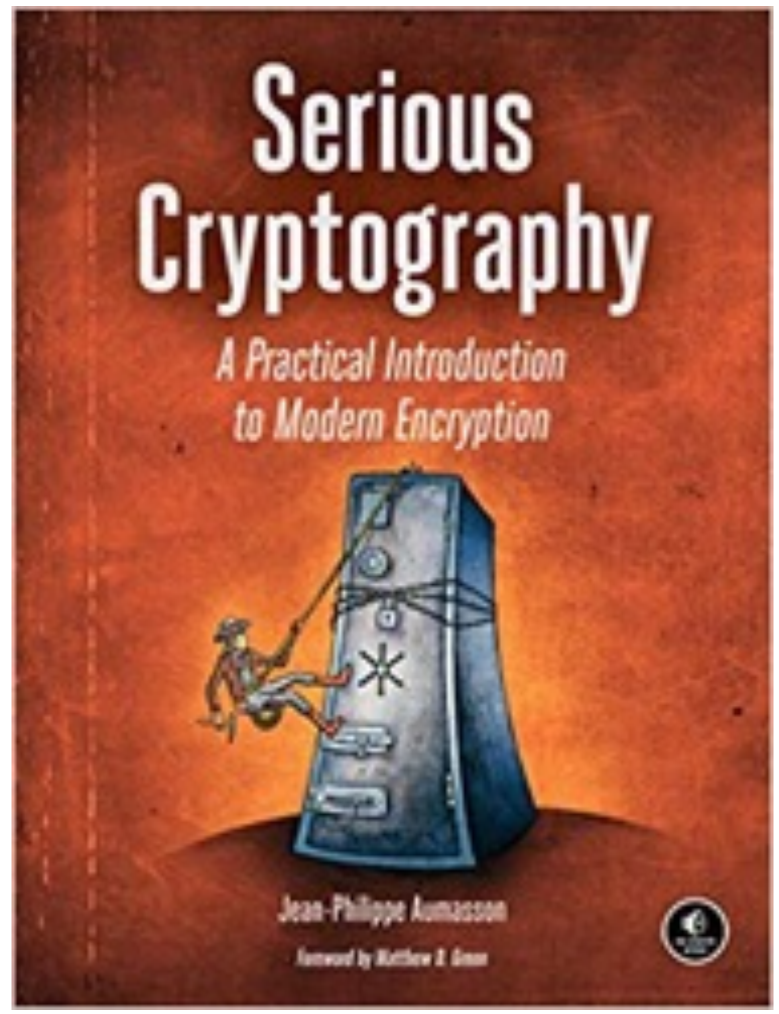
CNIT 141

Cryptography for Computer Networks



4. Block Ciphers

Updated 9-18-23

Topics

- What is a Block Cipher
- How to Construct Block Ciphers
- The Advanced Encryption Standard (AES)
- Implementing AES
- Modes of Operation
- How Things Can Go Wrong

History

- US: Federal standard: DES (1979 - 2005)
- KGB: GOST 28147-89 (1990 - present)
- in 2000, NIST selected AES, developed in Belgium
- They are all *block ciphers*

What is a Block Cipher

Block Cipher

E Encryption algorithm

K Key

P Plaintext block

C Ciphertext block

$C = \mathbf{E}(K, P)$

D Decryption algorithm

$P = \mathbf{D}(K, C)$

Security Goals

- Block cipher should be a *pseudorandom permutation (PRP)*
  - Attacker can't compute output without the key
- Attackers should be unable to find *patterns* in the inputs/output values
- The ciphertext should appear random

Block Size

- DES: 64 bit
- AES: 128 bit
- Chosen to fit into registers of CPUs for speed
- Block sizes below 64 are vulnerable to a *codebook attack*

Codebook Attack

- Suppose you can encrypt arbitrary plaintext, such as in a Wi-Fi network, but you don't know the key
  - Encrypt every possible plaintext, place in a *codebook*
  - Look up blocks of ciphertext in the *codebook*

Codebook Attack

- The codebook size depends on the block size
- Blocksize of 32 bits requires 2^32 entries
  - 4 billion, easily achieved
- Blocksize of 64 bits requires 2^64 entries
  - Impossible to achieve
- So blocksize should be 64 or larger

How to Construct Block Ciphers

Two Techniques

- Substitution-permutation (AES)
- Feistel (DES)

Rounds

- **R** is a *round* --in practice, a simple transformation
- A block cipher with three rounds:
  - $C = \mathbf{R_3}(\mathbf{R_2}(\mathbf{R_1}(P)))$
- **iR** is the inverse round function
  - $\mathbf{I} = \mathbf{iR_1}(\mathbf{iR_2}(\mathbf{iR_3}(C)))$

Round Key

- The round functions **R_1 R_2 R_3** use the same algorithm
- But a different *round key*
- Round keys are K_1, K_2, K_3, ... derived from the main key K using a *key schedule*

The Slide Attack and Round Keys

- Consider a block cipher with three rounds, and with all the round keys identical

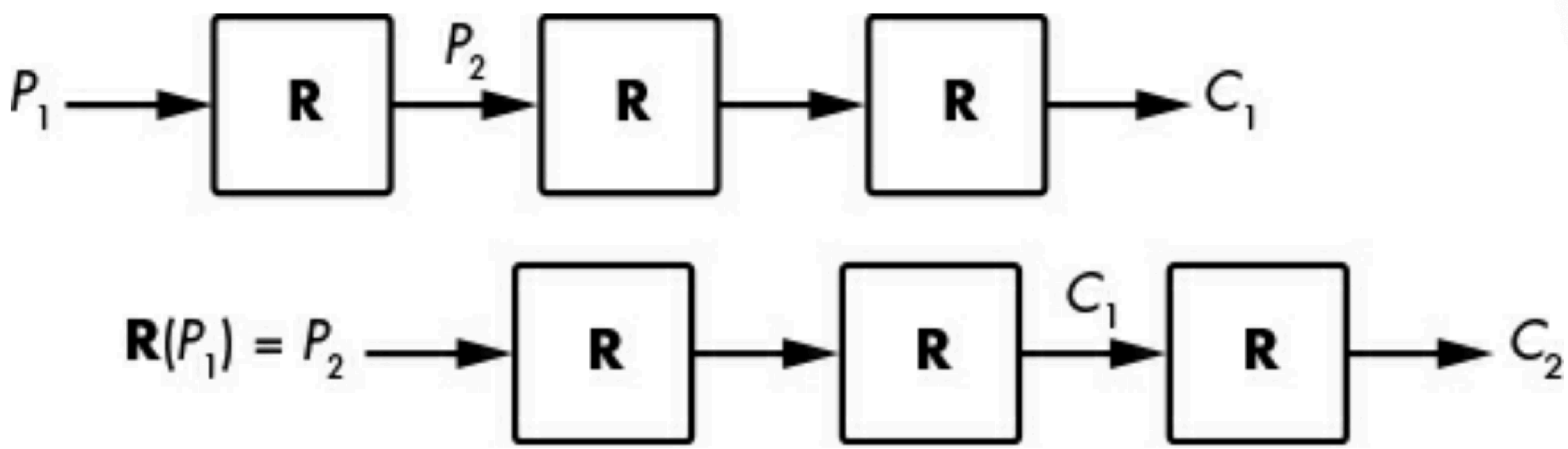


Figure 4-1: The principle of the slide attack, against block ciphers with identical rounds

The Slide Attack and Round Keys

- If an attacker can find plaintext blocks with $P_2 = \mathbf{R}(P_1)$
- That implies $C_2 = \mathbf{R}(C_1)$
- Which often helps to deduce the key

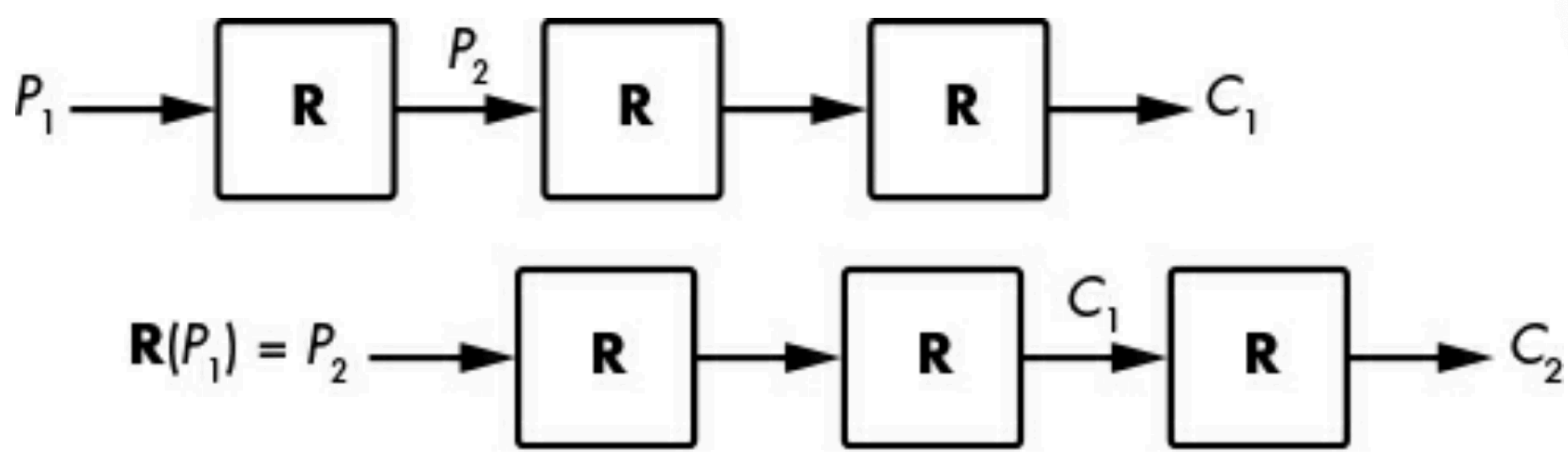


Figure 4-1: The principle of the slide attack, against block ciphers with identical rounds

The Slide Attack and Round Keys

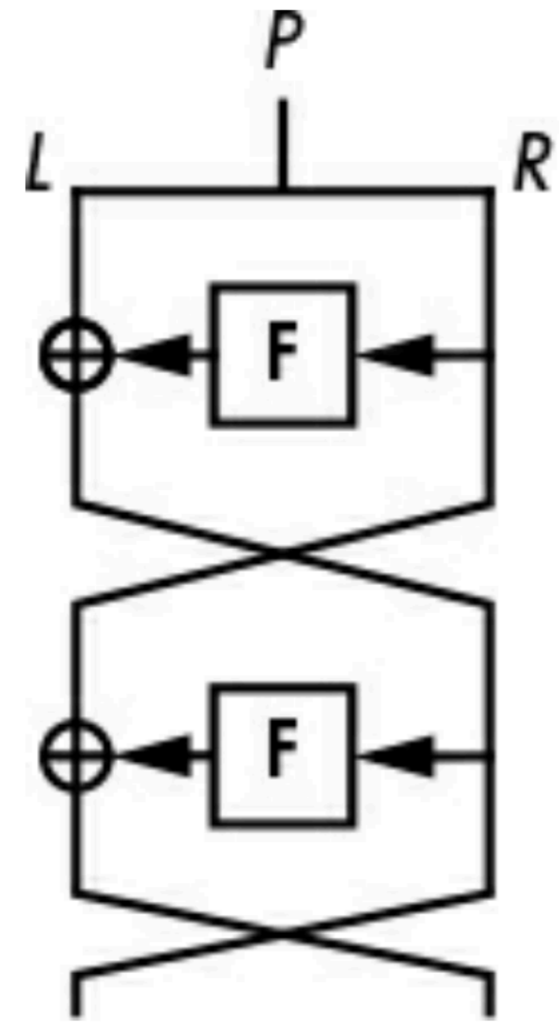
- The solution is to make all round keys different
- Note: the key schedule in AES is not one-way
  - Attacker can compute **K** from any $\mathbf{K_i}$
  - This exposes it to side-channel attacks, like measuring electromagnetic emanations

Substitution-Permutation Networks

- ***Confusion*** means that each ciphertext bit depends on several key bits
  - Provided by ***substitution*** using ***S-boxes***
- ***Diffusion*** means that changing a bit of plaintext changes many bits in the ciphertext
  - Provided by ***permutation***

Feistel Schemes

- Only half the plaintext is encrypted in each round
  - By the **F** substitution-permutation function
- Halves are swapped in each round
- DES uses 16 Feistel rounds



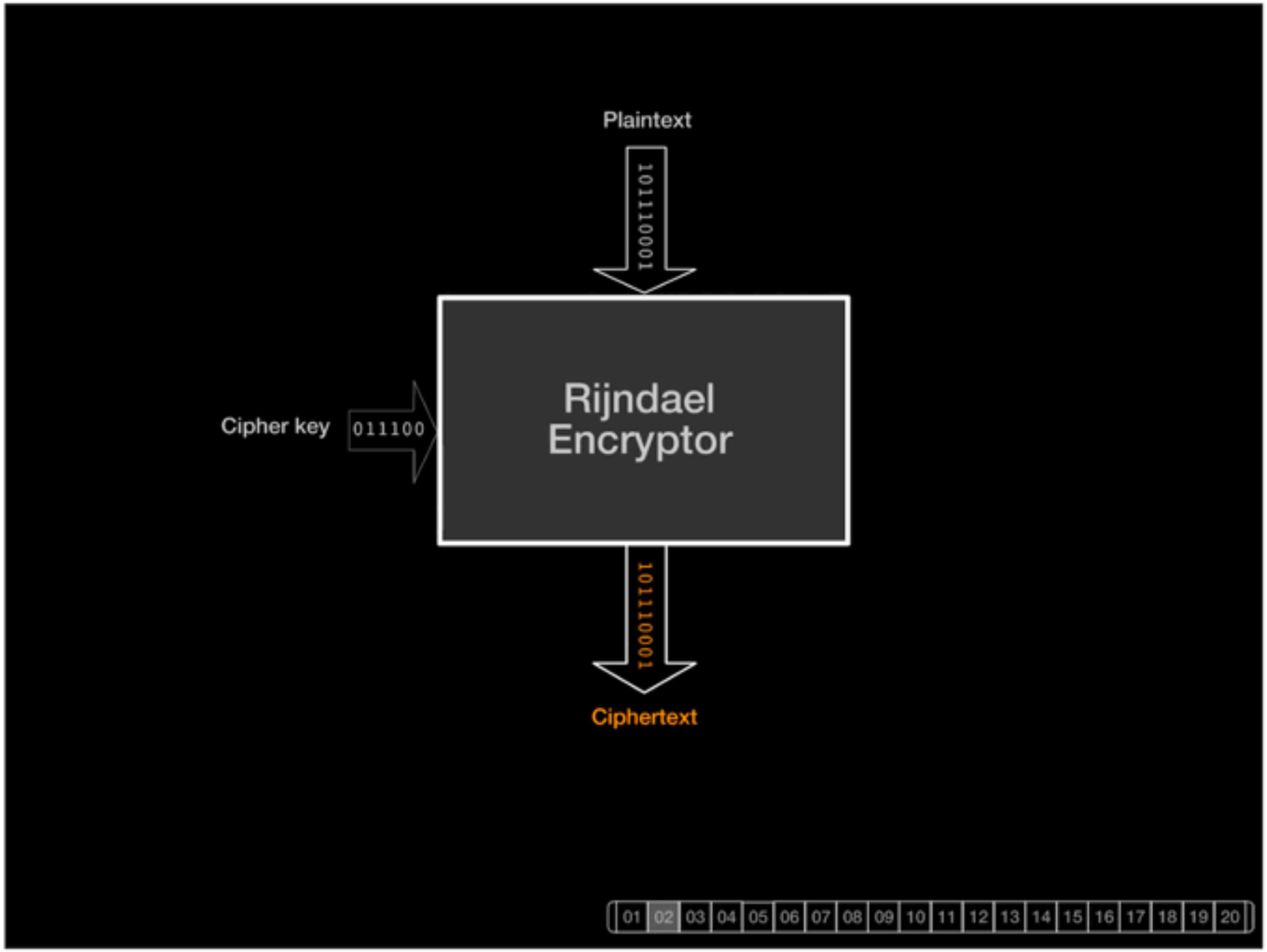
Kahoot!

4a

The Advanced Encryption Standard (AES)

DES

- DES had a 56-bit key
  - Cracked by brute force in 1997
- 3DES was a stronger version
  - Still considered strong, but slower than AES
- AES approved as the NIST standard in 2000



AES Rijndael Cipher explained as a Flash animation

- Link Ch 4a

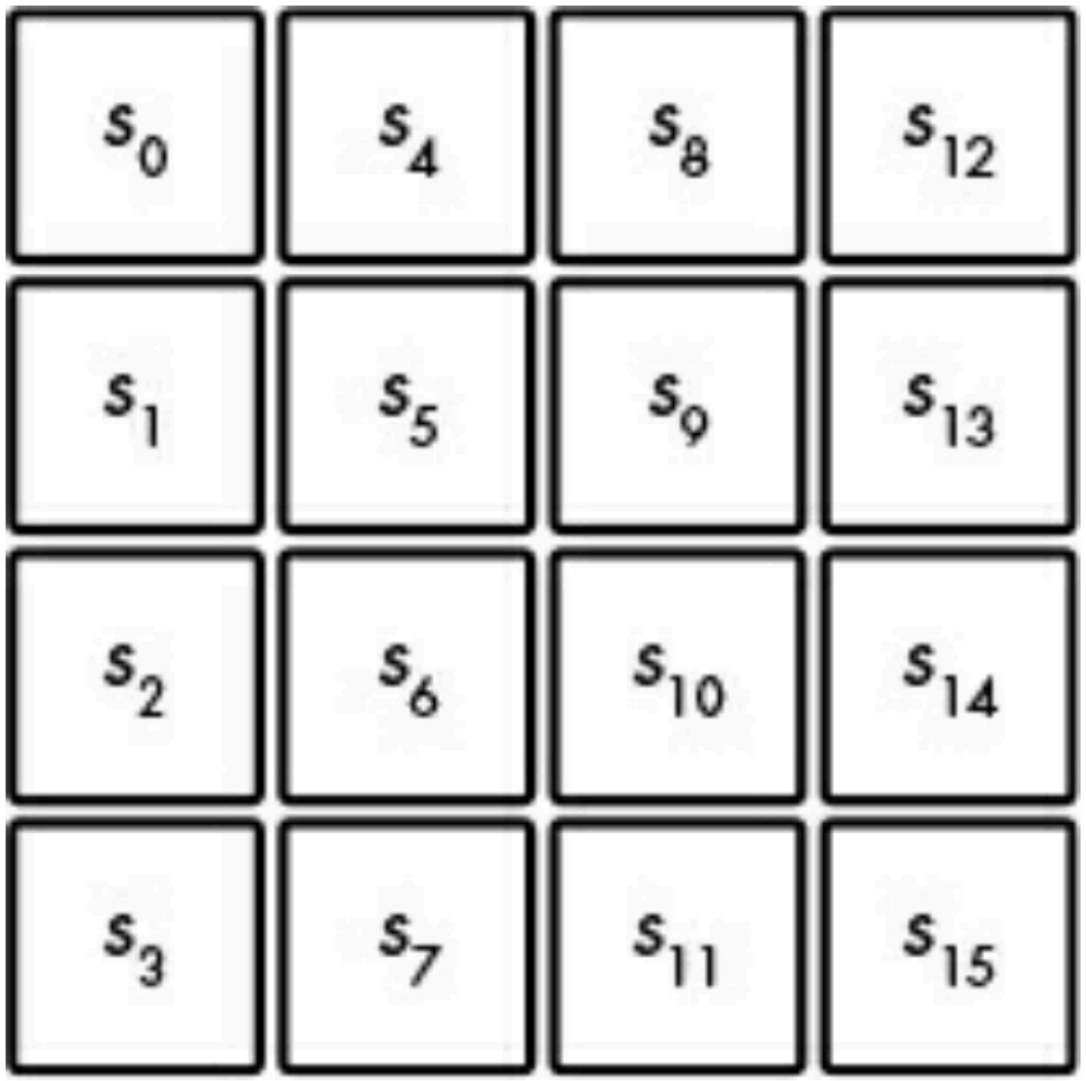
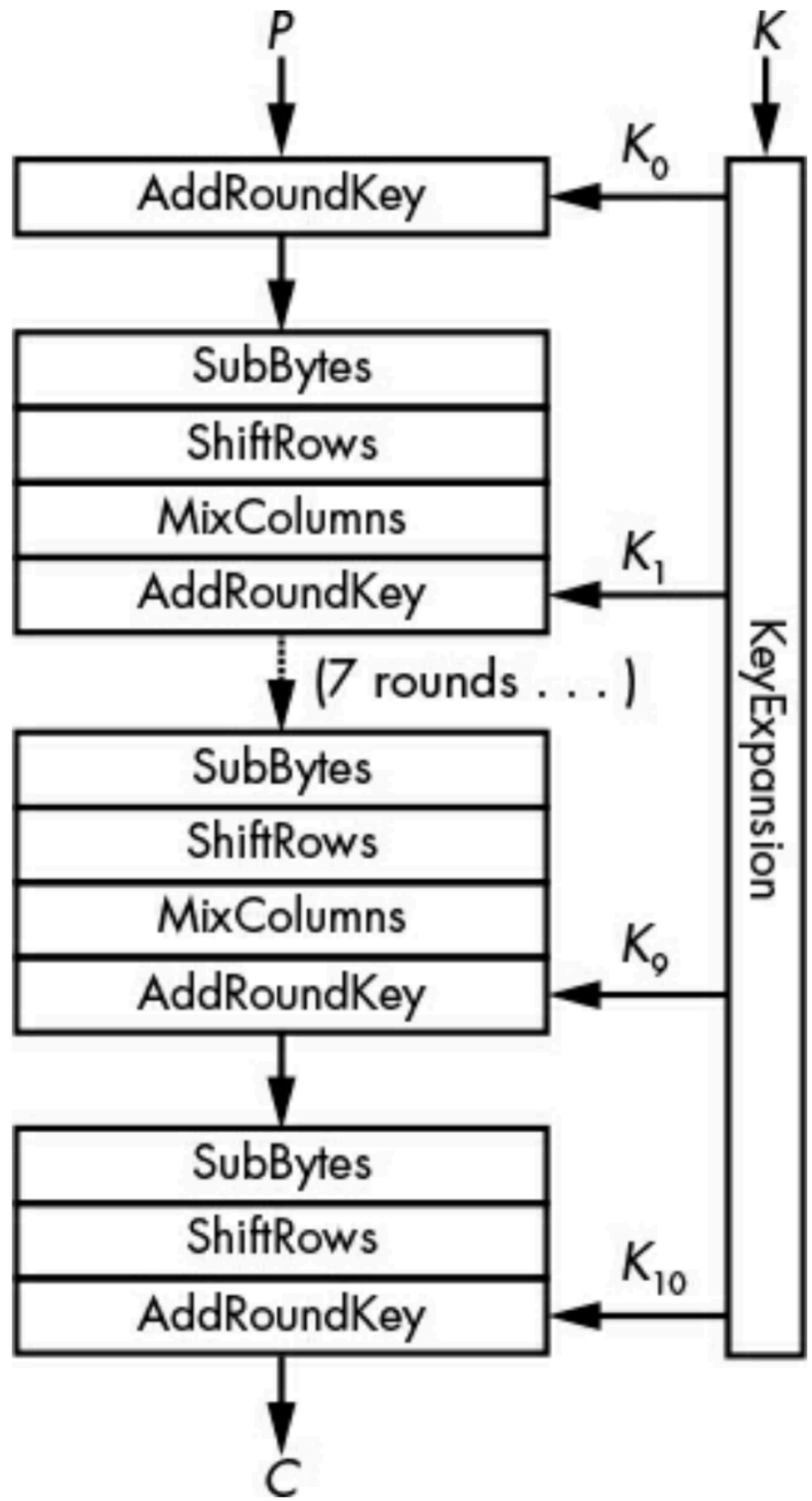


Figure 4-3: The internal state of AES viewed as a 4 × 4 array of 16 bytes


P
K
K0
AddRoundKey
SubBytes
ShiftRows
MixColumns
K1
AddRoundKey
(7 rounds . . .)
SubBytes
ShiftRows
MixColumns
K9
AddRoundKey
SubBytes
ShiftRows
K10
AddRoundKey
KeyExpansion
C

- **Without KeyExpansion, all rounds would use the same key, *K*, and AES would be vulnerable to slide attacks.**
- **Without AddRoundKey, encryption wouldn't depend on the key; hence, anyone could decrypt any ciphertext without the key.**
- **SubBytes brings nonlinear operations, which add cryptographic strength. Without it, AES would just be a large system of linear equations that is solvable using high-school algebra.**
- **Without ShiftRows, changes in a given column would never affect the other columns, meaning you could break AES by building four 2^{32}-element codebooks for each column. (Remember that in a secure block cipher, flipping a bit in the input should affect all the output bits.)**
- **Without MixColumns, changes in a byte would not affect any other bytes of the state. A chosen-plaintext attacker could then decrypt any ciphertext after storing 16 lookup tables of 256 bytes each that hold the encrypted values of each possible value of a byte.**

AES in Python 3

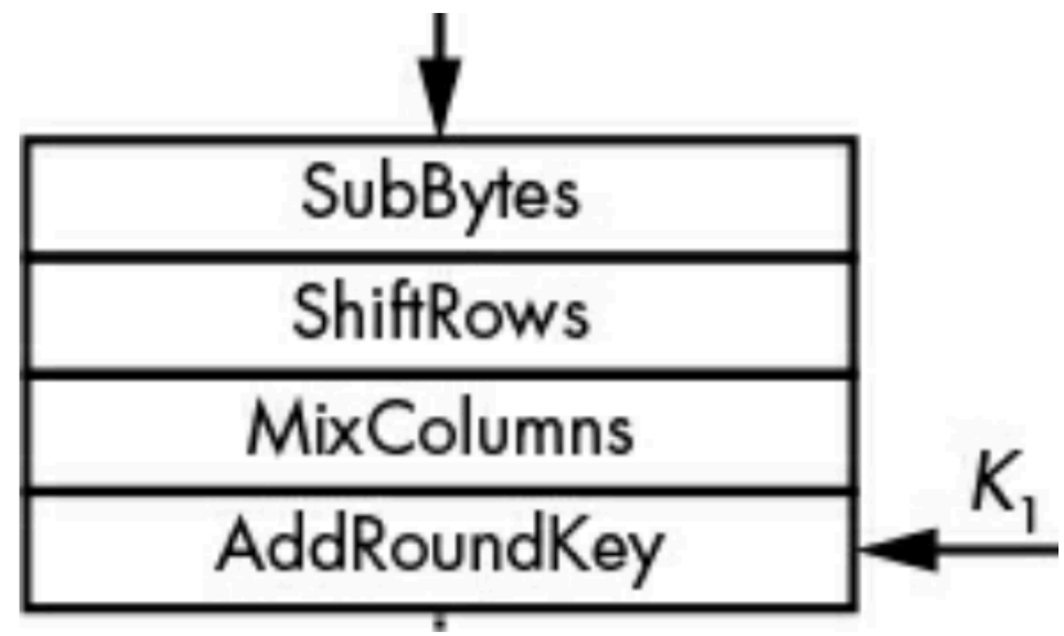
```
from Crypto.Cipher import
AES
key = b"0000111122223333"
cipher = AES.new(key,
AES.MODE_ECB)
a = b"Hello from AES!!"
ciphertext =
cipher.encrypt(a)
print(ciphertext.hex())

cipher = AES.new(key,
AES.MODE_ECB)
d =
cipher.decrypt(ciphertext)
print(d)
```

```
>>> from Crypto.Cipher import AES
>>> key = b"0000111122223333"
>>> cipher = AES.new(key, AES.MODE_ECB)
>>> a = b"Hello from AES!!"
>>> ciphertext = cipher.encrypt(a)
>>> print(ciphertext.hex())
59305bbea4c731a031f1423bb7cf643e
>>>
>>> cipher = AES.new(key, AES.MODE_ECB)
>>> d = cipher.decrypt(ciphertext)
>>> print(d)
b'Hello from AES!!'
```

Implementing AES

Improving Efficiency

- Implementing each step as a separate function works, but it's slow
- Combining them with "table-based implementations" and "native instructions" is faster
  - Using XORs and table lookups



OpenSSL Code is Table-Based

```
    /* round 1: */
    t0 = Te0[s0 >> 24] ^ Te1[(s1 >> 16) & 0xff] ^ Te2[(s2 >> 8) & 0xff] ^ Te3[s3 & 0xff] ^ rk[ 4];
    t1 = Te0[s1 >> 24] ^ Te1[(s2 >> 16) & 0xff] ^ Te2[(s3 >> 8) & 0xff] ^ Te3[s0 & 0xff] ^ rk[ 5];
    t2 = Te0[s2 >> 24] ^ Te1[(s3 >> 16) & 0xff] ^ Te2[(s0 >> 8) & 0xff] ^ Te3[s1 & 0xff] ^ rk[ 6];
    t3 = Te0[s3 >> 24] ^ Te1[(s0 >> 16) & 0xff] ^ Te2[(s1 >> 8) & 0xff] ^ Te3[s2 & 0xff] ^ rk[ 7];
    /* round 2: */
    s0 = Te0[t0 >> 24] ^ Te1[(t1 >> 16) & 0xff] ^ Te2[(t2 >> 8) & 0xff] ^ Te3[t3 & 0xff] ^ rk[ 8];
    s1 = Te0[t1 >> 24] ^ Te1[(t2 >> 16) & 0xff] ^ Te2[(t3 >> 8) & 0xff] ^ Te3[t0 & 0xff] ^ rk[ 9];
    s2 = Te0[t2 >> 24] ^ Te1[(t3 >> 16) & 0xff] ^ Te2[(t0 >> 8) & 0xff] ^ Te3[t1 & 0xff] ^ rk[10];
    s3 = Te0[t3 >> 24] ^ Te1[(t0 >> 16) & 0xff] ^ Te2[(t1 >> 8) & 0xff] ^ Te3[t2 & 0xff] ^ rk[11];
--snip--
```

Listing 4-2: The table-based C implementation of AES in OpenSSL

Timing Attacks

- The time required for encryption depends on the key
- Measuring timing leaks information about the key
- This is a problem with any efficient coding
- You could use slow code that wastes time
- A better solution relies on **hardware**

Native Instructions

- **AES-NI**
  - Processor provides dedicated assembly instructions that perform AES
  - Plaintext in register **xmm0**
  - Round keys in **xmm5** to **xmm15**
- **NI** makes AES ten times faster

```
PXOR       %xmm5,  %xmm0
AESENC     %xmm6,  %xmm0
AESENC     %xmm7,  %xmm0
AESENC     %xmm8,  %xmm0
AESENC     %xmm9,  %xmm0
AESENC     %xmm10, %xmm0
AESENC     %xmm11, %xmm0
AESENC     %xmm12, %xmm0
AESENC     %xmm13, %xmm0
AESENC     %xmm14, %xmm0
AESENCLAST %xmm15, %xmm0
```

Is AES Secure?

- AES implements many good design principles
- Proven to resist many classes of cryptoanalytic attacks
- But no one can foresee all possible future attacks
- So far, no significant weakness in AES-128 has been found

Modes of Operation

Electronic Code Book (ECB)

- Each plaintext block is encrypted the same way
- Identical plaintext blocks produce identical ciphertext blocks

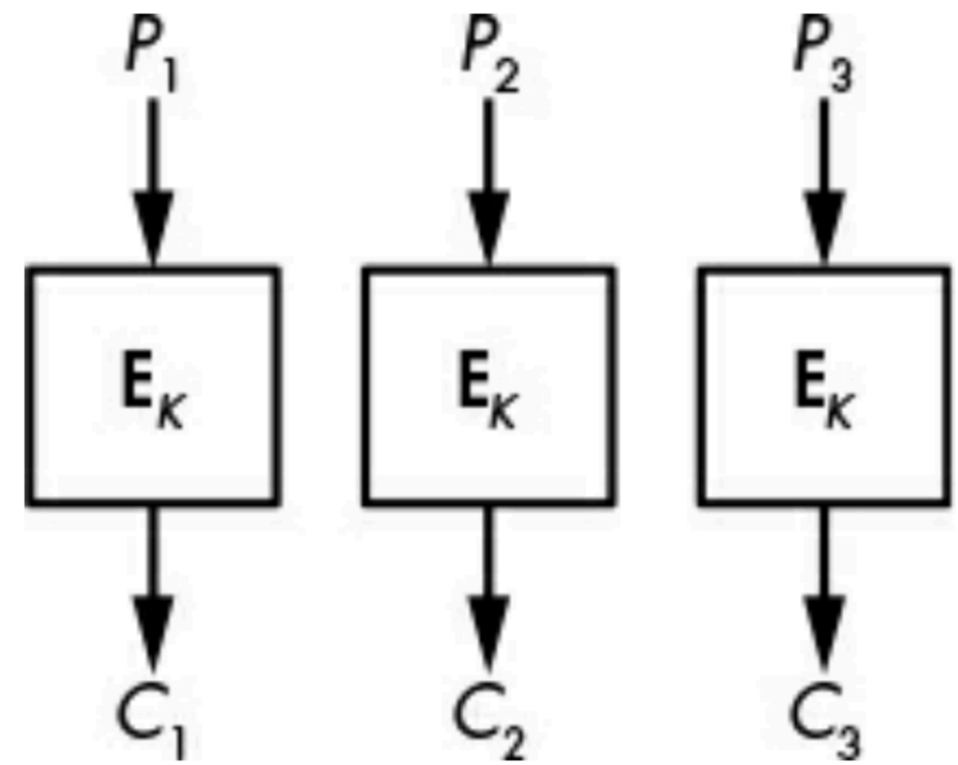


Figure 4-6: The ECB mode

AES-ECB

- If plaintext repeats, so does ciphertext

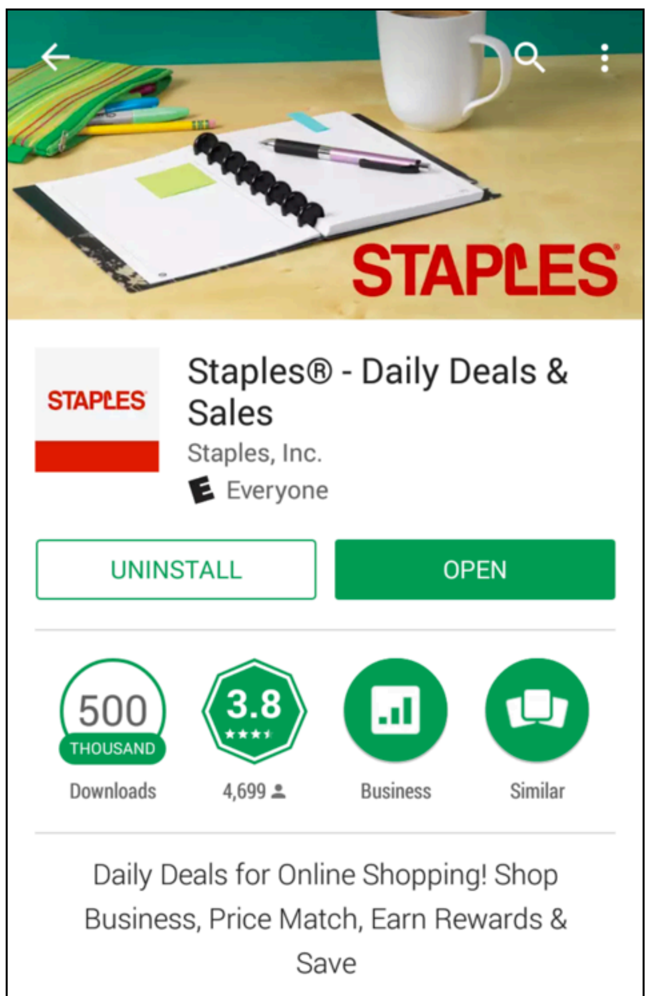
```
plaintext = b"DEAD MEN TELL NODEAD MEN TELL NO"
ciphertext = cipher.encrypt(plaintext)
ciphertext.hex()
```

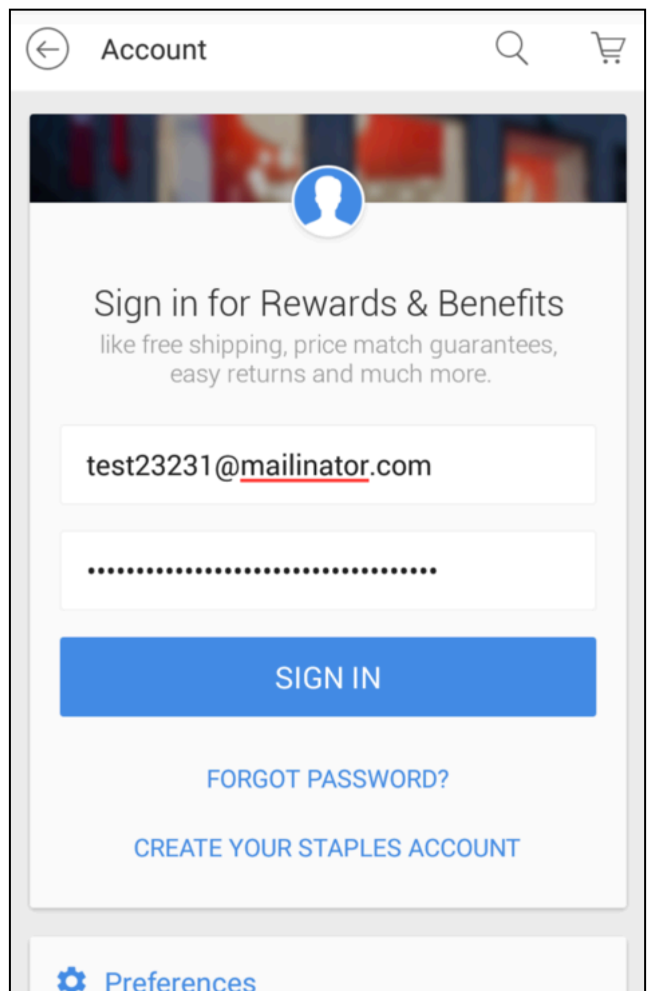
```
>>> plaintext = b"DEAD MEN TELL NODEAD MEN TELL NO"
>>> ciphertext = cipher.encrypt(plaintext)
>>> ciphertext.hex()
'66cc2a4741a704c7849450129bd29ce566cc2a4741a704c7849450129bd29ce5'
```

Staples Android App

The encryption uses AES in ECB (Electronic Code Book) mode, as shown by making an account with this password:

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa123A
```





- Link Ch 4b

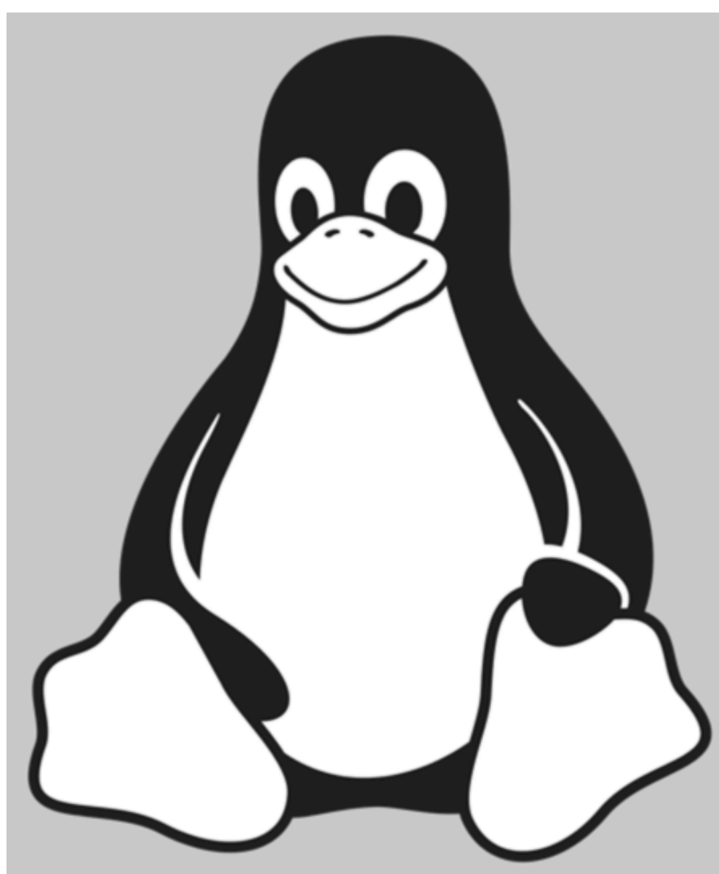
Encrypted Password Repeats

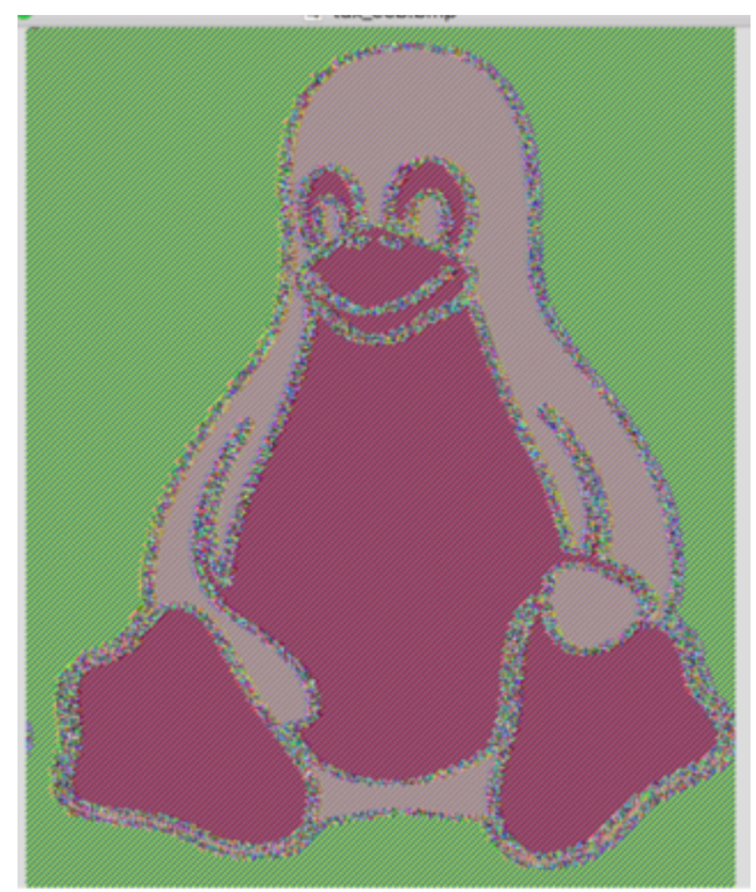
```
./shared_prefs/com.staples.mobile.cfa.xml:    <string name="encryptedPassword">
Ex+zjrCIlgw/kkZ0dIRfPhMfs46wiJYMP5JGdHSEXz4VWo8KfvDtbU+NuhSpui58</string>
```

```
>>> a="Ex+zjrCIlgw/kkZ0dIRfPhMfs46wiJYMP5JGdHSE
Xz4VWo8KfvDtbU+NuhSpui58"
>>> a1 = base64.b64decode(a)
>>> for c in a1:
...   d = hex(ord(c))
...   print d[2:],
...
13 1f b3 8e b0 88 96 c 3f 92 46 74 74 84 5f 3e
13 1f b3 8e b0 88 96 c 3f 92 46 74 74 84 5f 3e
15 5a 8f a 7e f0 ed 6d 4f 8d ba 14 a9 ba 2e 7c
>>>
```

ECB Mode

- Encrypted image retains large blocks of solid color





Cipher Block Chaining (CBC)

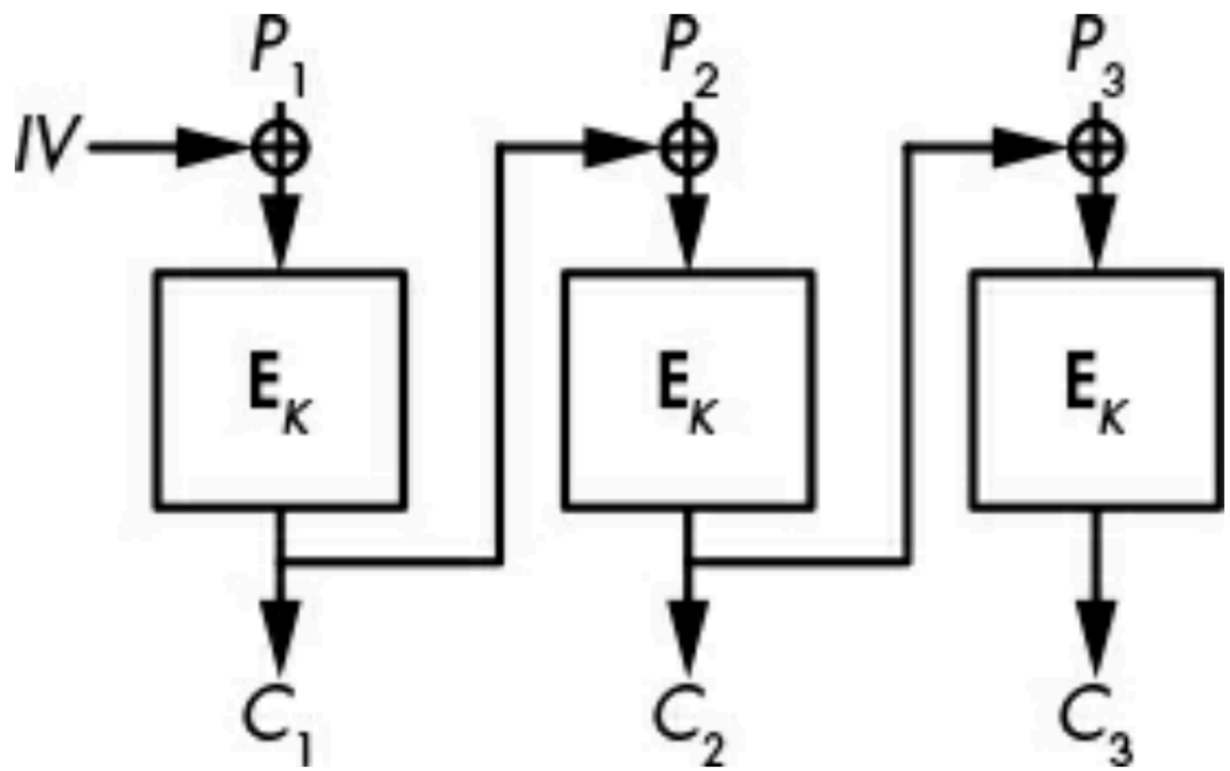
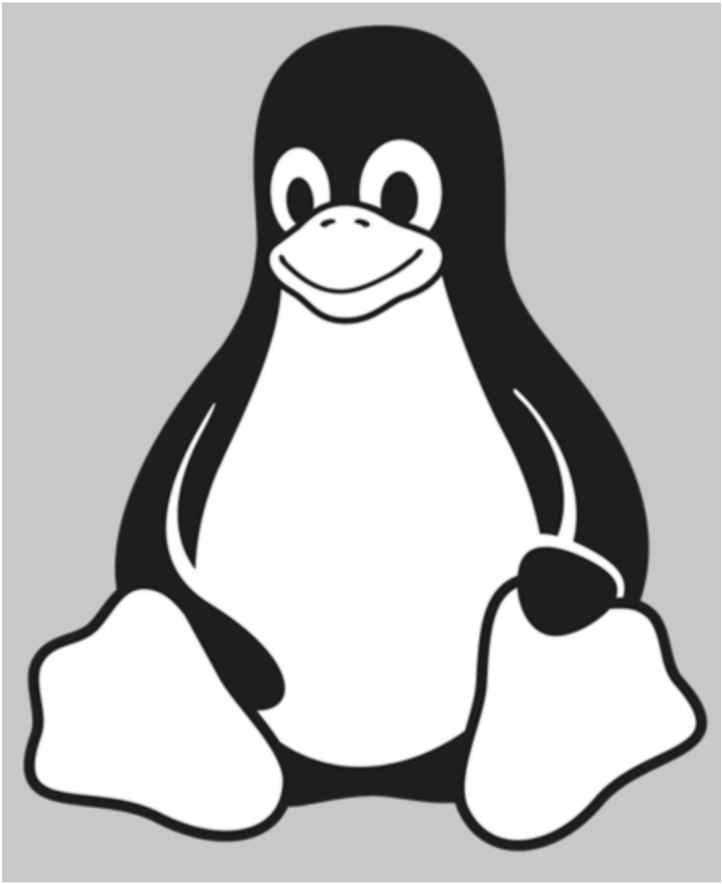


Figure 4-8: The CBC mode

- Uses a key and an **initialization vector (IV)**
- Output of one block is the **IV** for the next block
- **IV** is not secret; sent in the clear

CBC Mode

- Encrypted image shows no patterns



Choosing IV

- If the same IV is used every time
  - The first block is always encrypted the same way
  - Messages with the same first plaintext block will have identical first ciphertext blocks

Parallelism

- ECB can be computed in parallel
  - Each block is independent
- CBC requires serial processing
  - Output of each block used to encrypt the next block

Message Length

- AES requires 16-byte blocks of plaintext
- Messages must be **padded** to make them long enough

```
>>> plaintext = "HELLO"
>>> ciphertext = cipher.encrypt(plaintext)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python2.7/site-packages/Crypto/Cipher/blocka
lgo.py", line 244, in encrypt
    return self._cipher.encrypt(plaintext)
ValueError: Input strings must be a multiple of 16 in length
>>>
```

PKCS#7 Padding

- If one byte of padding is needed, use **01**
- If two bytes of padding are needed, use **0202**
- If three bytes of padding are needed, use **030303**
- ...
- If fifteen bytes of padding are needed, use **0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f**
- If no bytes of padding are needed, add an entire block of sixteen chr(16) characters, or **10101010101010101010101010101010**

- The last byte of the plaintext is always between '\x00' and '\10'
- Discard that many bytes to get original plaintext

Padding Oracle Attack

- Almost everything uses PKCS#7 padding
- But if the system displays a "Padding Error" message the whole system shatters like glass
- That message is sufficient side-channel information to allow an attacker to forge messages without the key

Ciphertext Stealing

- An alternative to padding
  - Prevents padding oracle attacks
- Pad with zeroes
- Swap last two blocks of ciphertext
- Discard extra bytes at the end
  - Images on next slides from Wikipedia

Ciphertext Stealing

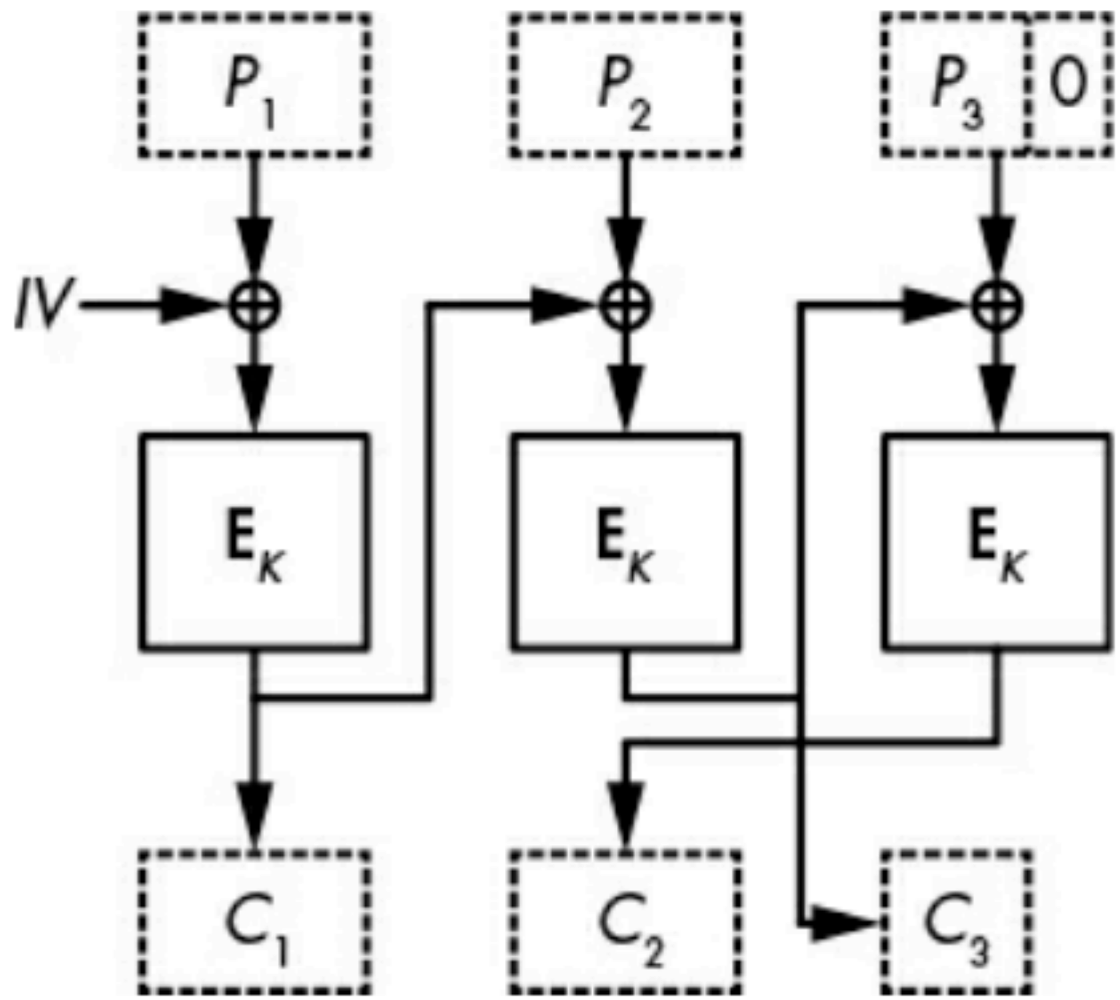


Figure 4-9: Ciphertext stealing for CBC-mode encryption

Security of Ciphertext Stealing

- No major problems
- Inelegant and difficult to get right
- NIST SP 800-38A specifies three different ways to implement it
- Rarely used

Counter (CTR) Mode

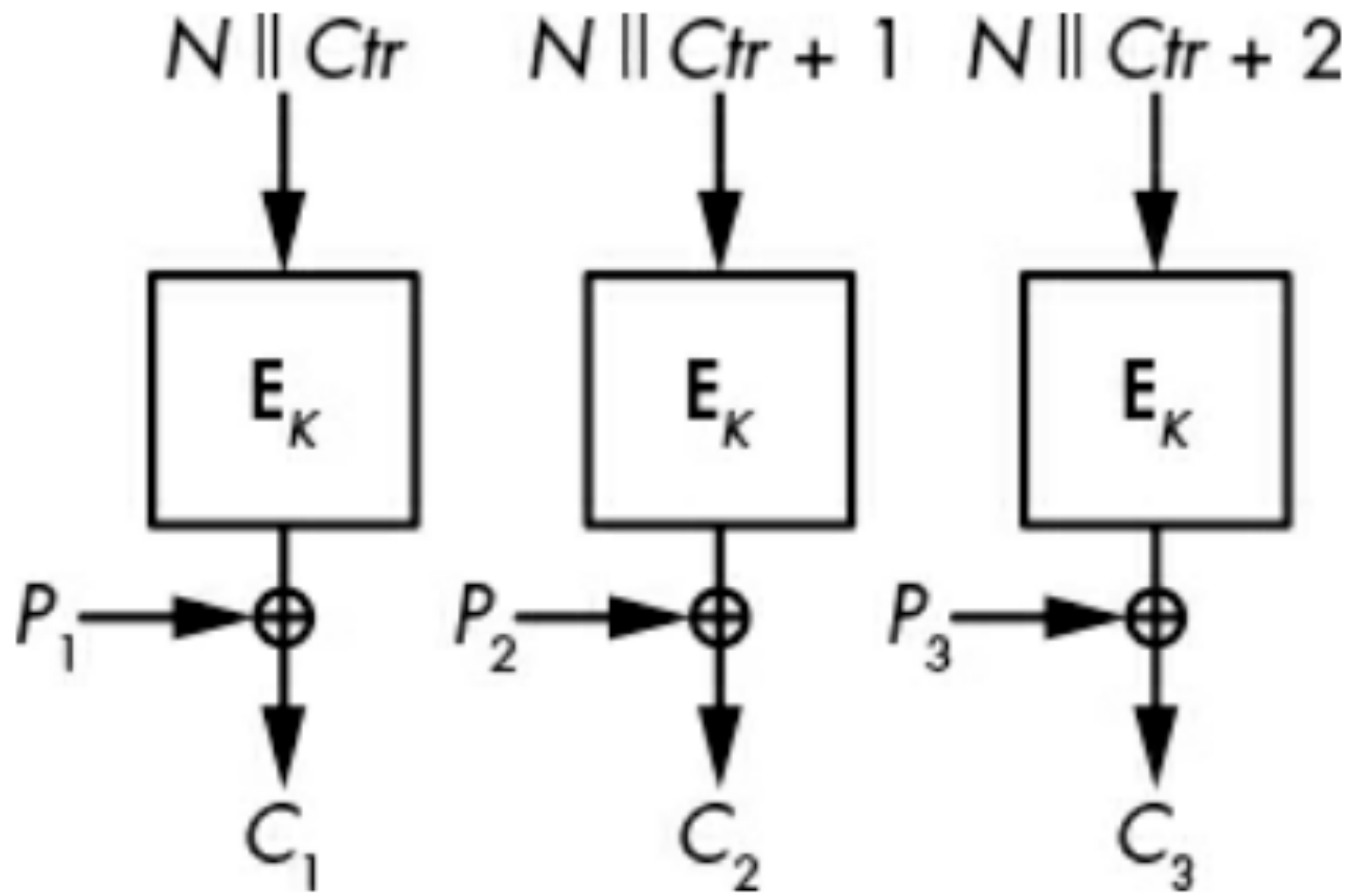


Figure 4-10: The CTR mode

- Produces a pseudorandom byte stream
- XOR with plaintext to encrypt

Nonce

- Use a different **N** for each message
- **N** is not secret, sent in the clear

Nonce Must not be Reused

```
$ ./aes_ctr.py
k = 130a1aa77fa58335272156421cb2a3ea
enc(00010203) = b23d284e
enc(b23d284e) = 00010203
```

- The first block produces the same bitstream if a nonce and key are re-used

No Padding

- CTR mode uses a block cipher to produce a pseudorandom byte stream
- Creates a stream cipher
- Message can have any length
- No padding required

Parallelizing

- CTR is faster than any other mode
- Stream can be computed in advance, and in parallel
- Before even knowing the plaintext

How Things Can Go Wrong

Two Attacks

- Meet-in-the-middle
- Padding oracle

Meet-in-the-Middle Attacks

- 3DES does three rounds of DES
- Why not 2DES?

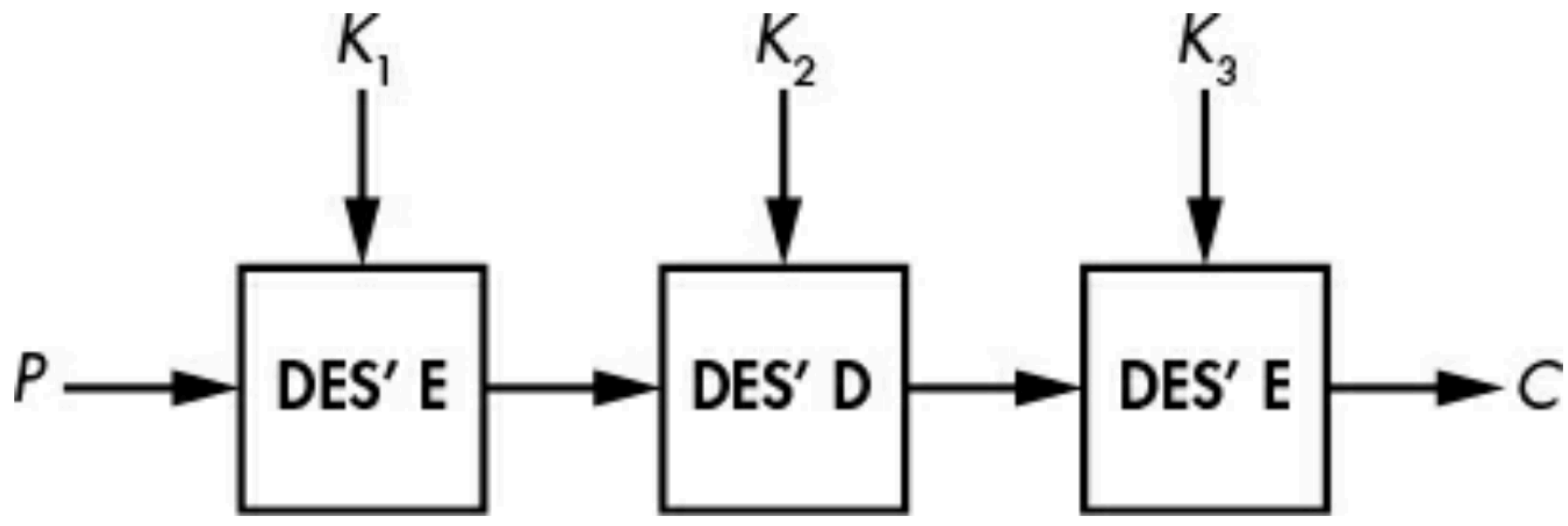


Figure 4-11: The 3DES block cipher construction

Attacking 2DES

- Two 56-bit keys, total 112 bits
- End-to-end brute force would take 2^112 calculations

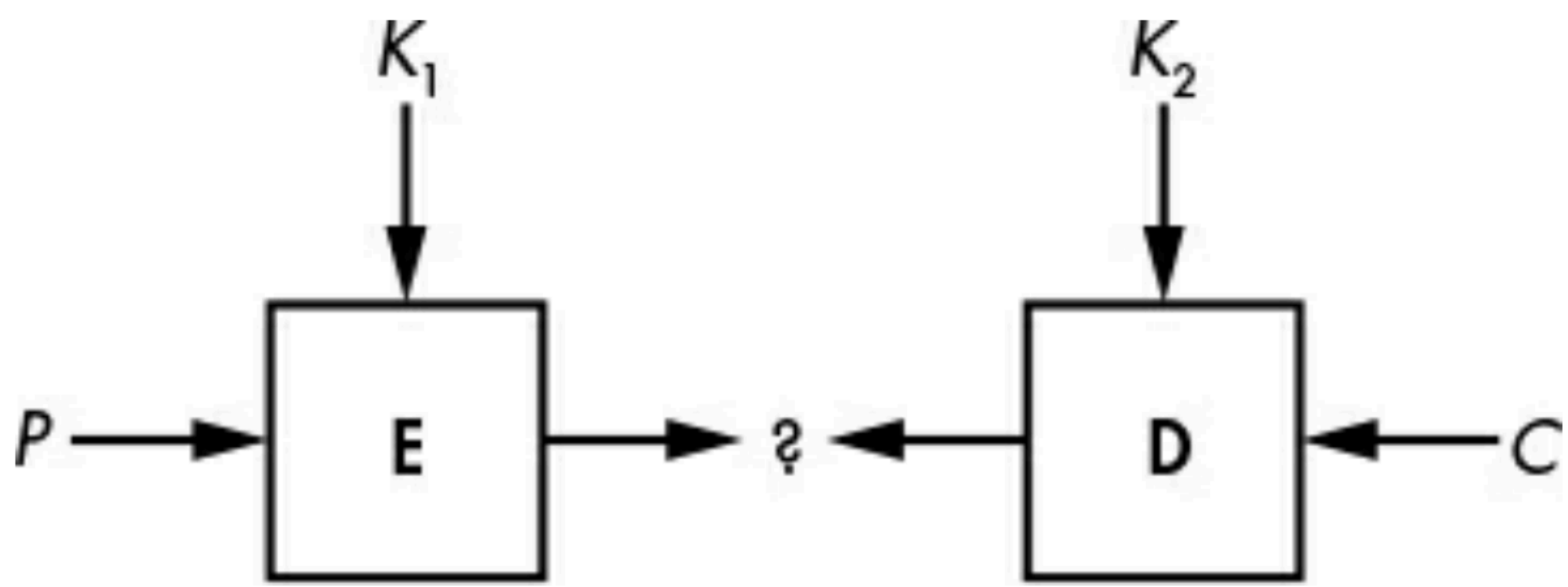


Figure 4-12: The meet-in-the-middle attack

Attacking 2DES

- Attacker inputs known ***P*** and gets ***C***
- Wants to find $\mathbf{K_1}$, $\mathbf{K_2}$

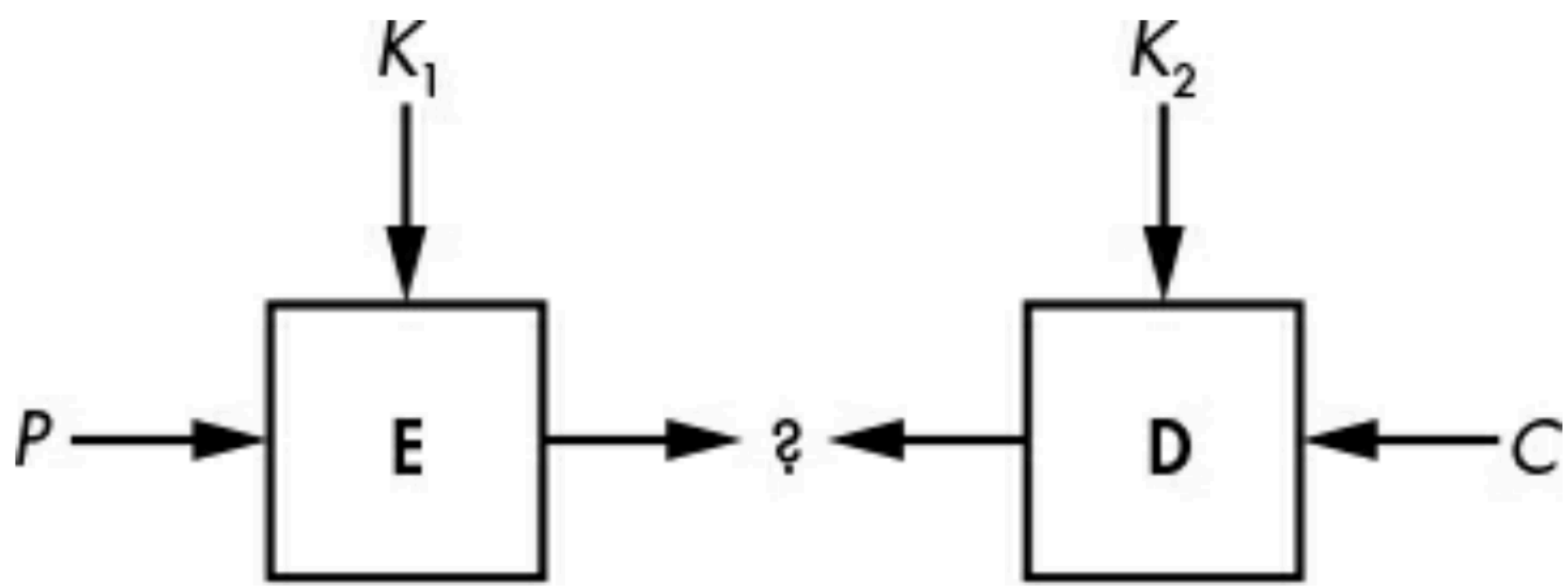


Figure 4-12: The meet-in-the-middle attack

Attacking 2DES

- Make a list of ***E***(**K_1**, ***P***) for all 2^56 values of **K_1**
- Make a list of ***D***(**K_2**, ***P***) for all 2^56 values of **K_2**
- Find the item with the same values in each list
- This finds **K_1** and **K_2** with 2^57 computations

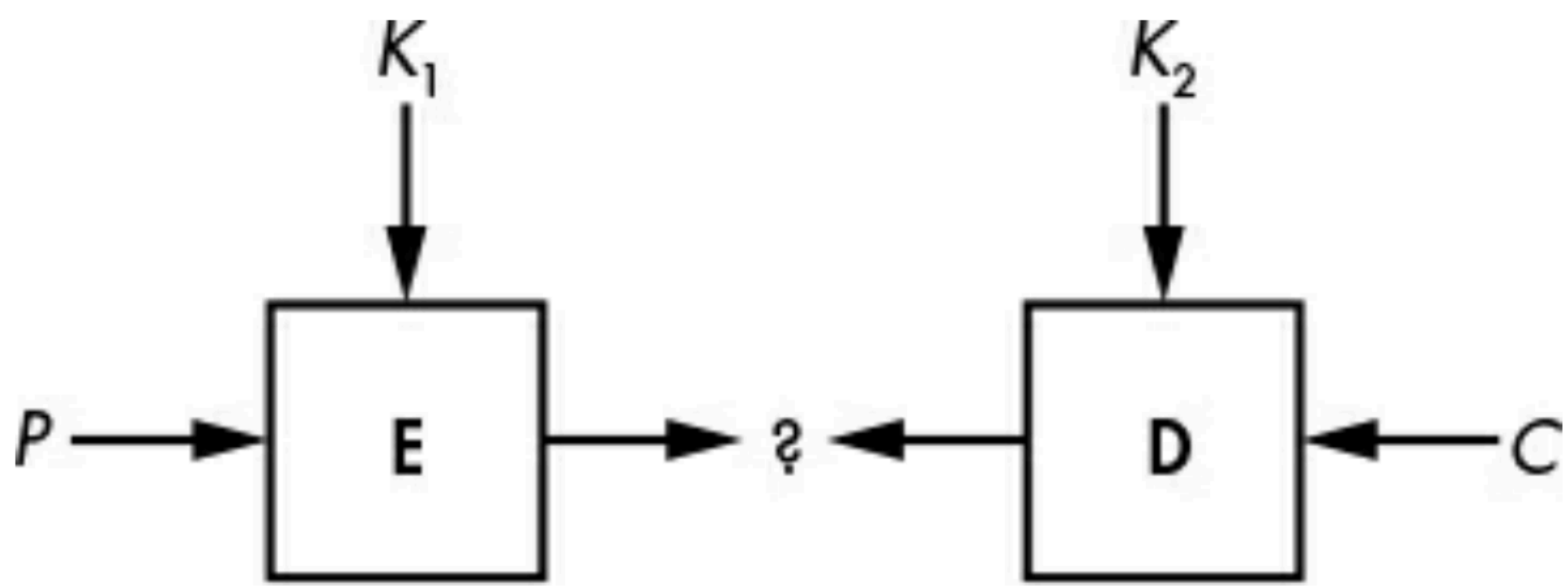


Figure 4-12: The meet-in-the-middle attack

Meet-in-the-Middle Attack on 3DES

- One table has 2^56 entries
- The other one has 2^112 entries
- 3DES has 112 bits of security

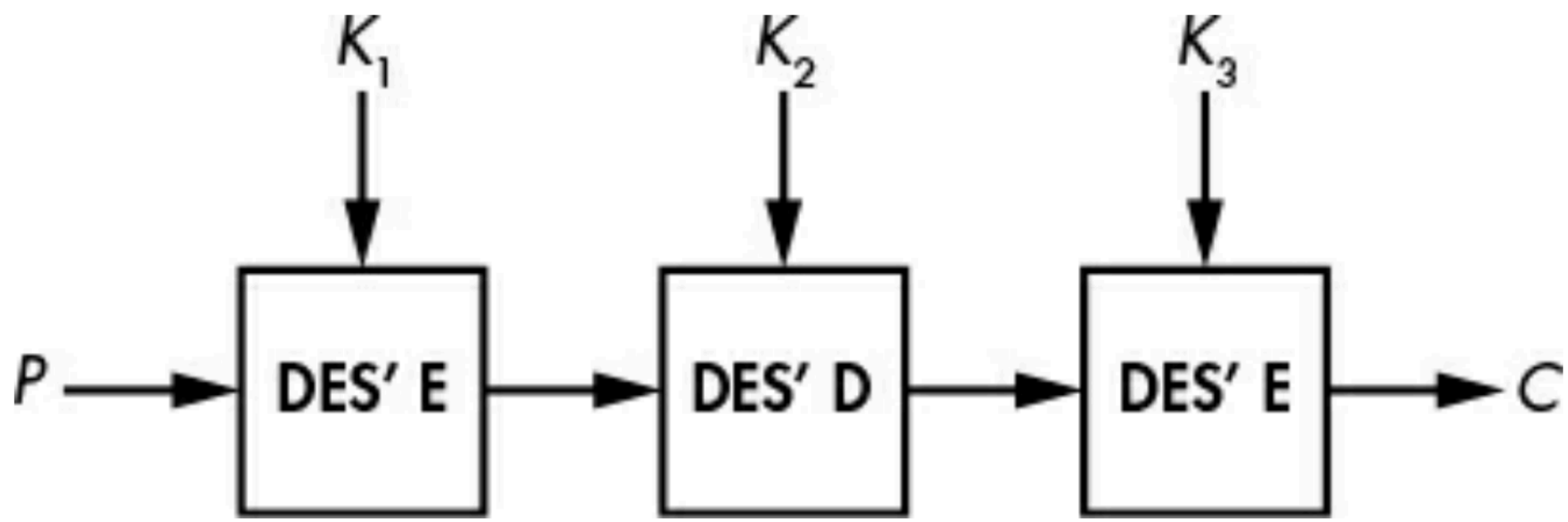
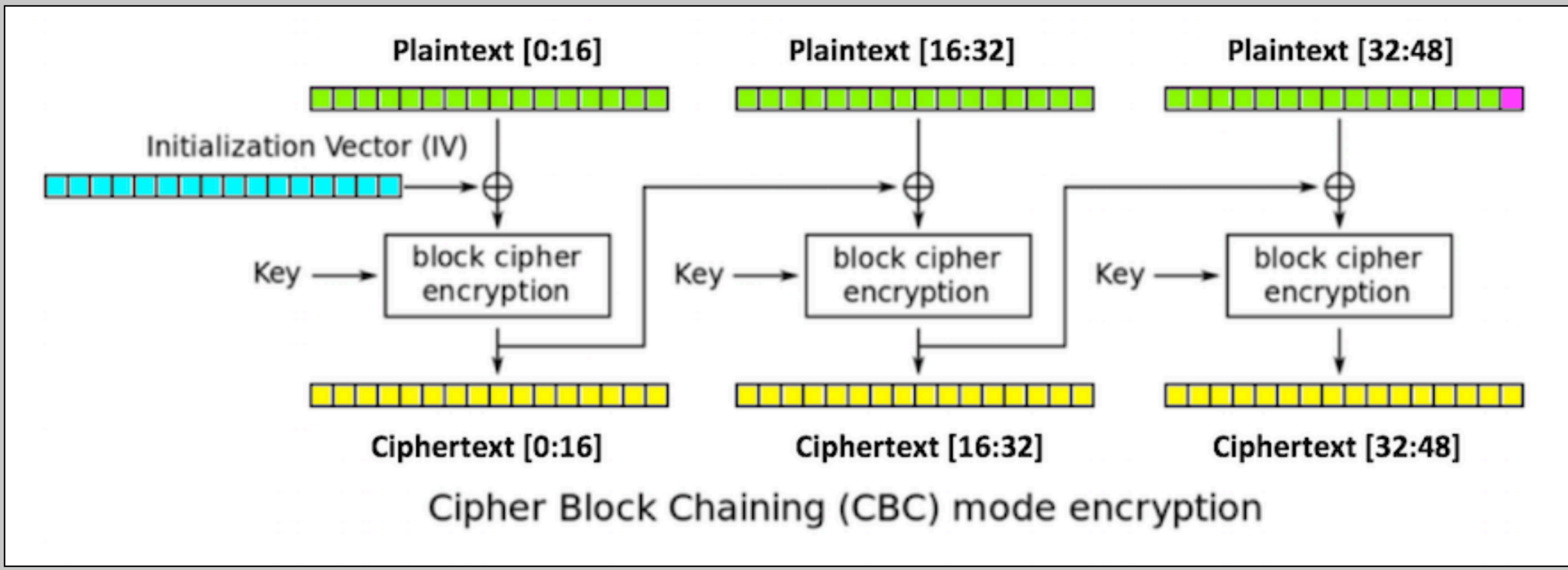


Figure 4-11: The 3DES block cipher construction

Padding Oracle

Encrypting 47 Bytes

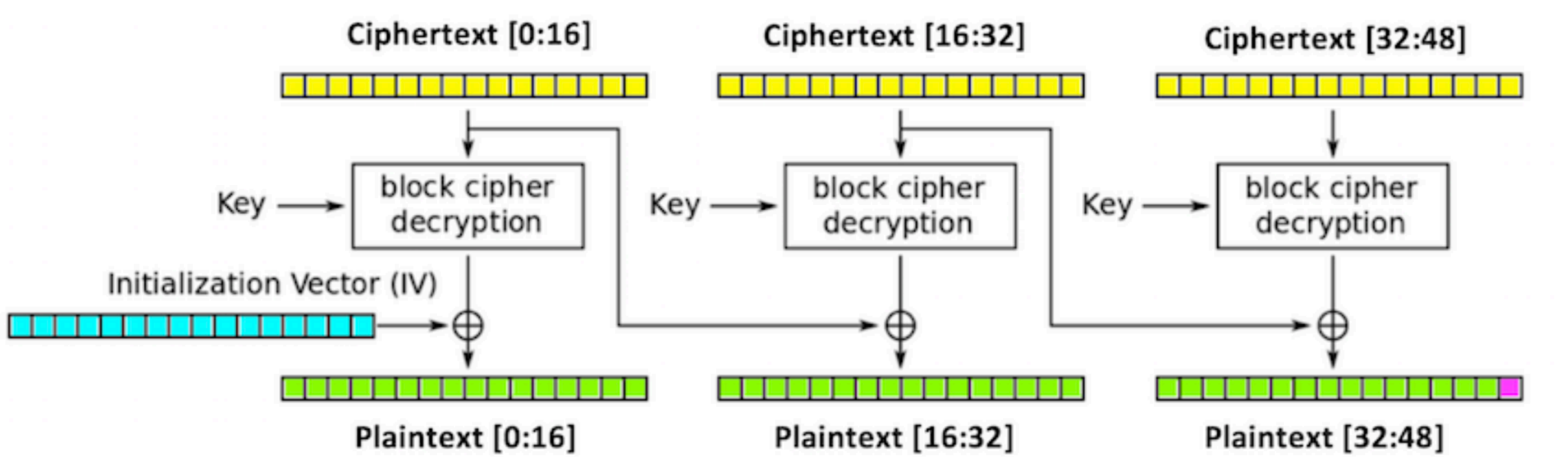
Suppose the plaintext is only 47 bytes long. In that case, a byte of **padding** (purple) must be added, as shown below.



Padding Oracle

Decryption

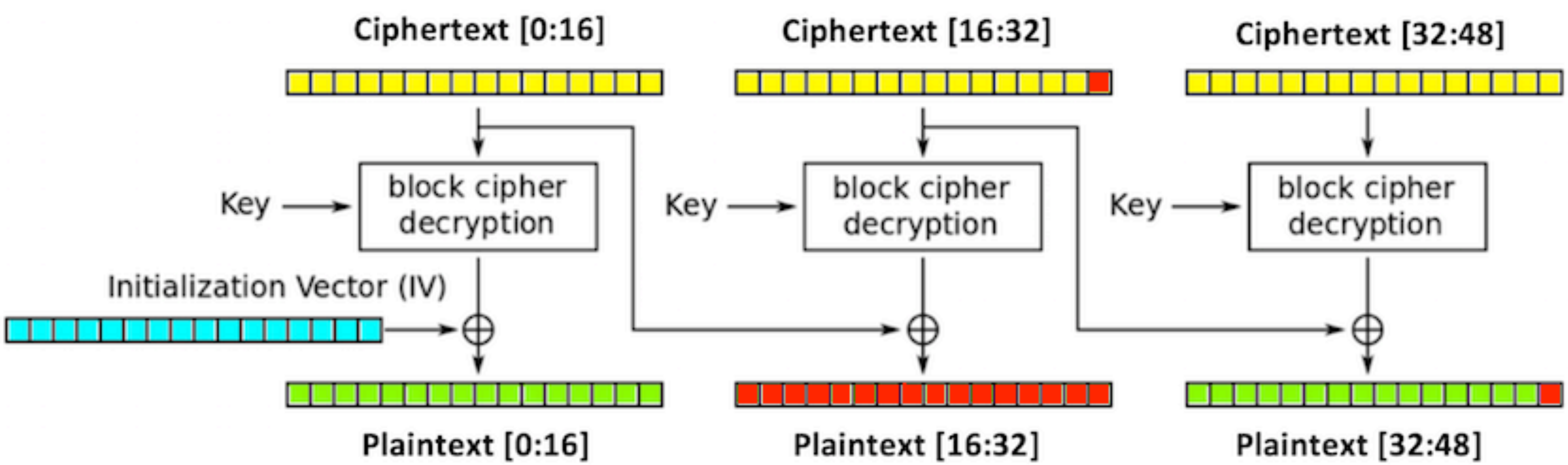
Using the same key and **iv** (blue), the **ciphertext** (yellow) can be decrypted to find the **plaintext** (green). The last byte is the **padding** (purple), as shown below.



Cipher Block Chaining (CBC) mode decryption

Padding Oracle

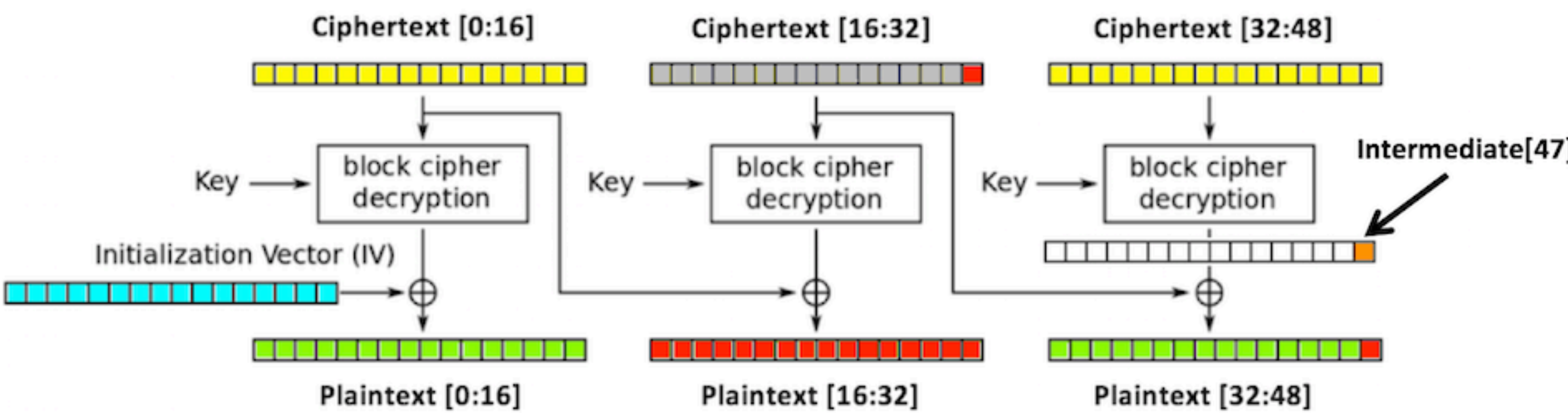
- Change the last byte in second block
- This changes the 17 bytes shown in red



Cipher Block Chaining (CBC) mode decryption

Padding Oracle

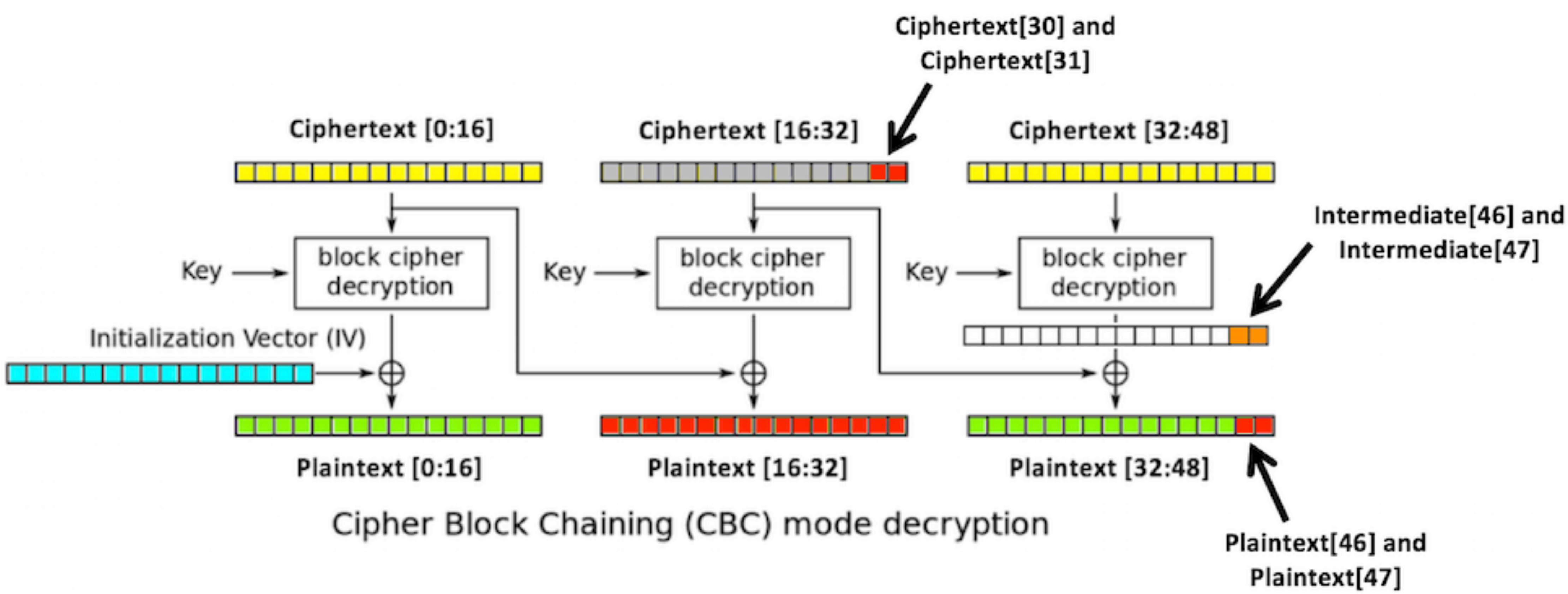
- Try all 256 values of last byte in second block
- One of them has valid padding of '\x01'
- This determines the orange byte



Cipher Block Chaining (CBC) mode decryption

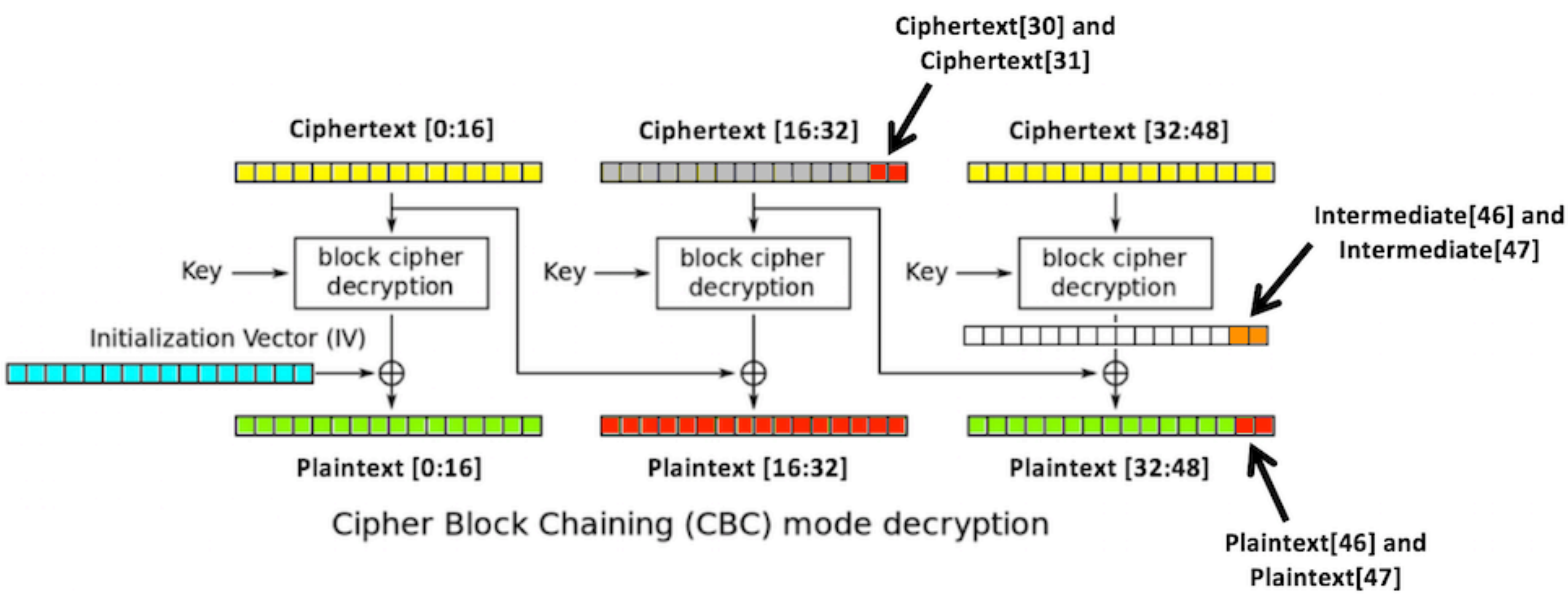
Padding Oracle

- Continue, 256 guesses finds the next orange byte

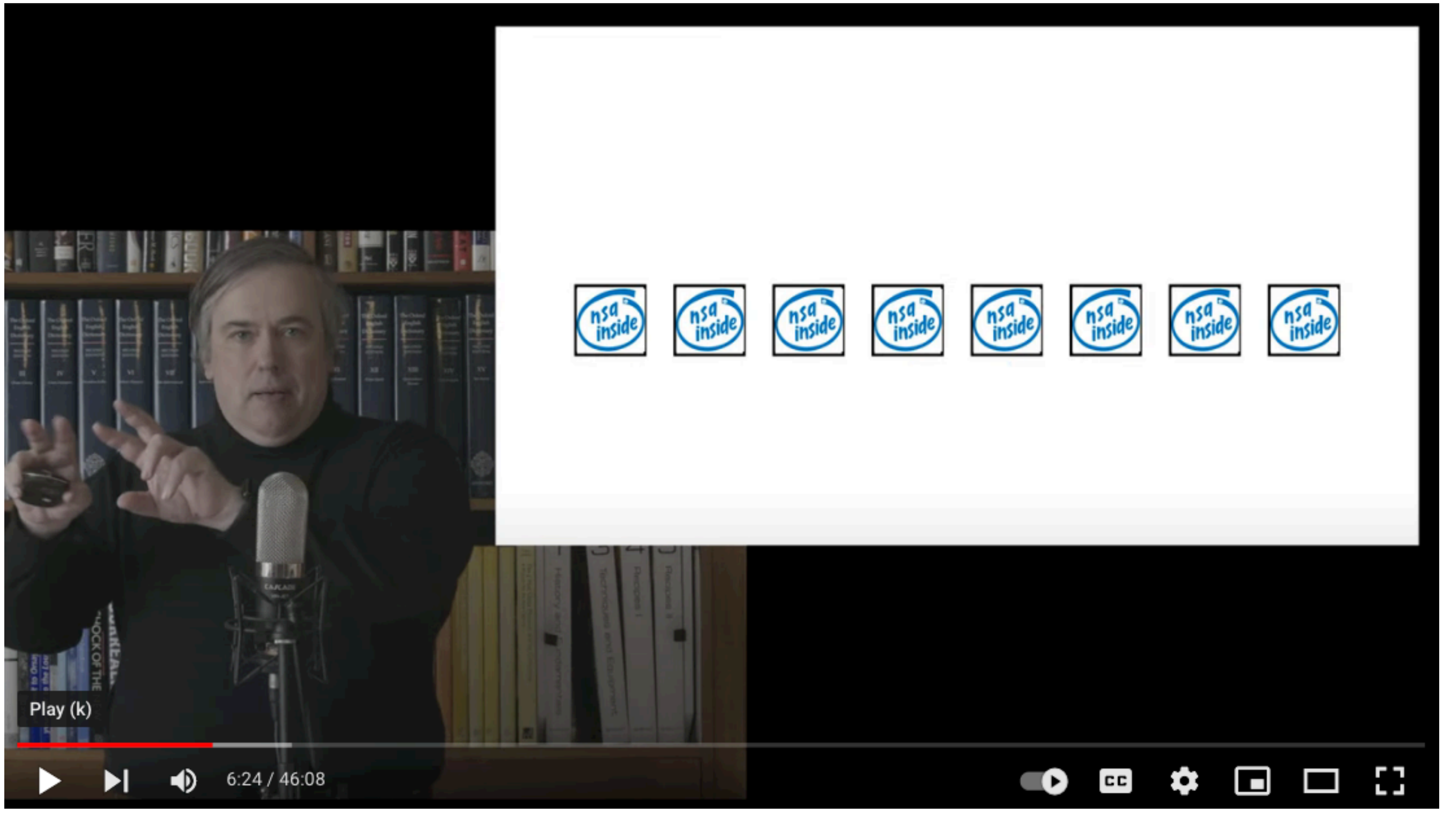


Padding Oracle

- Once the orange bytes are known
  - Attacker can forge plaintext[32:48] to say anything



NSA



- 3:13 - 8:40
- https://www.youtube.com/watch?v=tPEi0mnUY_o

NOBUS

- The NSA wants us to use cryptography that is
  - Weak enough so the NSA can break it
  - Strong enough so no one else can break it
- "Nobody But Us"
- This depends on the NSA keeping secrets

Edward Snowden

- Leaked NSA secrets in 2013
- https://www.youtube.com/watch?v=0hLjuVyIIrs

DES and the NSA

- The NSA first weakened DES
  - Shortening the 128-bit key in the original "Lucifer" system from IBM to 56 bits
- The NSA also strengthened it
  - Improving the "S-Box" to resist differential cryptanalysis, a technique that was secret at the time
- This created a NOBUS system

4b