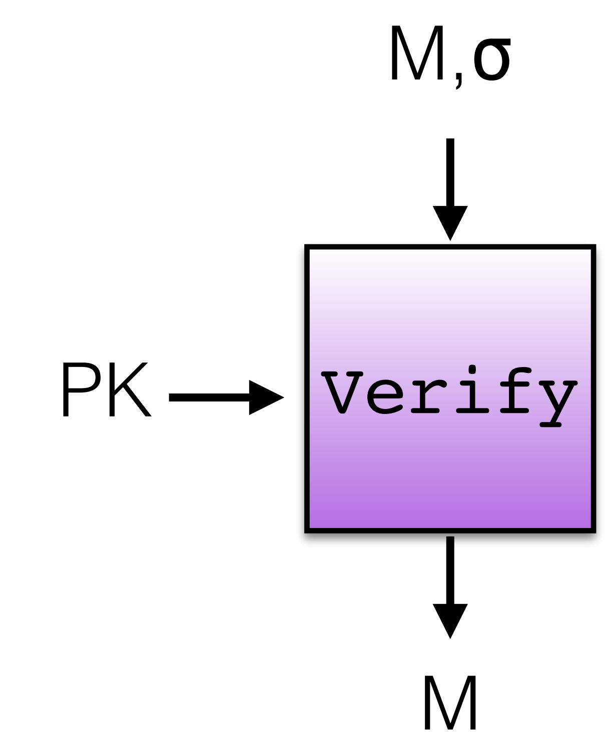
Outline of Lecture 6

1. Message Authentication Codes (MACs)
2. Cryptographic Hash Functions
3. Authenticated Encryption (AE)
4. AE Case Study: Padding Oracle Attacks
5. Public Key Encryption
6. **Digital Signatures**

Digital Signatures Schemes

A <u>digital signature scheme</u> consists of three algorithms **`KeyGen`**, **`Sign`**, and **`Verify`**



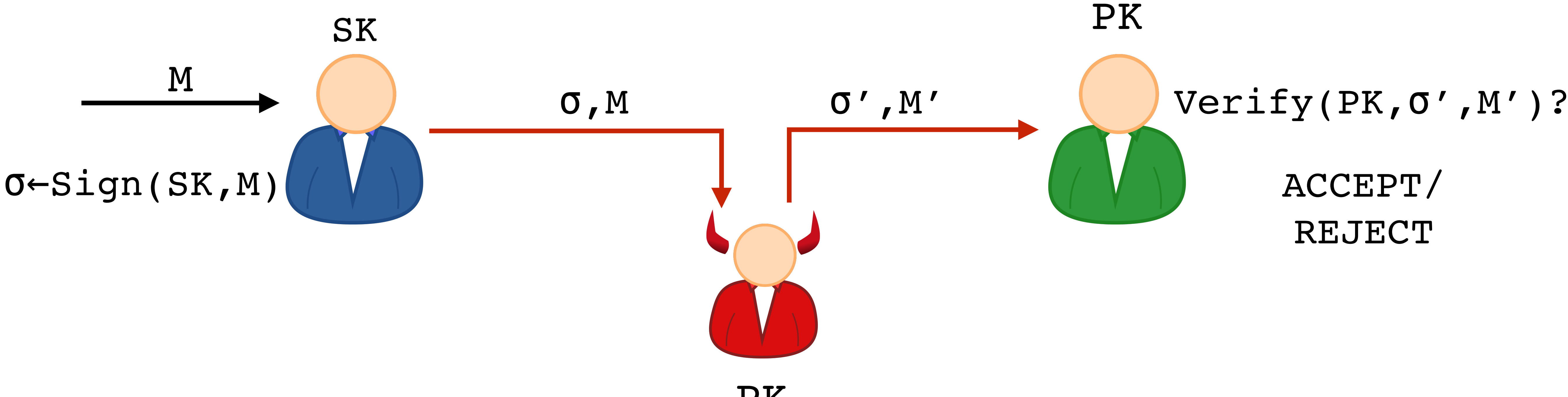


<u>`KeyGen`</u>: Outputs two keys. PK published openly, and SK kept secret.

<u>`Sign`</u>: Uses SK to produce a "signature" **σ** on M.

<u>`Verify`</u>: Uses PK to check if signature **σ** is valid for M.

Digital Signature Security Goal: Unforgeability



Scheme satisfies **unforgeability** if it is unfeasible for Adversary (who knows `PK`) to fool Bob into accepting `M'` not previously sent by Alice.

Plain RSA Signatures

```
RSA-KeyGen(n)
1. Choose random n-bit primes p and q
2. N = pq; phi(N) = (p-1)(q-1)
3. Set e to default value
4. Find d such that ed == 1 (mod phi(N))
5. Output
     PK = (N,e)
     SK = (N,d)
```

```
RSA-Sign(SK,M)
1. Parse PK as (N,d)
2. Compute σ = M^d (mod N)
3. Output σ
```

```
RSA-Verify(PK,σ,M)
1. Parse PK as (N,e)
2. Compute V = σ^e (mod N)
3. Accept if V==M
```

- Note similarity to encryption
- **Intuition**: Computing "**e**-th roots modulo **N**" is hard if you can't factor **N**
- **Warning**: Not secure as stated; needs further protections

"Plain" RSA Weaknesses

Assume $e=3$.

```
RSA-Sign(SK,M)

1. Parse PK as (N,d)
2. Compute σ = M^d (mod N)
3. Output σ
```

```
RSA-Verify(PK,σ,M)

1. Parse PK as (N,e)
2. Compute V = σ^e (mod N)
3. Accept if V==M
```

To forge a signature on message M: Find number σ such that $\sigma^3 = M \bmod N$

$M=1$ weakness: If $M=1$ then it is easy to forge. Take $\sigma=1$:

$$\sigma^3 = 1^3 = 1 = M \bmod N$$

Cube-M weakness: If M is a *perfect cube* then it is easy to forge.
Just take $\sigma = M^{1/3}$:, i.e. the usual cube root of M:

Example: To forge on $M=8$, which is a perfect cube, set $\sigma=2$.

$$\sigma^3 = 2^3 = 8 = M \bmod N$$

(Intuition: If cubing does not "wrap modulo N", then it is easy to un-do.)

Further "Plain" RSA Weaknesses

Assume $e=3$.

```
RSA-Sign(SK,M)

1. Parse PK as (N,d)
2. Compute σ = M^d (mod N)
3. Output σ
```

```
RSA-Verify(PK,σ,M)

1. Parse PK as (N,e)
2. Compute V = σ^e (mod N)
3. Accept if V==M
```

To forge a signature on message M: Find number σ such that $\sigma^3 = M \bmod N$

***Malleability* weakness:** If σ is a valid signature for M, then it is easy to forge a signature on $M' = 8M \bmod N$.

Given (M,σ), compute $\sigma' = (2 * \sigma \bmod N)$

Then Verify((N,3),M',σ') checks:

$$(\sigma')^3 = (2 * \sigma \bmod N)^3 = (2^3 * \sigma^3 \bmod N) = (2^3 * M \bmod N) = 8M \bmod N$$

Further "Plain" RSA Weaknesses

Assume `e=3`.

```
RSA-Sign(SK,M)

1. Parse PK as (N,d)
2. Compute σ = M^d (mod N)
3. Output σ
```

```
RSA-Verify(PK,σ,M)

1. Parse PK as (N,e)
2. Compute V = σ^e (mod N)
3. Accept if V==M
```

To forge a signature on message M: Find number σ such that $\sigma^3 = M \bmod N$

***Backwards signing* weakness:** Generate *some* valid signature by picking σ′ first, and then defining $M' = (\sigma'^3 \bmod N)$

Then `Verify((N,3),M′,σ′)` checks:

$$(\sigma')^3 = (M' \bmod N)$$

Further "Plain" RSA Weaknesses

Assume e=3.

```
RSA-Sign(SK,M)

1. Parse PK as (N,d)
2. Compute σ = M^d (mod N)
3. Output σ
```

```
RSA-Verify(PK,σ,M)

1. Parse PK as (N,e)
2. Compute V = σ^e (mod N)
3. Accept if V==M
```

To forge a signature on message M: Find number σ such that $\sigma^3 = M \bmod N$

Summary:

- Plain RSA Signatures allow several types of forgeries
- It was sometimes argued that these forgeries aren't important: If M is english text, then M' is unlikely to be meaningful for these attacks
- But often they are damaging anyway

RSA Signatures with Encoding

- Select a function `Encode` that "randomizes" messages
- Sign/Verify `R=Encode(M)` instead of `M`

```
RSA-Sign(SK,M)
1. Parse PK as (N,d)
2. R = Encode(M)
3. Compute σ = R^d (mod N)
4. Output σ
```

```
RSA-Verify(PK,σ,M)
1. Parse PK as (N,e)
2. R = Encode(M)
3. Compute V = σ^e (mod N)
4. Accept if V==R
```

- For example, signing `M=1` now requires signing `Encode(1)` instead of `1`

Encoding must be chosen with extreme care.

The End