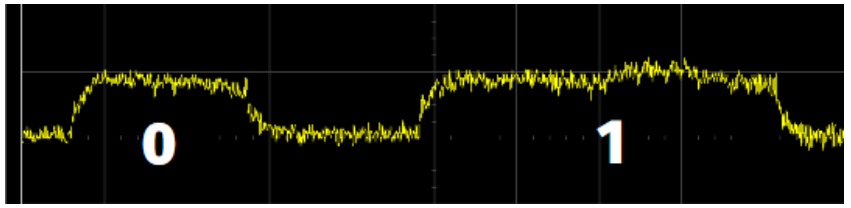
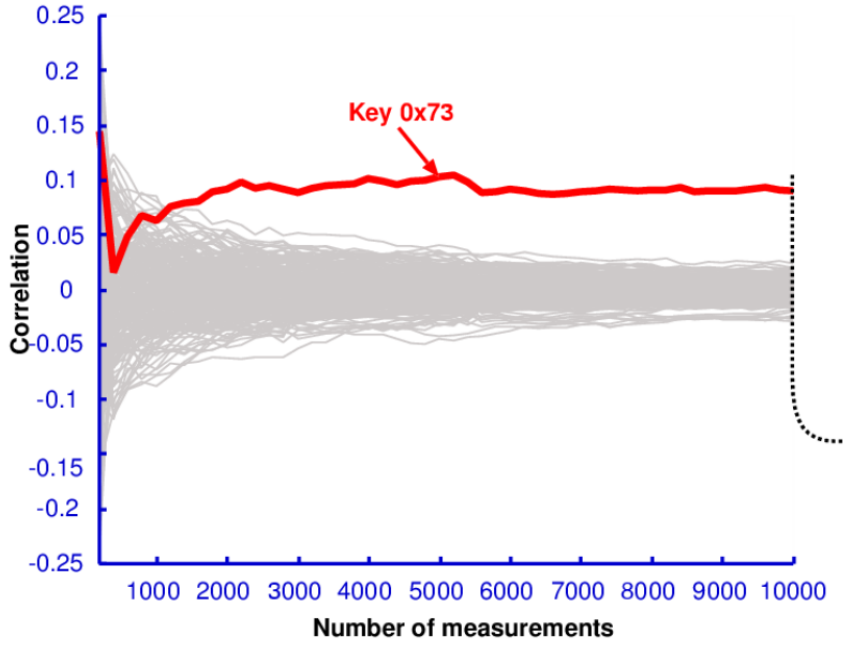
Lecture 11: Padding oracles

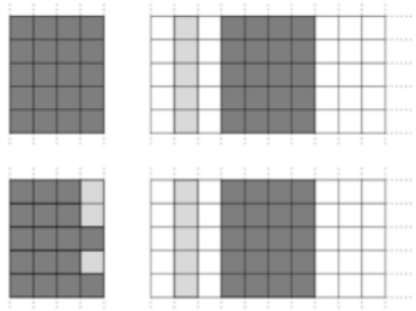
Last week: Power analysis and timing attacks

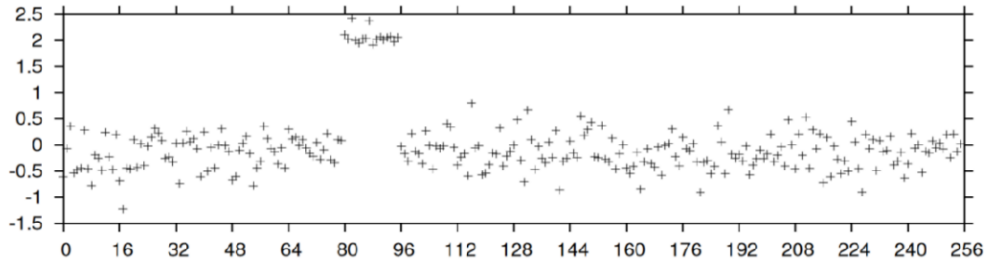
Power analysis (SPA, DPA, template)





Timing attacks (cache: prime+probe, evict+time; network)



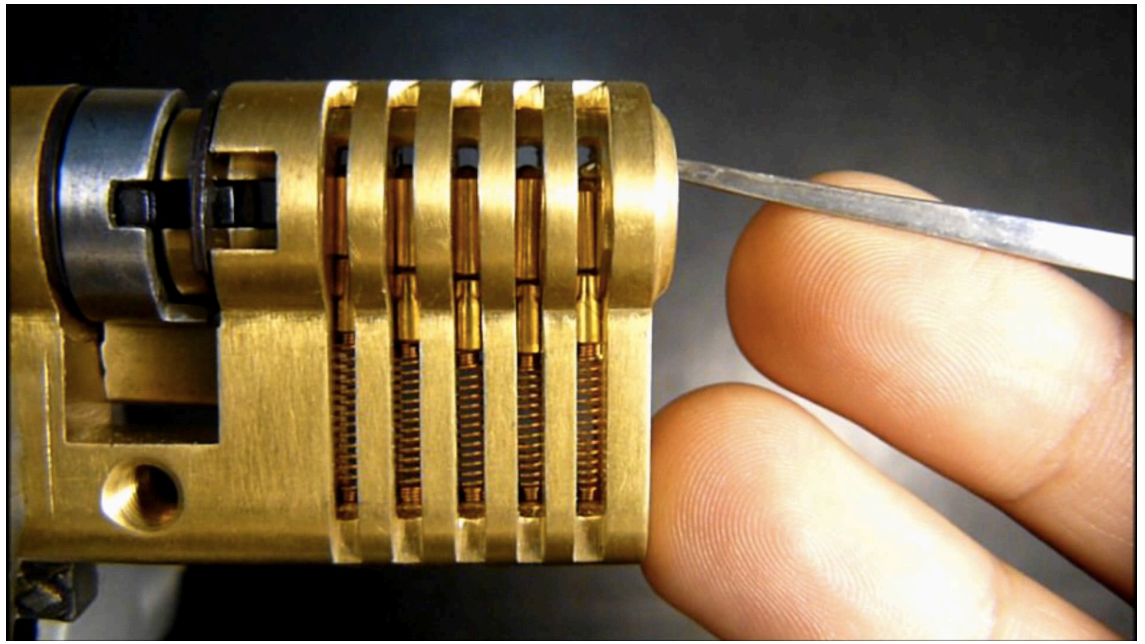


Today: Padding Oracle Attacks

- Last week: Attacks on *AES*
  - Exploit knowledge of power, timing
- Today: Attacks on *modes* of AES: CBC mode (plus other ingredients)
  - Exploit knowledge of error messages

Divide and conquer

- Attacks follow a *divide and conquer* approach: break 1 byte at a time
- For each byte, simply *guess* all 256 values *and check* which one works
- (Think: how you see crypto broken in any Hollywood movie)

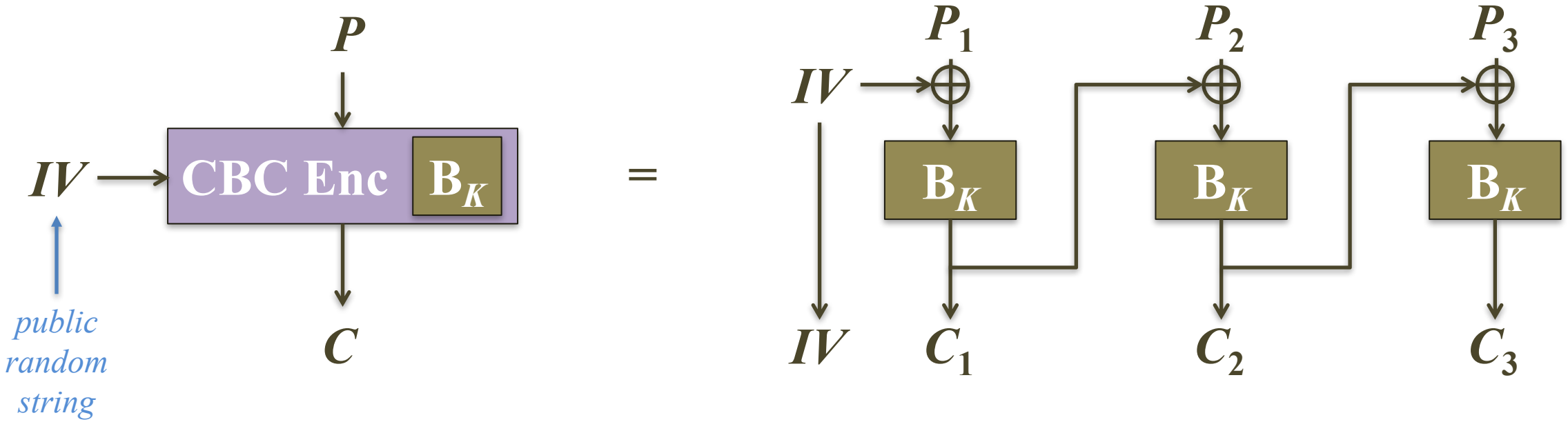


Padding Oracles [Vaudenay 2002]

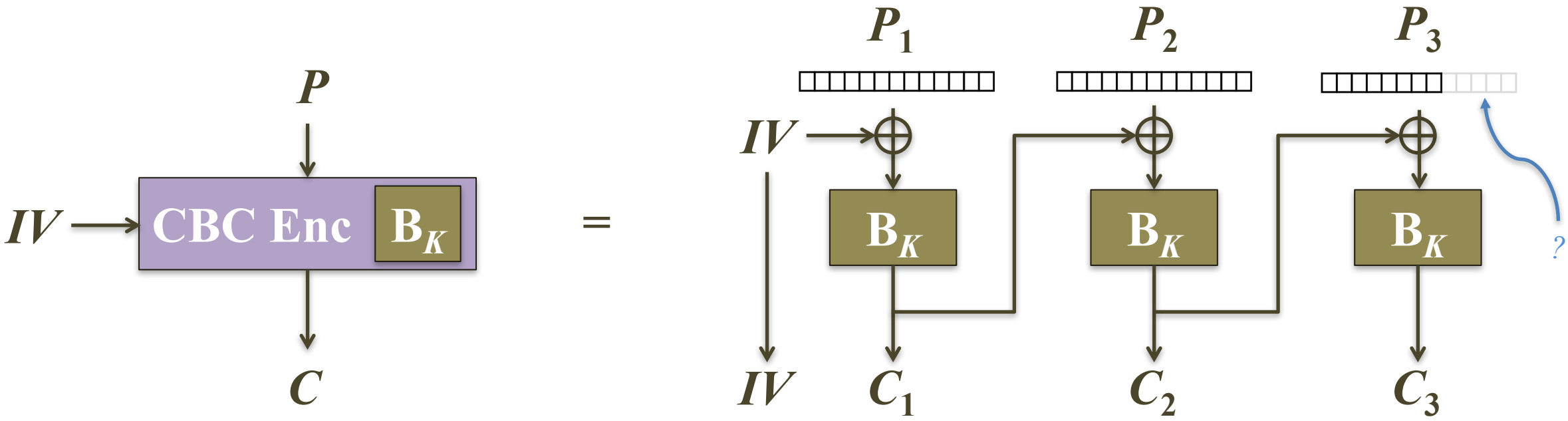
- Main idea: Exploit error messages for different kinds of malformed input to recover the plaintext

- Four building blocks:
  1. CBC Mode
  2. PKCS#7 Padding
  3. Authenticate-then-Encrypt
     1. How to authenticate???
  4. Error messages

Building block 1: CBC mode (encryption)



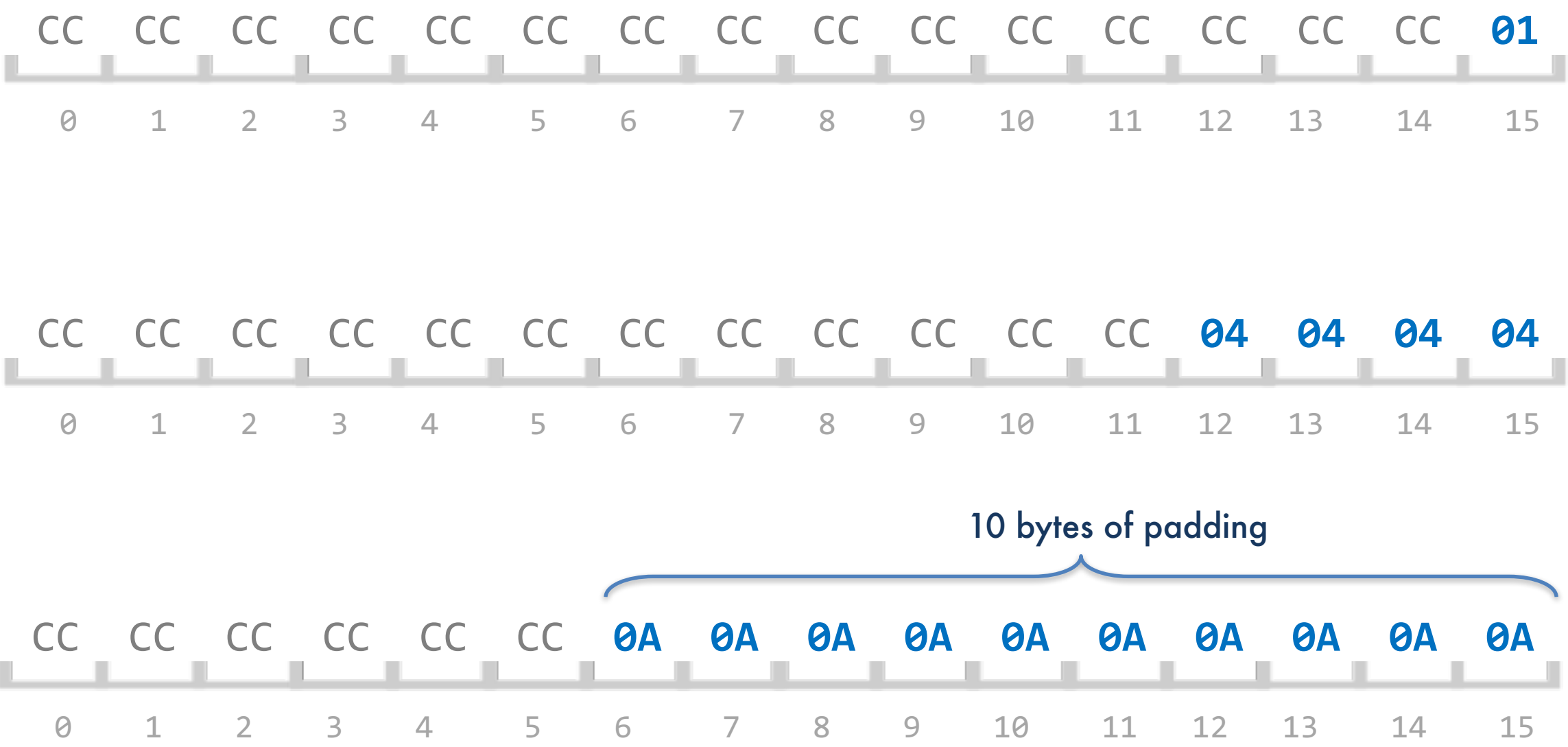
Recall: CBC mode needs **padding**



For today: Length of *P* = any number of bytes

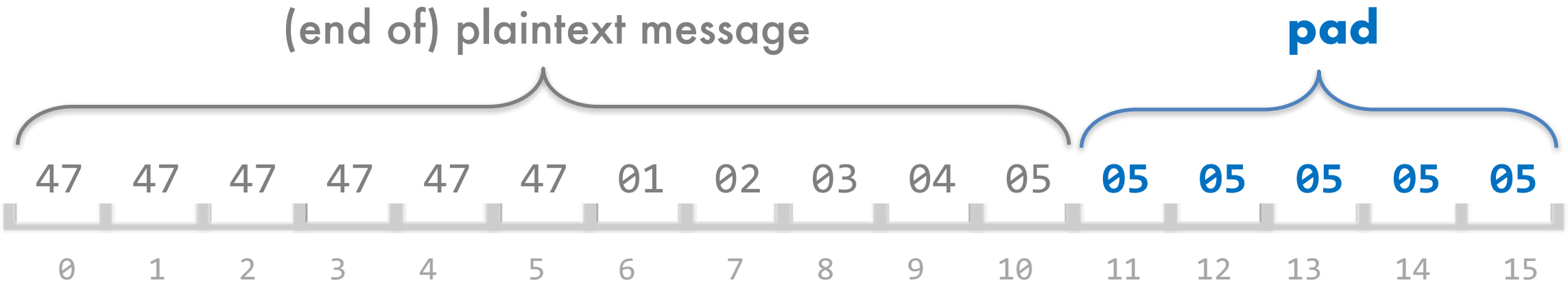
- Will not "split" bytes
- Might not be multiple of 16

Building block 2: PKCS #7 padding

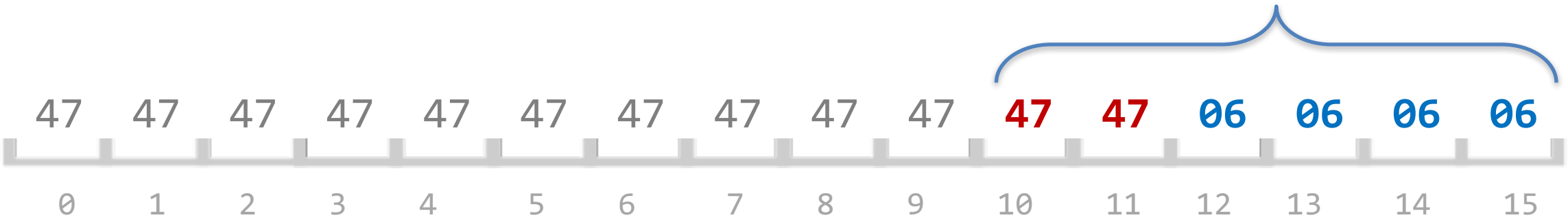


PKCS #7 padding

Padding adds N whole bytes, each of value N



What if padding is invalid?



Return error message "Invalid Padding"

(Building block 2 for padding oracle attacks)

Building block 3: Authenticate-Then-Encrypt

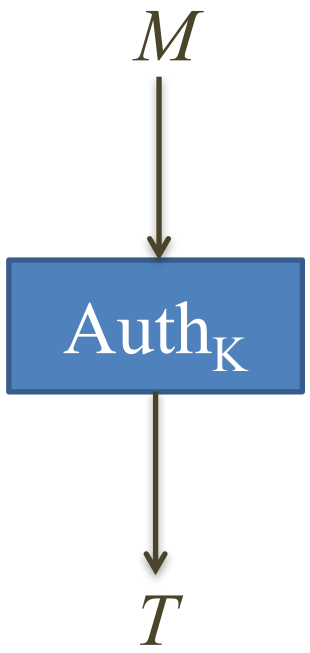
But first…

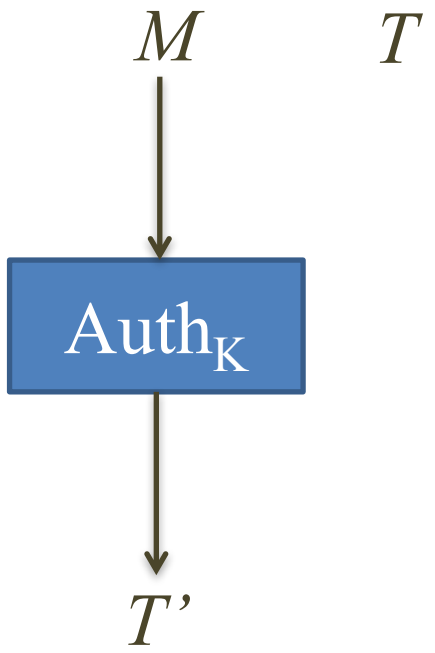
Building block 3.5: Authenticate ???

Sneak preview to something you'll talk about next week/month.

Building block 3.5: Authenticate

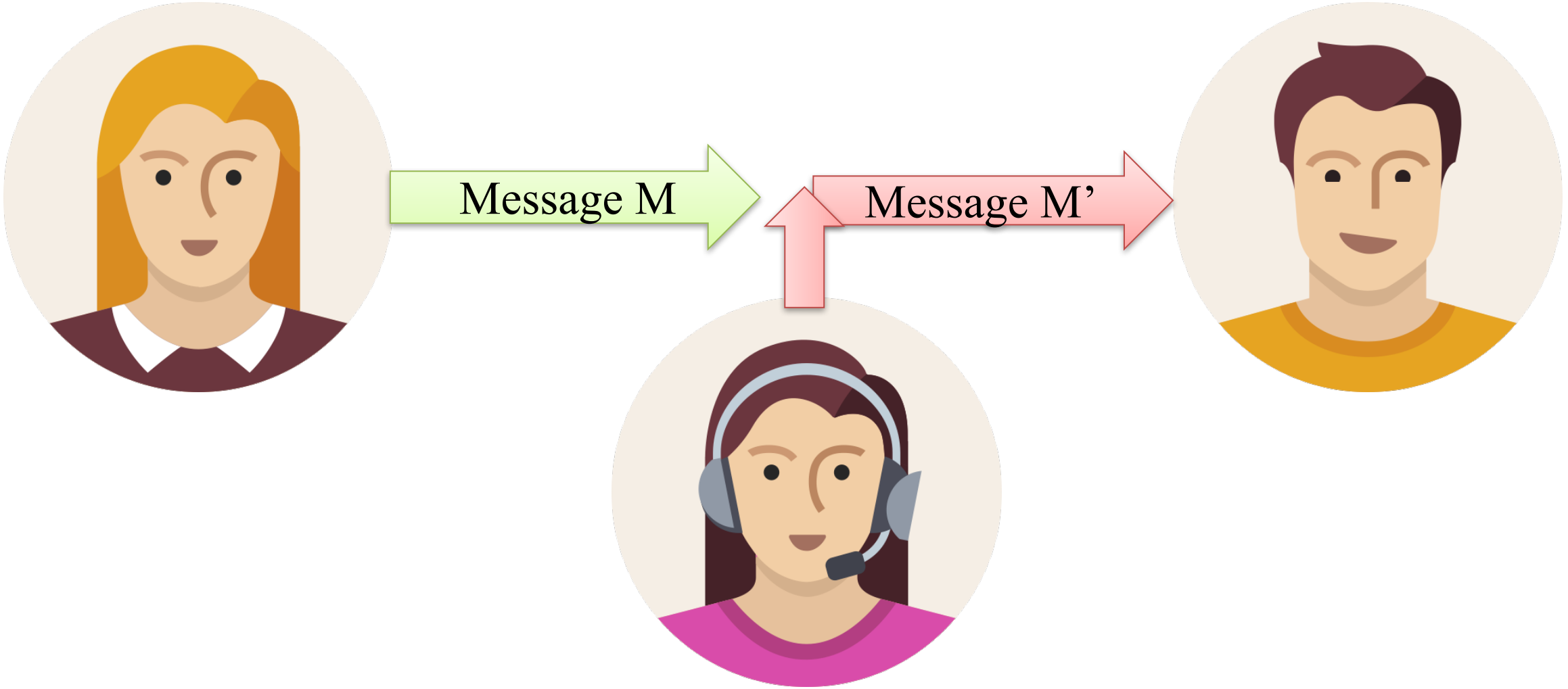
What if Mallory replaces our ciphertext on the wire?





$T = T'$?

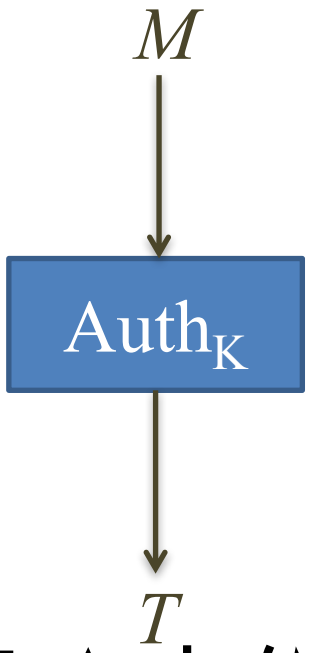
Forget confidentiality. What about authenticity?



Building block 3.5: Authenticate

Imagine the following (keyed) authentication mechanism:

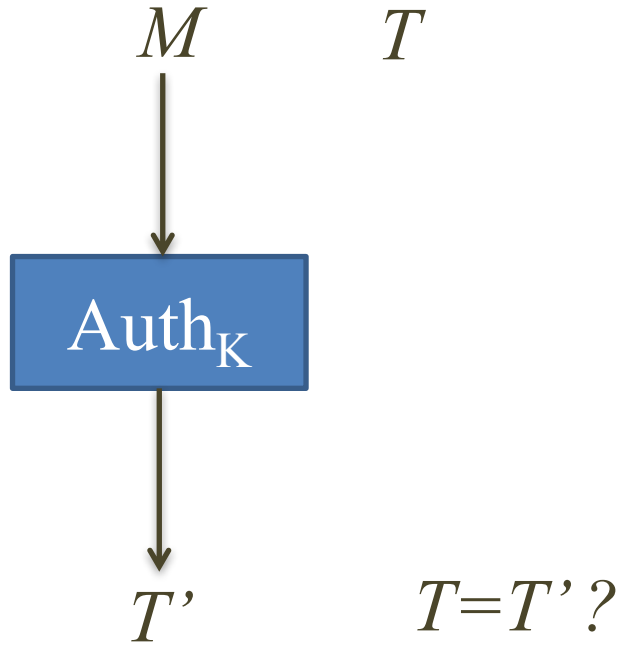
Alice wants to send M.



Computes $T=Auth_K(M)$

Sends (M,T)

Bob receives (M,T)

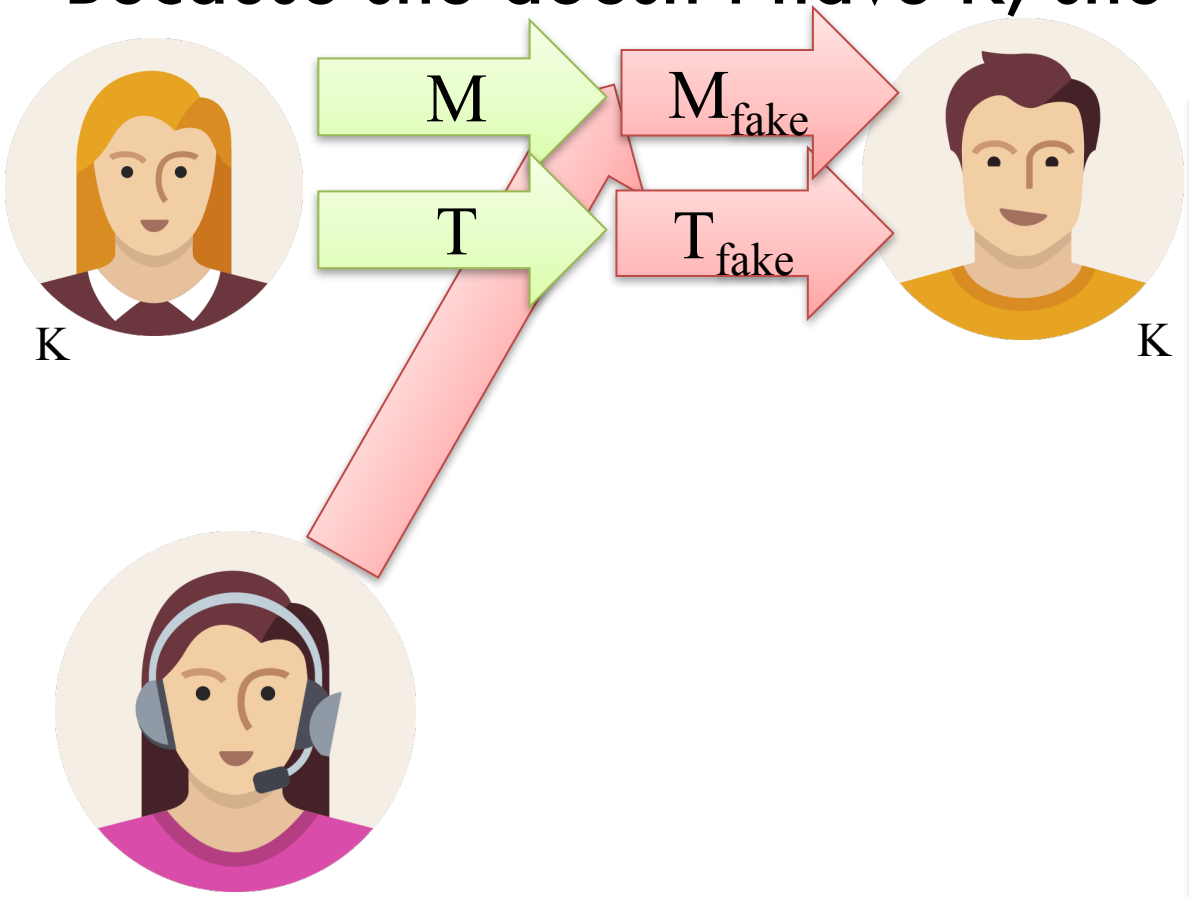


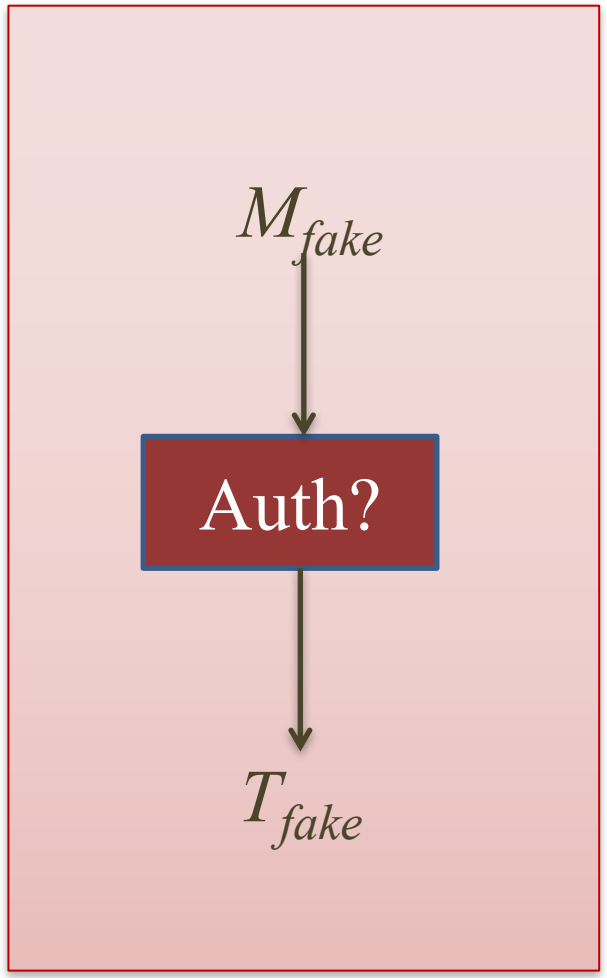
Computes $T'=Auth_K(M)$

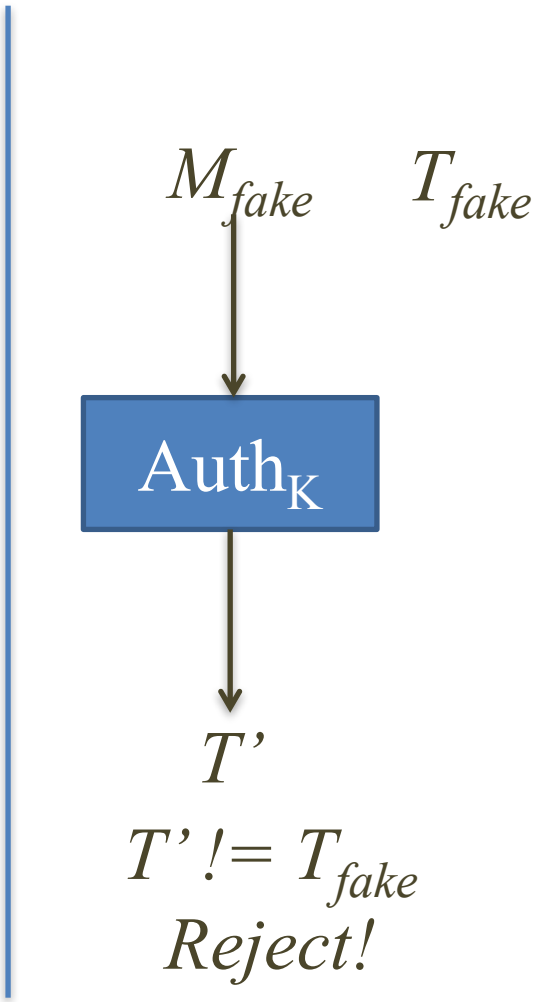
If T = T', know sender of M had K

Building block 3.5: Authenticate

Because she doesn't have K, she can't authenticate new messages…

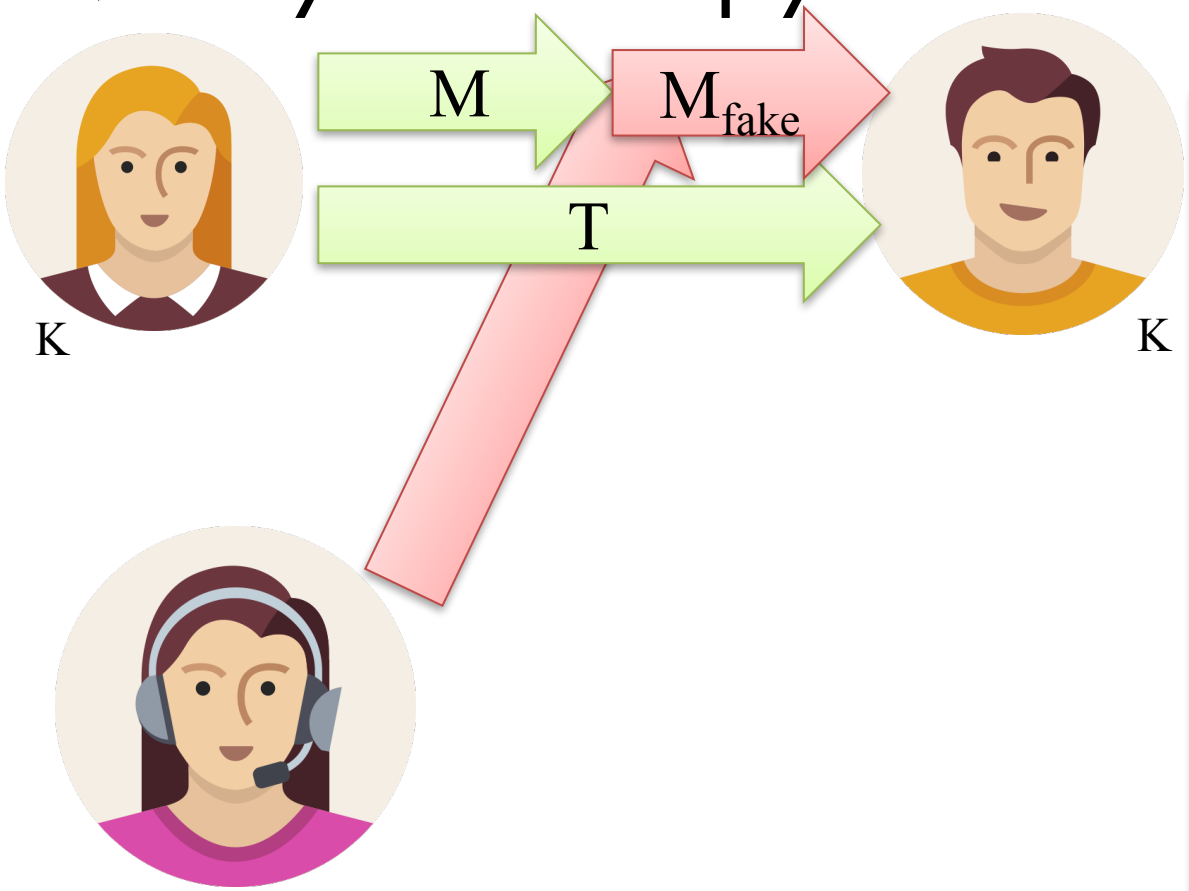


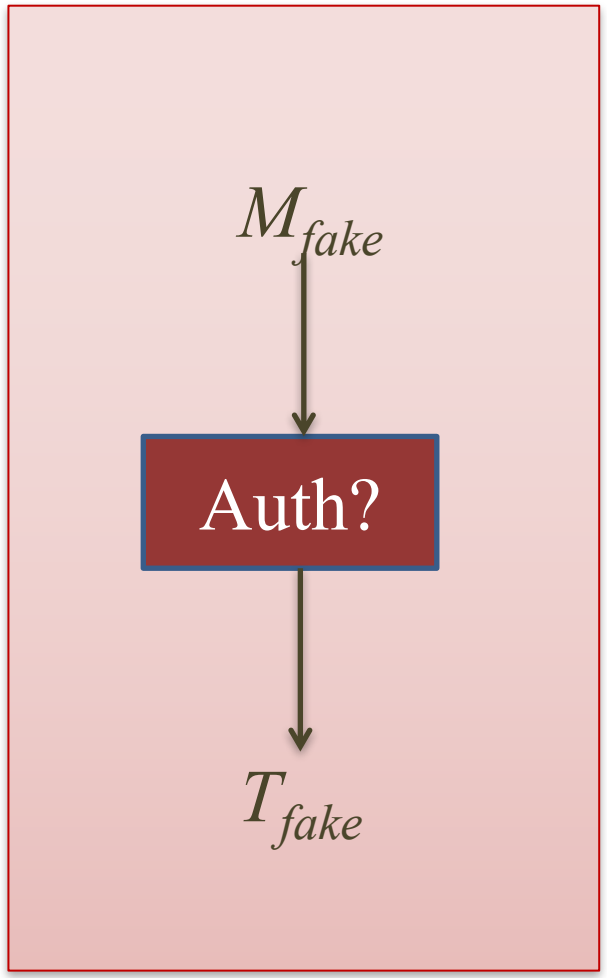


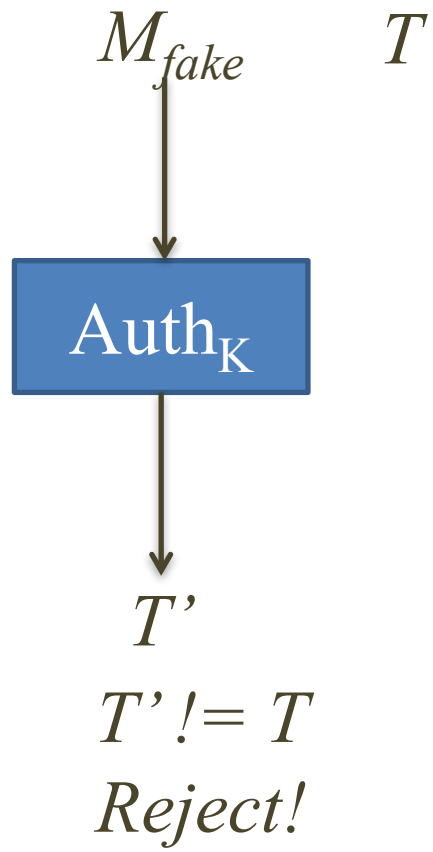


Building block 3.5: Authenticate

Mallory cannot simply reuse the existing tag for a different message…

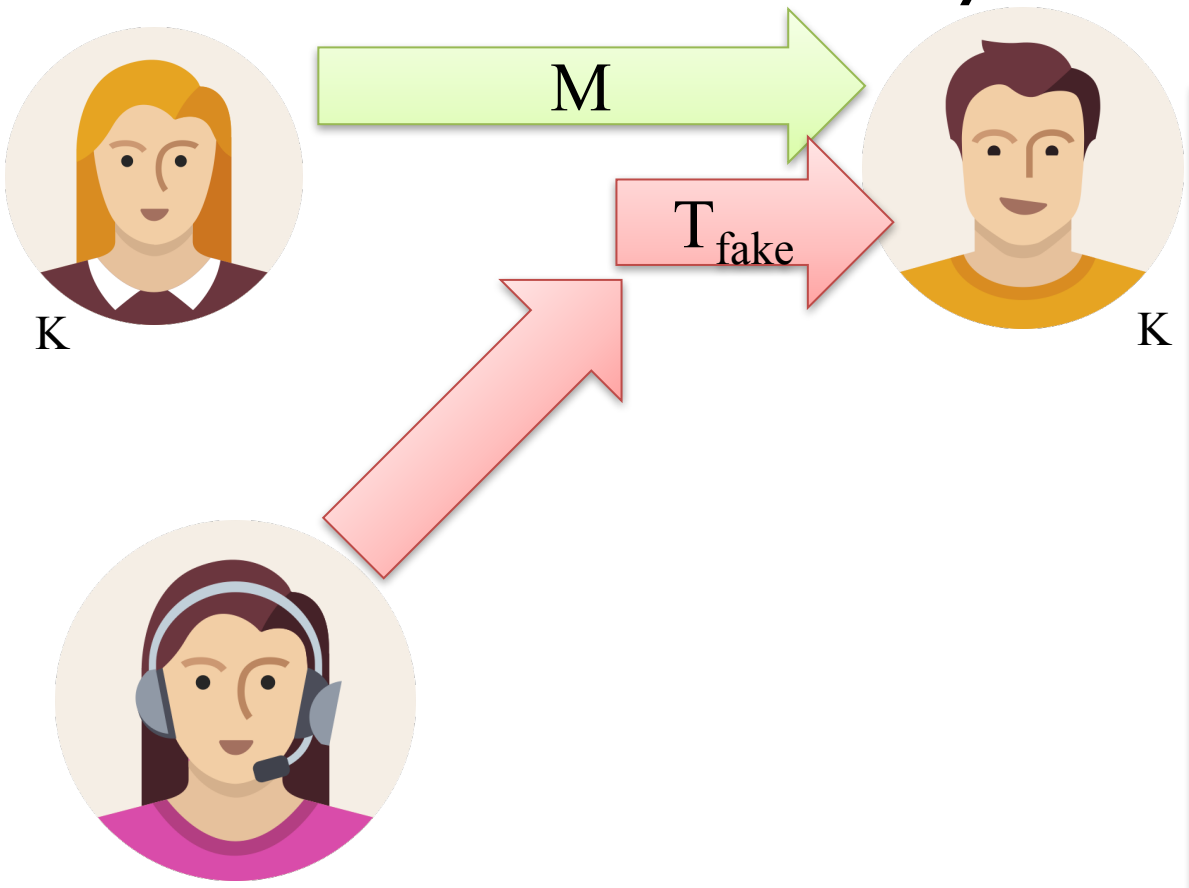


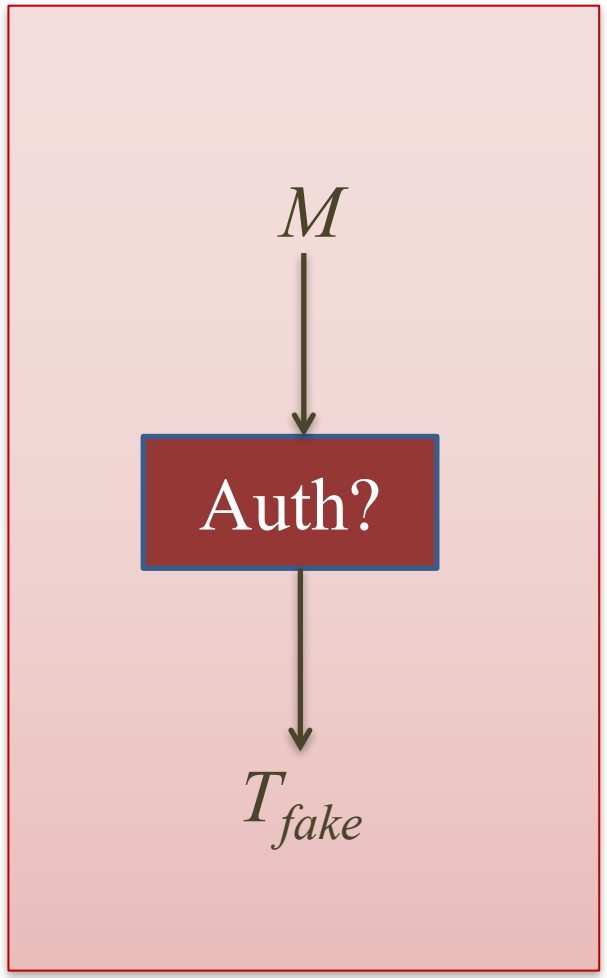


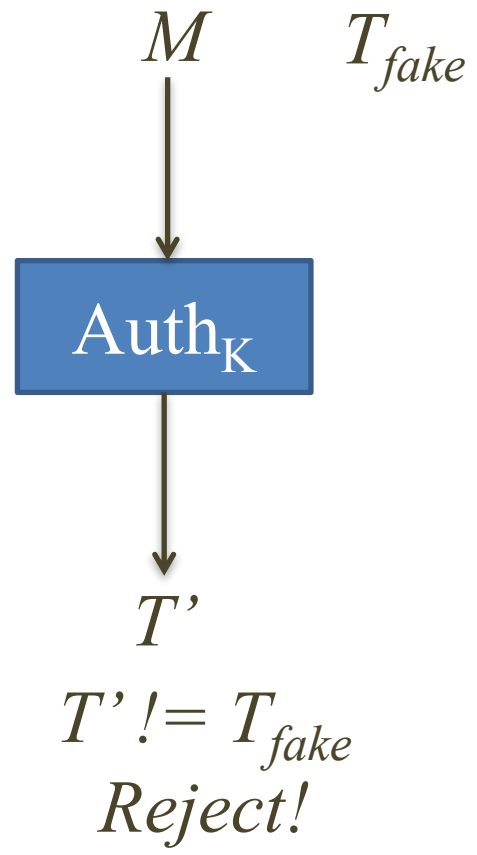


Building block 3.5: Authenticate

Even if she knows M, Mallory can't generate new T for it without knowing K…

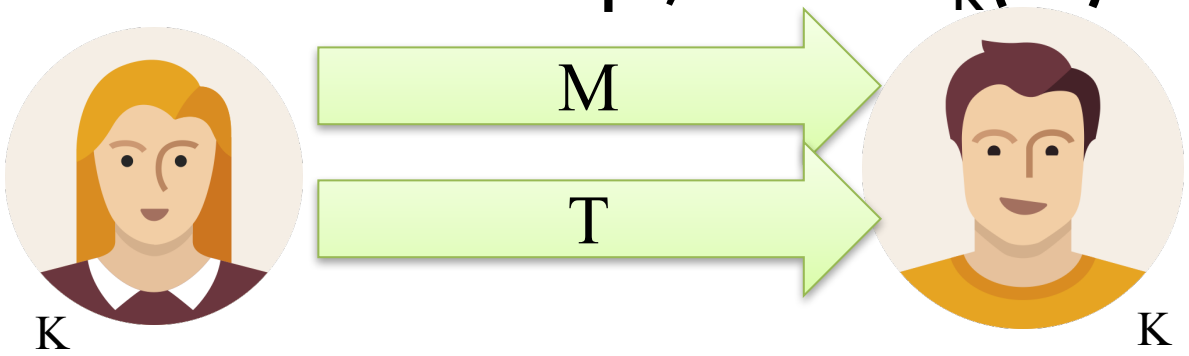


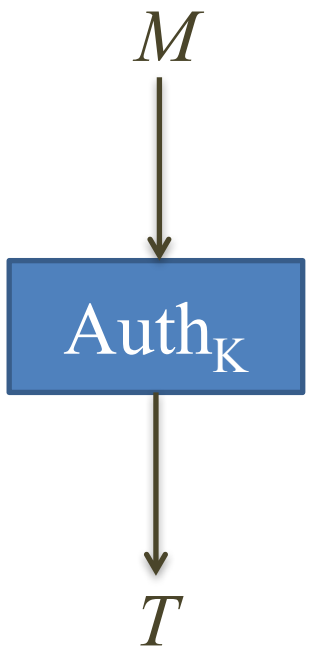


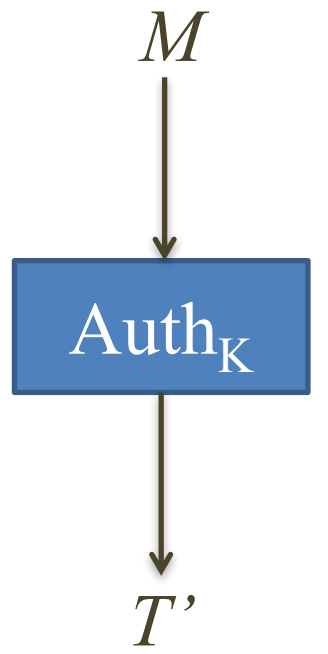


Building block 3.5: Authenticate

For Bob to accept, $T=Auth_K(M)$







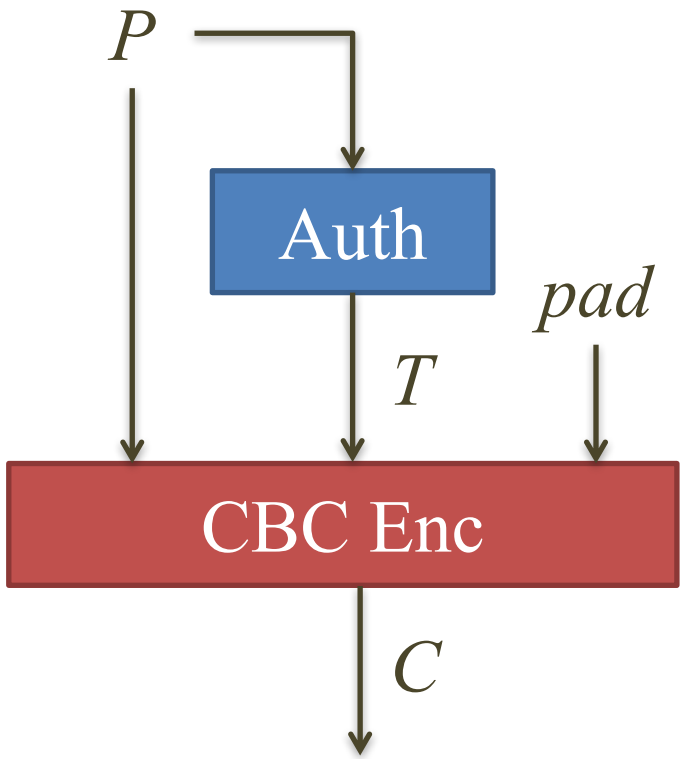
$T' == T$

Accept

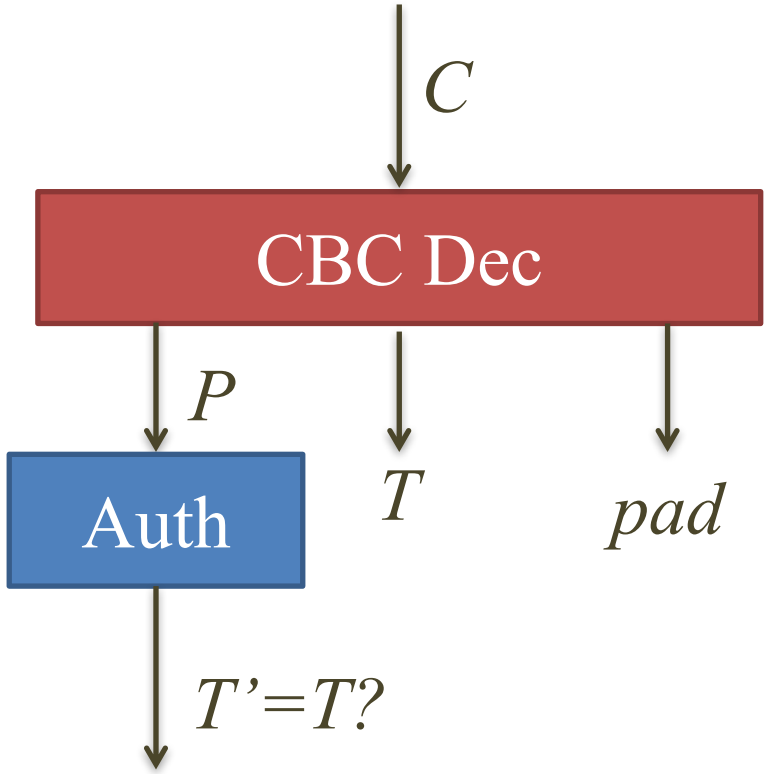
Building block 3: Auth-then-Encrypt (hmmm)

Consider the following scheme for encrypting and authenticating plaintext P:

1. Let T = Auth(P)
2. Let pad = PKCS7(P || T) (|| is concatenation)
3. Return C = CBC_Enc(P || T || pad)



Decryption-then-verification (hmmm)



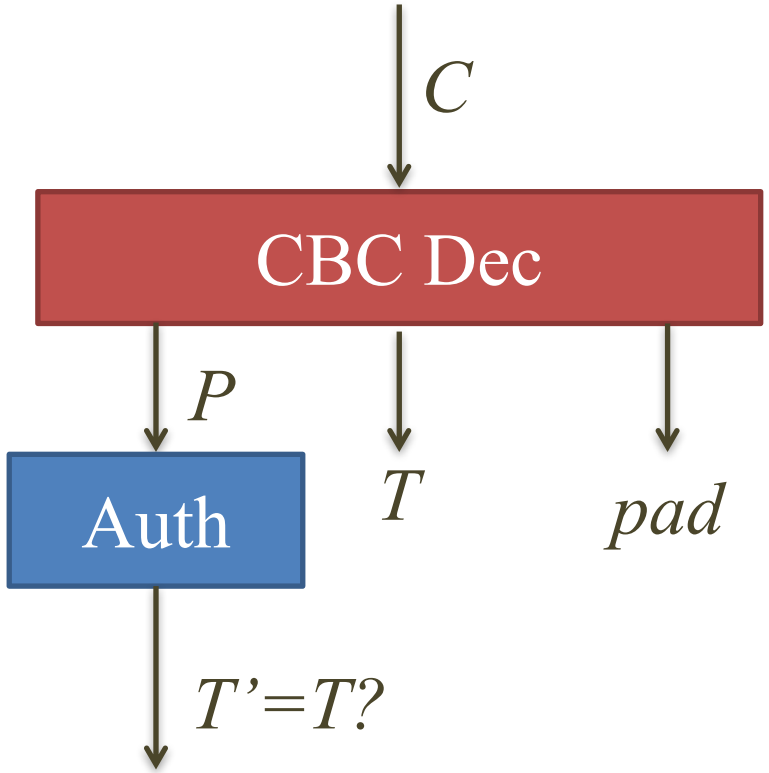
Decrypting/authenticating ciphertext C:

1. Let (P, T, pad) = CBC_Dec(C)

2. Let T′ = Auth(P)
3. Check whether T′=T

Valid plaintext

Building block 4: Error codes



Decrypting/authenticating ciphertext C:

1. Let (P, T, pad) = CBC_Dec(C)

Invalid Padding

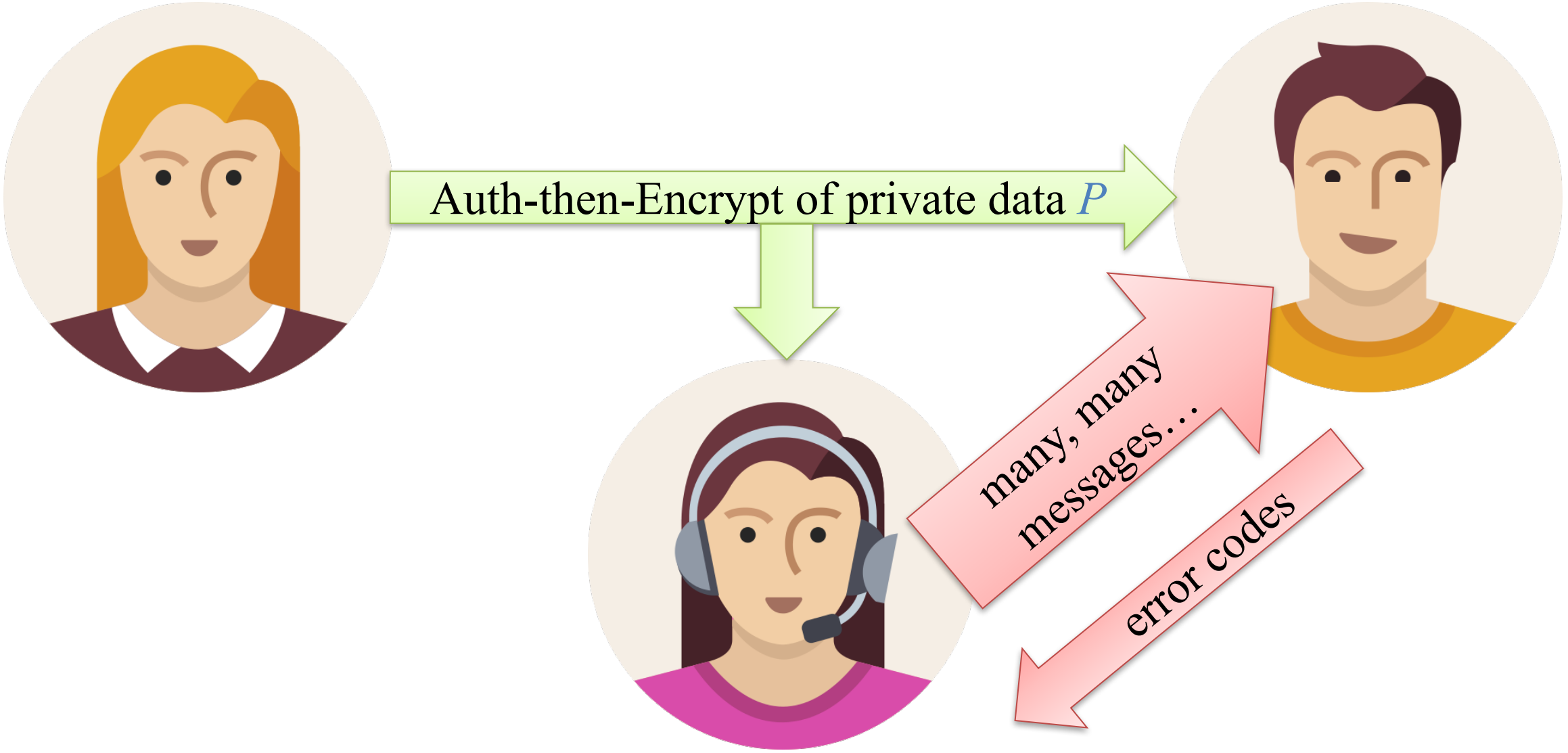
2. Let T′ = Auth(P)
3. Check whether T′=T

Exploit these distinct error messages to recover P

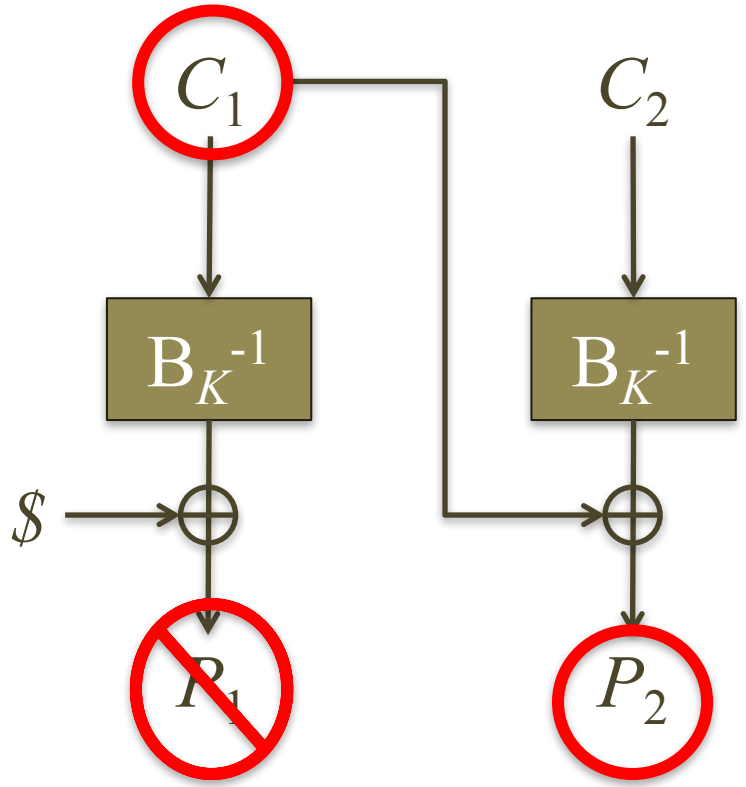
Reject Auth

Valid plaintext

Attack setup



Problems with CBC decryption?

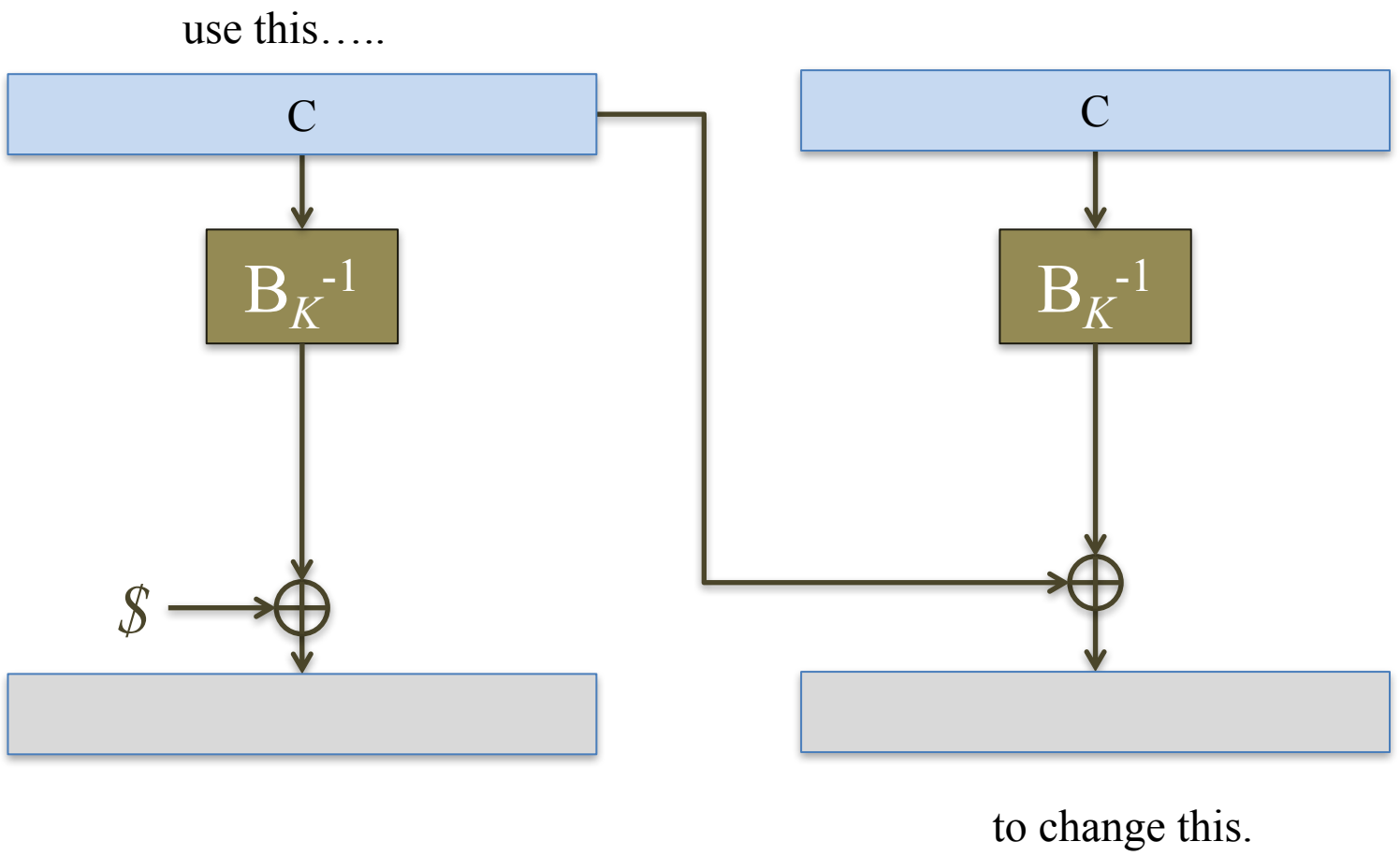


- Formally:
  - Doesn't provide integrity
  - Isn't nonce-respecting
  - …
- Specific concerns to exploit today: Malleability
  - Altering ciphertext block C_1 changes plaintext block P_2 *in a byte by byte manner*! (Destroys P_1 in the process, but no matter)

Padding oracle attack exceuction

Attack procedure

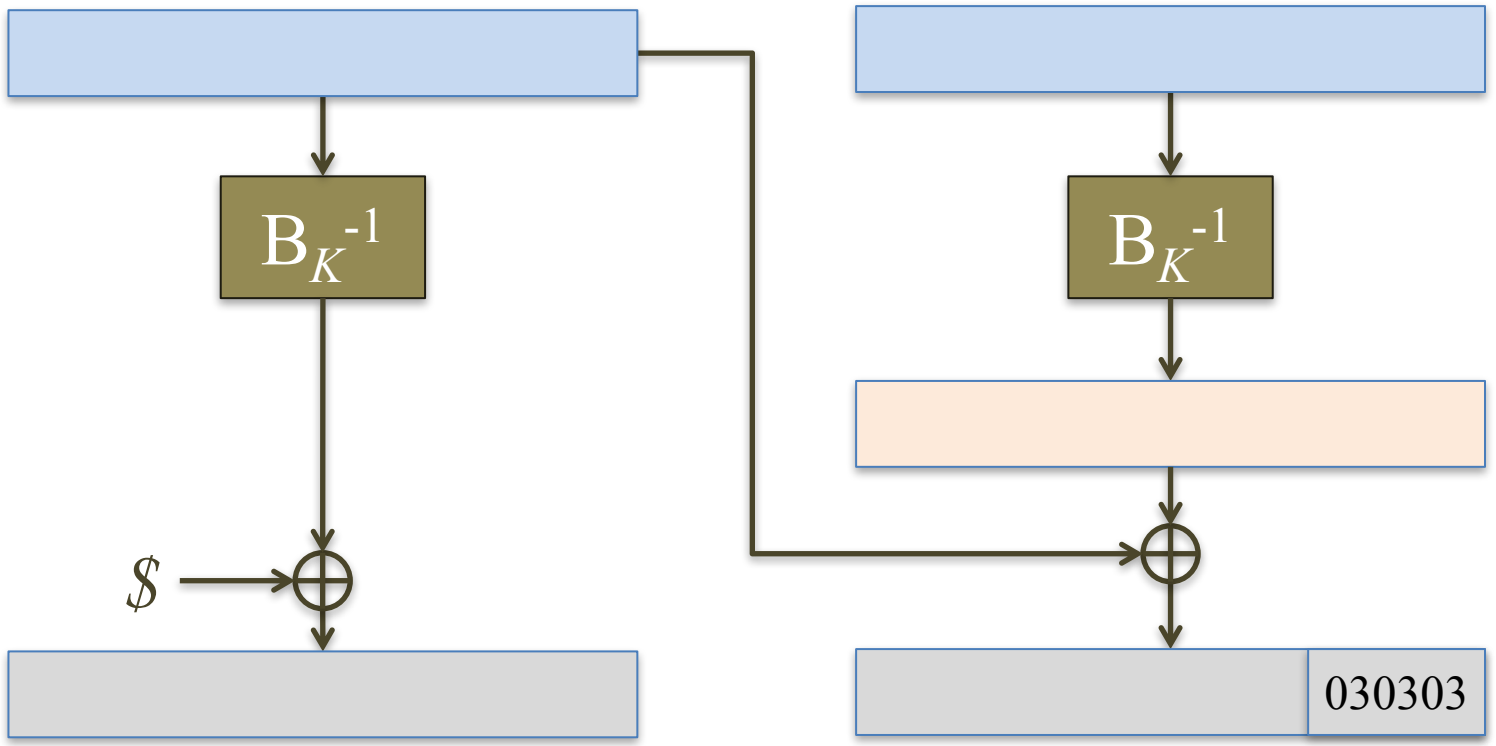
- Send 256 CTs to Bob,one for each value of c
- Probably all will fail.
  - 255 of the failures will be due to bad padding
  - 1 failures will have valid pad, bad MAC
- Save the value of c causing the 2nd error!



Padding oracle attack exceuction

Attack procedure

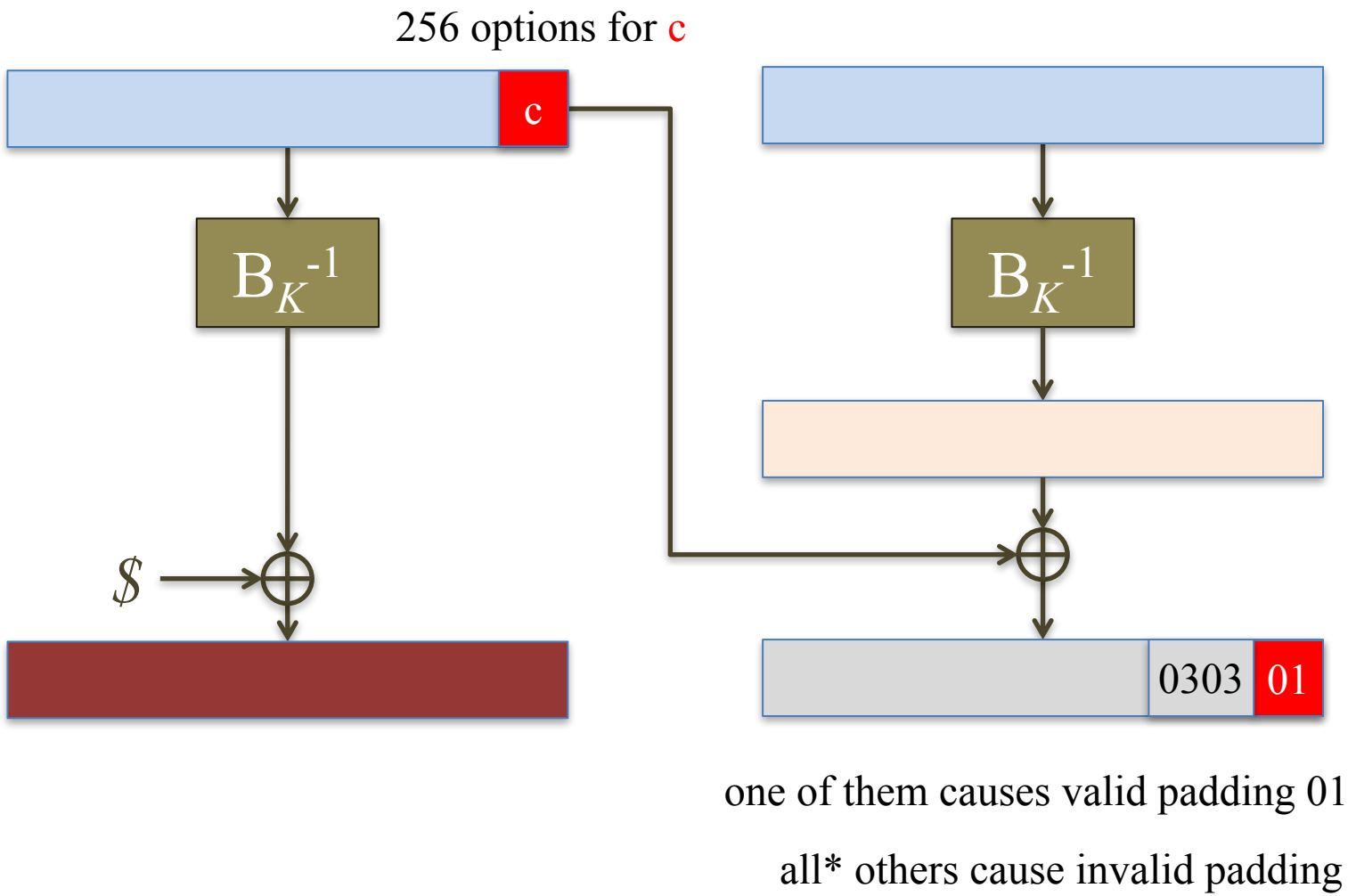
- Send 256 CTs to Bob,one for each value of c
- Probably all will fail.
  - 255 of the failures will be due to bad padding
  - 1 failures will have valid pad, bad MAC
- Save the value of c causing the 2nd error!



Padding oracle attack exceuction

Attack procedure

- Send 256 CTs to Bob,one for each value of c
- Probably all will fail.
  - 255 of the failures will be due to bad padding
  - 1 failures will have valid pad, bad MAC
- Save the value of c causing the 2nd error!



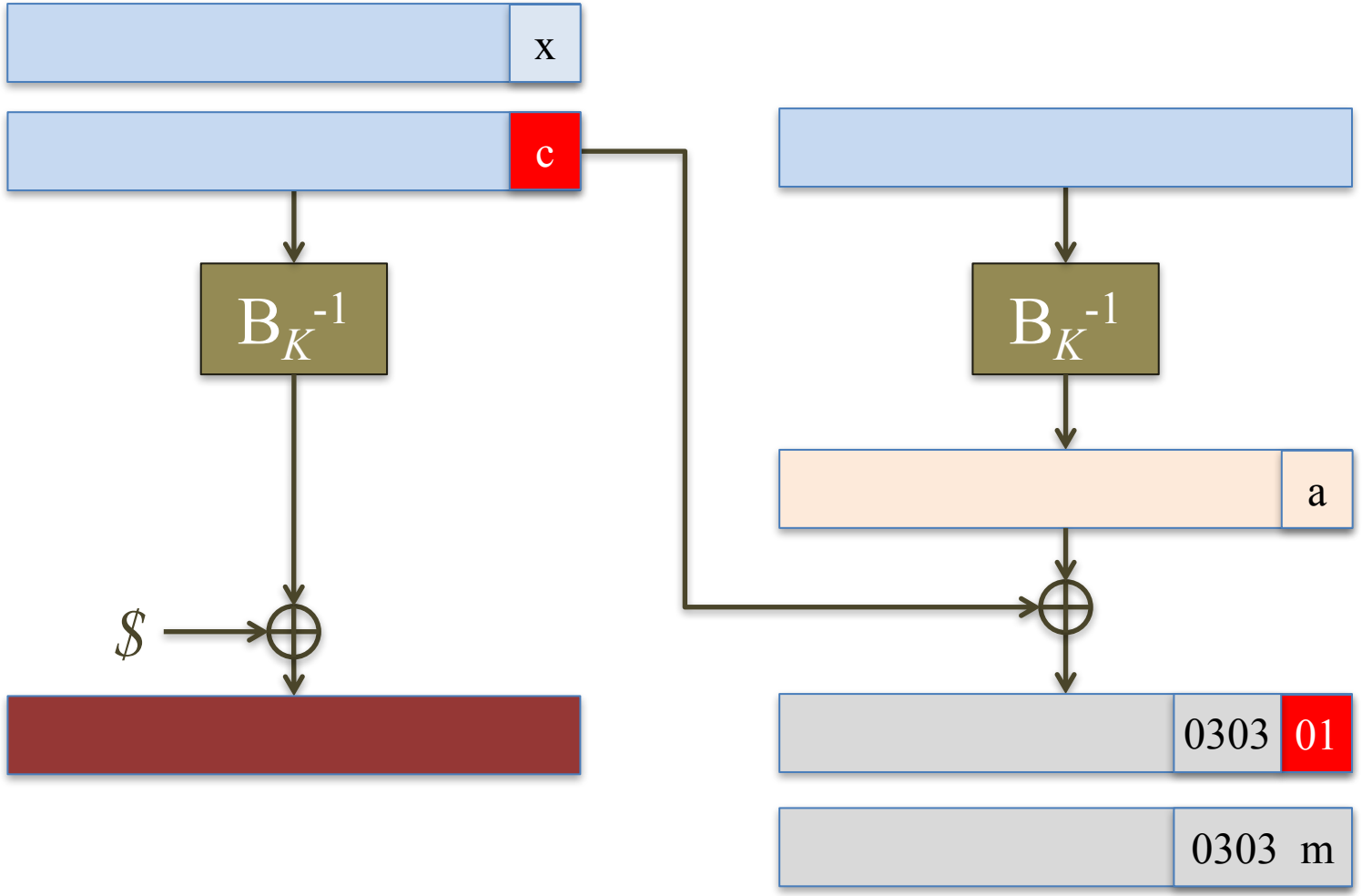
Recover plaintext byte

We can compute a two ways:

a = x ⊕ m

a = c ⊕ 0x01

Compute original message byte m = x ⊕ c ⊕ 0x01



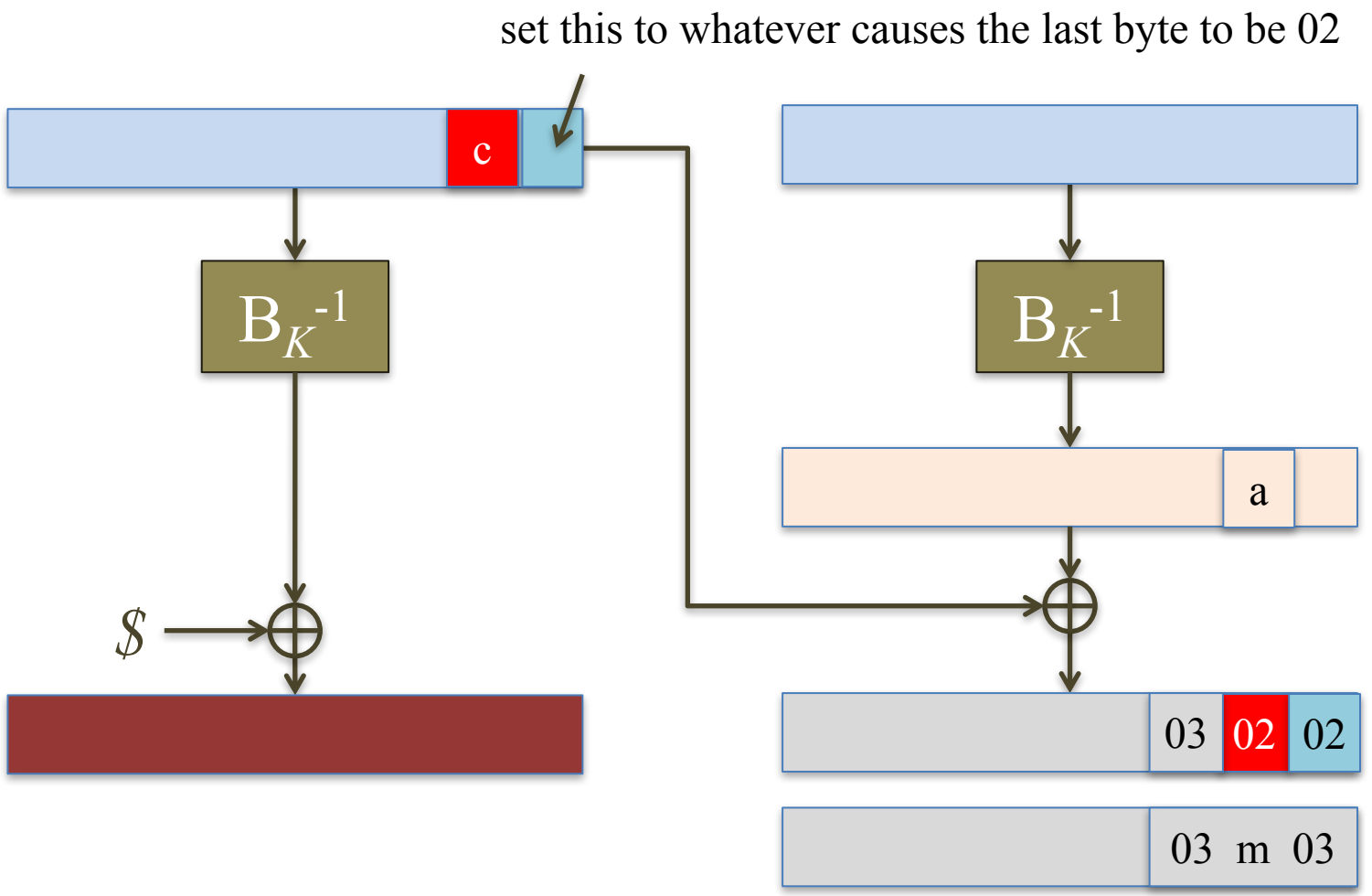
Recover NEXT plaintext byte!

We can compute a two ways:

a = x ⊕ m

a = c ⊕ 0x02

Compute original message byte m = x ⊕ c ⊕ 0x02



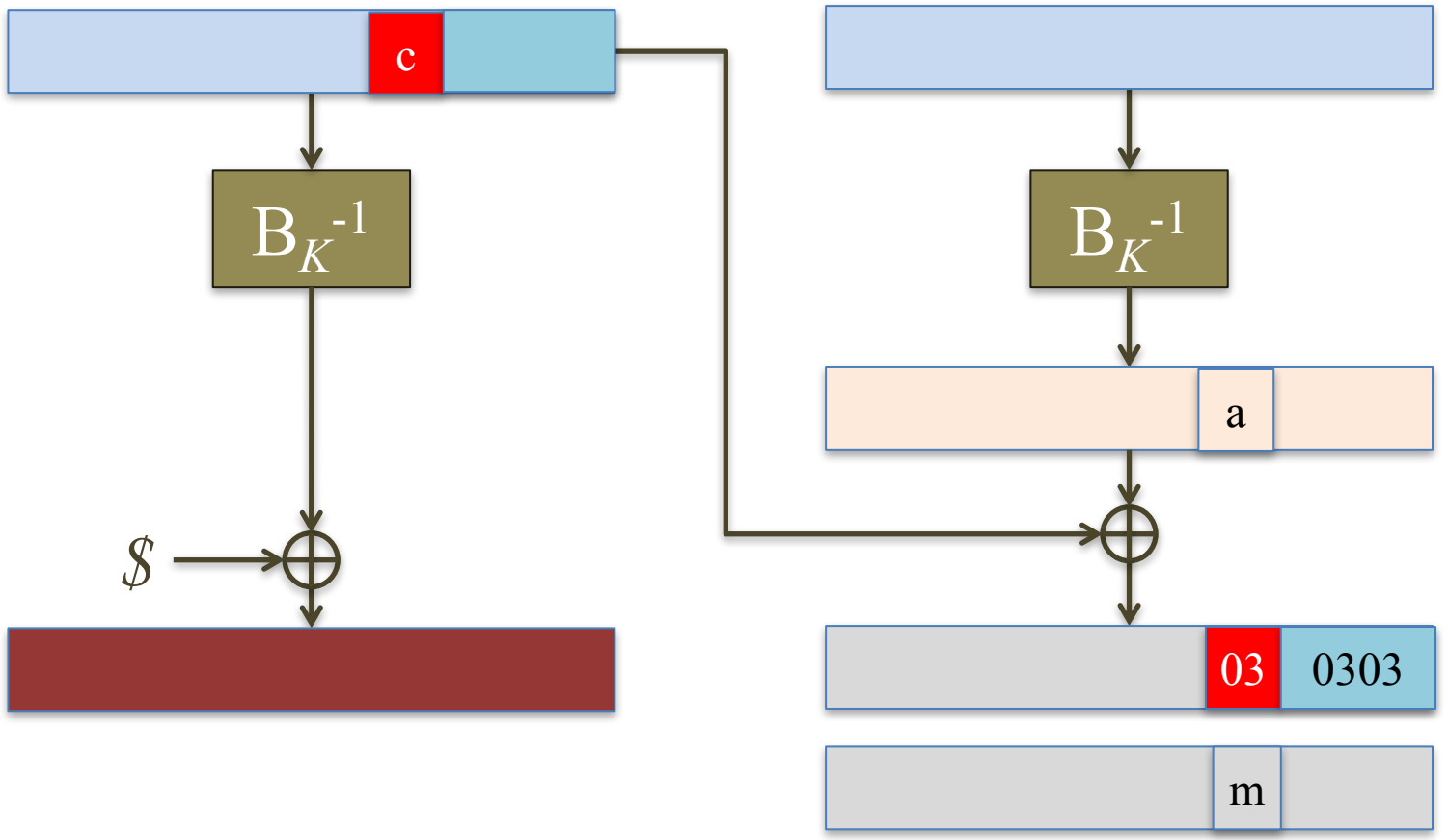
Rinse and repeat!

We can compute a two ways:

a = x ⊕ m

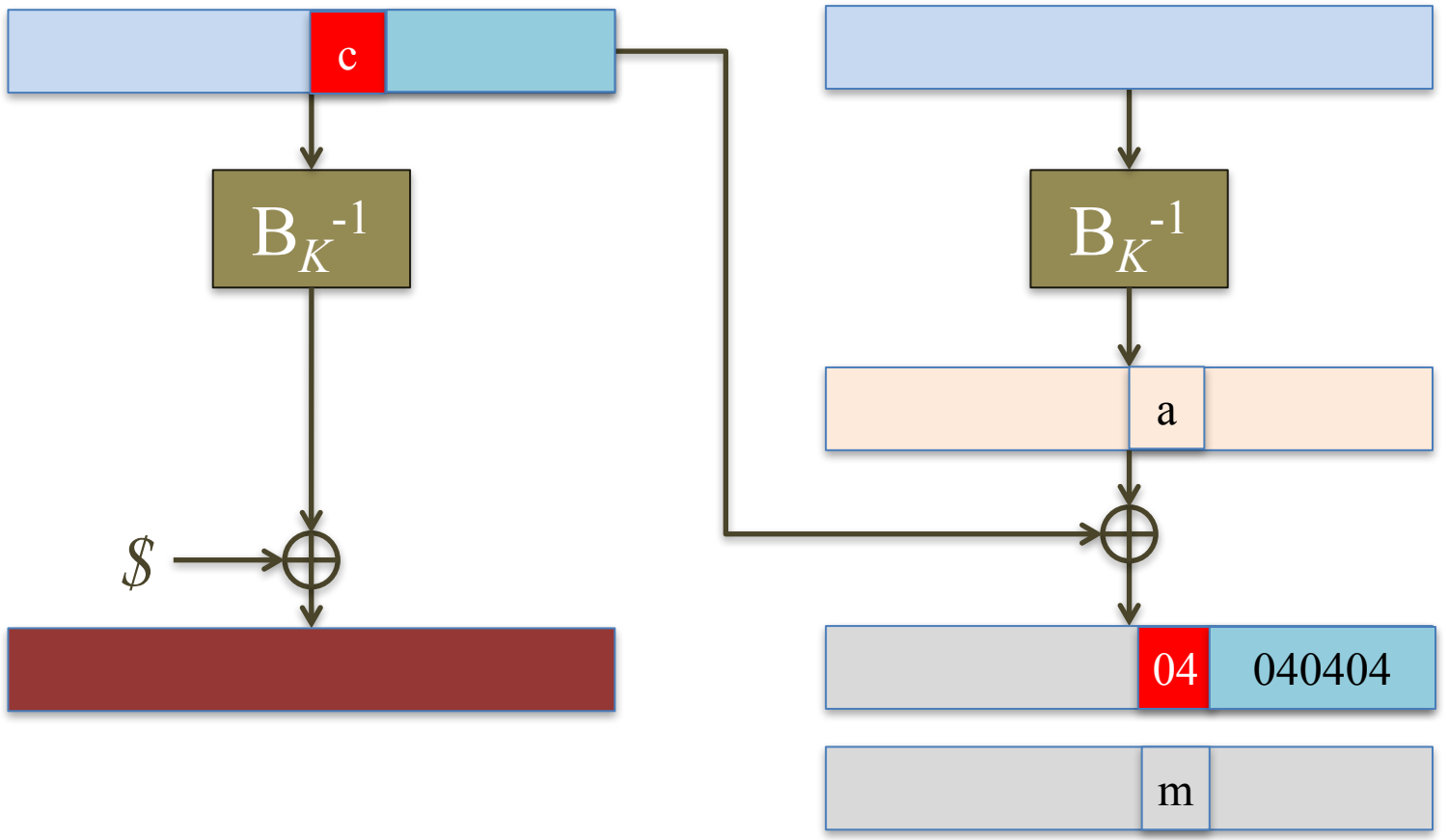
a = c ⊕ 0x03

Compute original message byte m = x ⊕ c ⊕ 0x03



Rinse and repeat!

We can recover the entire plaintext this way!



To recap

- Mallory knows C for unknown msg
- She uses mauling power to make all 256 options of final byte
- Exactly one will have the final byte 01 & thus look like a valid pad!
- The other mauled messages will have an invalid tag, such as 030302

private P	tag T	pad 030303

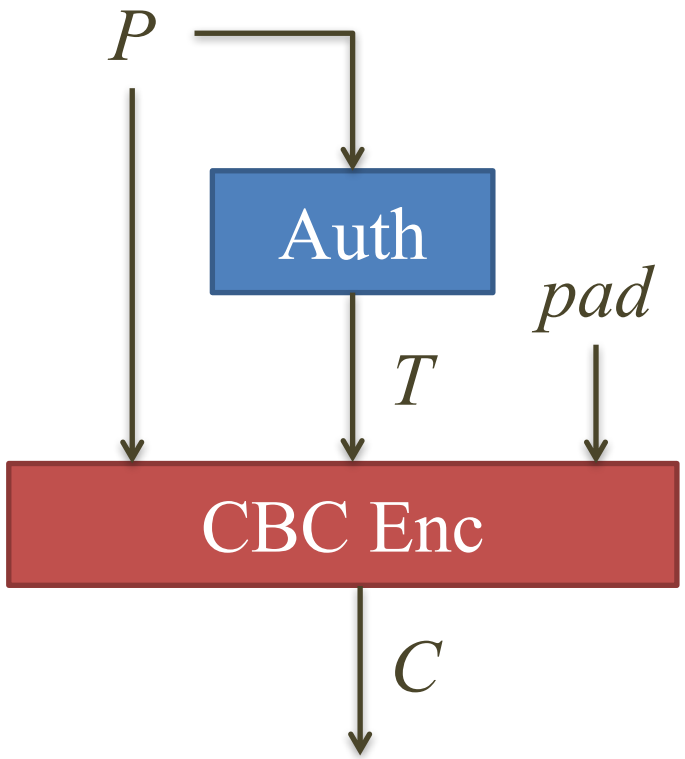
private P	tag T'	pad 0303xx

private P	tag (T' || 0303)	pad 01

- Different error messages → can distinguish between these two cases
- Use distinction to learn actual value of the final byte (here, 03)
- Repeat this process byte-by-byte to recover the entire plaintext!

To recap: What were the issues here?

- Ability to distinguish between different kinds of errors
- Malleability (CBC mode)
- Authentication was not first step
  - More on this later

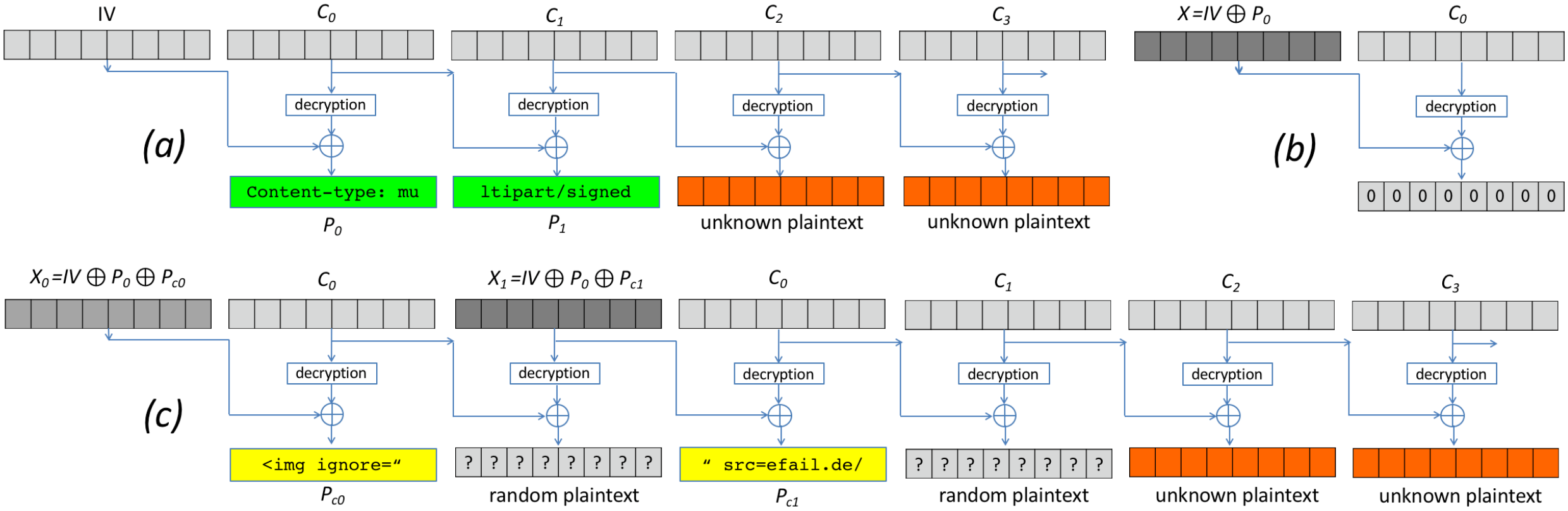


Another way to attack malleable modes: injection

Poddebniak et al, 2018, “EFAIL”

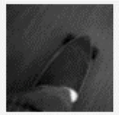
S/MIME and OpenPGP encrypted emails + CBC mode

Utilize DNS server under attacker control: learn content of message by injecting an image at “attacker.com/<encrypted-email-contents>



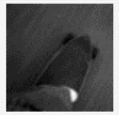
Padding Oracle Timeline

- 2002: Serge Vaudenay discovers CBC padding oracle attacks
- 2002-11: Extensions to specific systems like XML Encryption
- 2011: BEAST (Browser Exploit Against SSL/TLS) builds Java applet to perform the padding oracle in TLS 1.0
- 2013: Lucky 13 (TLS messages with 2 correct padding bytes processed faster than 1)
- 2014: POODLE (Padding Oracle On Downgraded Legacy Encryption) finds that the straightforward oracle works on SSL 3.0
- 2015: Extended Lucky 13 attack on Amazon's timing-independent TLS implementation
- 2017: TLS 1.3 breaks backward compatibility, permits Enc-then-Auth

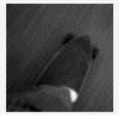


Jan Schaumann @jschauma
Attack timeline T given a theoretical vulnerability V:

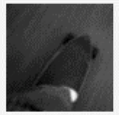
T0: academic research shows an attack is possible
Industry: Pfft, unrealistic.



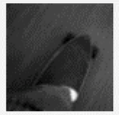
Jan Schaumann @jschauma
T1: nation-state attackers use the attack covertly
Industry: See, nobody's using this, nothing to worry about.



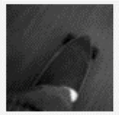
Jan Schaumann @jschauma
T2: academic research shows an attack is feasible with $$$$
Industry: We still have time.



Jan Schaumann @jschauma
T3: actually sophisticated attackers start using it
Industry: We should do something. Not today, but definitely Soon(tm).



Jan Schaumann @jschauma
T4: attacks become commodity, metasploit plugin appears
Industry: *gulp* *scrambles* *panic*



Jan Schaumann @jschauma
T5: with much pain, industry eliminates attack vector
Industry: yay, we're all good now

How can we fix this?

- Bob's solution: return the *same* error message in cases #1 and #2
- Remember the three cases
  1. Invalid padding
  2. Valid padding, wrong Auth
  3. Valid padding, right Auth
- Required effort
  - ⇒ Read the padding bytes
  - ⇒ Read padding bytes, compute the Auth
  - ⇒ Read padding bytes, compute the Auth
- Mallory's countermeasure: can still distinguish the two cases by observing the *time* that the MAC-then-Encrypt system takes to execute!
- Bob's new solution: ensure that crypto software's running time is *independent* of input; here, perform the HMAC test whether the padding is correct or not
- Mallory's new countermeasure: exploit timing variation within HMAC itself ☹

How to fix these vulnerabilities?

Can check to make sure you're operating on exactly what you expect

...but you had better make sure that this check is itself timing-independent

...and even then the fix might introduce a side-channel of its own

Basically, timing-independence is really hard!

OpenSSL Fact @OpenSSLFact Jul 24, 2013

/*The aim of right-shifting md_size is so that the compiler doesn't figure out that it can remove div_spoiler...which I hope is beyond it.*/

(So is software in general.)

OpenSSL Fact @OpenSSLFact Sep 3, 2012

/* [we should] obviate the ugly and illegal kludge in CRYPTO_mem_leaks_cb. Otherwise the code police will come and get us.*/

OpenSSL Fact @OpenSSLFact Jan 22, 2013

/* EEK! Experimental code starts */

OpenSSL Fact @OpenSSLFact Sep 5, 2012

/* BIG UGLY WARNING! This is so damn ugly I wanna puke ... ARGH! ARGH! ARGH! Let's get rid of this macro package. Please? */

What can you do?

Use good *crypto* coding conventions

This page lists coding rules with for each a description of the problem addressed (with a concrete example of failure), and then one or more solutions (with example code snippets).

Contents [hide]

Validate code for timing independence

agl / ctgrind

Code | Issues 0 | Pull requests 1 | Projects 0 | Pulse | Graphs

Checking that functions are constant time with Valgrind

3 commits | 1 branch | 0 releases

Branch: master | New pull request

Adam Langley C++ support and constify pointers ...

Makefile	Initial import
README	A couple of typos
ctgrind.c	C++ support and constify pointers
ctgrind.h	C++ support and constify pointers
test.c	Initial import
valgrind.patch	Initial import

README

```
Checking that functions are constant time with Valgrind.
```

Related sources of error

Compression oracles

- Basic idea: if you apply compression before encryption, then post-compression message length reveals some information about message!
- 2012: CRIME (Compression Ratio Info-leak Made Easy) recovers secret web cookies over HTTPS connections, hijacks sessions
- 2013: BREACH (Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext)

Format oracles

- Basic idea: if a higher-level protocol expects underlying message to obey some structural rules, can tinker with ciphertext until you find something that works
- 2011: "How to break XML encryption"
- 2015: "How to break XML encryption – automatically"

Summary: attacker oracles

- When we talk of 'oracles' in cryptanalysis, we mean that somehow the system is providing the attacker with the ability to compute f(P) for some function f
- There are many possible sources of these 'oracles'
  - Error messages
  - Message length, if a compression function is applied pre-encryption
  - Expected formatting of the underlying message (e.g., XML)
  - Time for a computation to finish
  - Performance speedup in running time due to the cache
  - Power consumed by the device