TECHNISCHE UNIVERSITÄT WIEN
Vienna | Austria

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

SECURITY & PRIVACY GROUP

Introduction to Security (UE, 192.082)

Lecture 4: Crypto Challenges

Dr. Marco Squarcina & Dr. Mauro Tempesta

The RSA Cryptosystem

RSA in a Nutshell

- Let p, q be two large primes and $N = p \cdot q$
- Let ϕ be the Euler's totient function:

$$\begin{aligned} \phi(N) &= \left| \{k \mid 1 \leq k \leq N \wedge \gcd(k, N) = 1\} \right| \\ &= (p-1) \cdot (q-1) \end{aligned}$$

Public key: (e, N) **Private key:** (d, N)

with $d \cdot e \equiv 1 \pmod{\phi(N)}$

Encryption

$C \equiv P^e \pmod{N}$

Decryption

$P \equiv C^d \pmod{N}$

Modulus Factorization

- The security of RSA can be trivially broken if you **factor** N
  - that's why we require large primes p, q!

- From the factorization you can compute $\phi(N)$ and recover the **private exponent** d

$$d = e^{-1} \pmod{\phi(N)}$$

inverse of e modulo $\phi(N)$

Issues with Small Messages

- We know that $C \equiv P^e \pmod{N} \Rightarrow C + k \cdot N = P^e \quad (k \geq 0)$
- It's common to pick small values for e to speed-up encryption (e.g., 3 or 17)
- If the plaintext P is also small, the ciphertext C is not "much bigger" than N

$$P = \sqrt[e]{C + k \cdot N}$$

bruteforce

Issues with Small Messages

- We know that $C = P^e \pmod{N} \Rightarrow C + k \cdot N = P^e \quad (k \geq 0)$
- It's co… encryp…
- If the p… t "much bigger" than N

Solutions?
Any ideas?

$$P = \sqrt[e]{C + k \cdot N}$$

bruteforce

SOLUTION!!!!


Padding
Padding
Padding
Plaintext
Padding
Padding
Padding
Modulus

Message Padding

- Make the plaintext almost *as big as* N by adding **padding**
  - Use a padding scheme that makes it possible to easily distinguish plaintext and padding

- **DON'T** use deterministic padding schemes, always add some **randomness** to the plaintext
  1) The same message encrypted multiple times using different padding yields different ciphertexts
  2) … and there exist some trivial attacks that can be performed if you have multiple ciphertexts of the same message

Common Modulus Attack

- Suppose that the message P is encrypted twice under different public keys sharing the same modulus N

$$\text{Pubkey 1: } (e_1, N) \quad \Rightarrow \quad C_1 \equiv P^{e_1} \pmod{N}$$

$$\text{Pubkey 2: } (e_2, N) \quad \Rightarrow \quad C_2 \equiv P^{e_2} \pmod{N}$$

- If $\gcd(e_1, e_2) = 1$, by Bezout's identity there exist integers x, y such that

$$x \cdot e_1 + y \cdot e_2 = 1$$

use the Extended Euclidean algorithm to compute them

- Then we have

$$C_1^x \cdot C_2^y = P^{x \cdot e_1 + y \cdot e_2} \equiv P \pmod{N}$$

More on Padding

- Another attack uses multiple ciphertexts of the same message encrypted under the same public exponent but different moduli
  - find out the plaintext using the Chinese Remainder Theorem
- So… rely on (standard) padding schemes that add some randomness to the plaintext!
  - OAEP (PKCS#1 v2)
- Got it! Are we done now?

Coppersmith's Short Pad Attack

Boneh & Durfee's Attack

Wiener's Attack

YES

well, only for this lecture…

Timing Attacks

Partial Key Exposure Attacks

Padding Oracle Attacks (Bleichenbacher)

Vulnerable Block Cipher Modes

Symmetric Encryption

- Symmetric ciphers operate on blocks of a **fixed** length
  - DES: 64 bits
  - AES: 128 bits
- When a message is longer than the cipher block size, it is **split** in several blocks

- Block **cipher modes** determine how the different blocks are handled during encryption / decryption
  - ECB, CBC, OFB, CTR, GCM, …

- What if the last block is smaller than the block size?
  - I guess you know the answer…

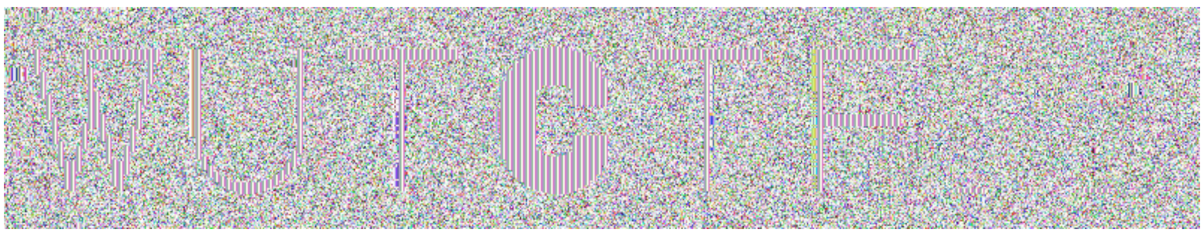
SOLUTION!!!!
Padding
Padding
Padding
Last block
Padding
Padding
Padding
Block size

PKCS#7 Padding

- In PKCS#7 padding:
  - If we need to add n bytes to the plaintext so that its size is a multiple of the block size, append n bytes of value n to the plaintext
  - If n = 0, add an entire block of padding bytes
- Example (block size = 8 bytes, hex-encoded):
  - Plaintext: `6369616f`
  - Padded: `6369616f04040404`

ECB is bad… stop using it… please!

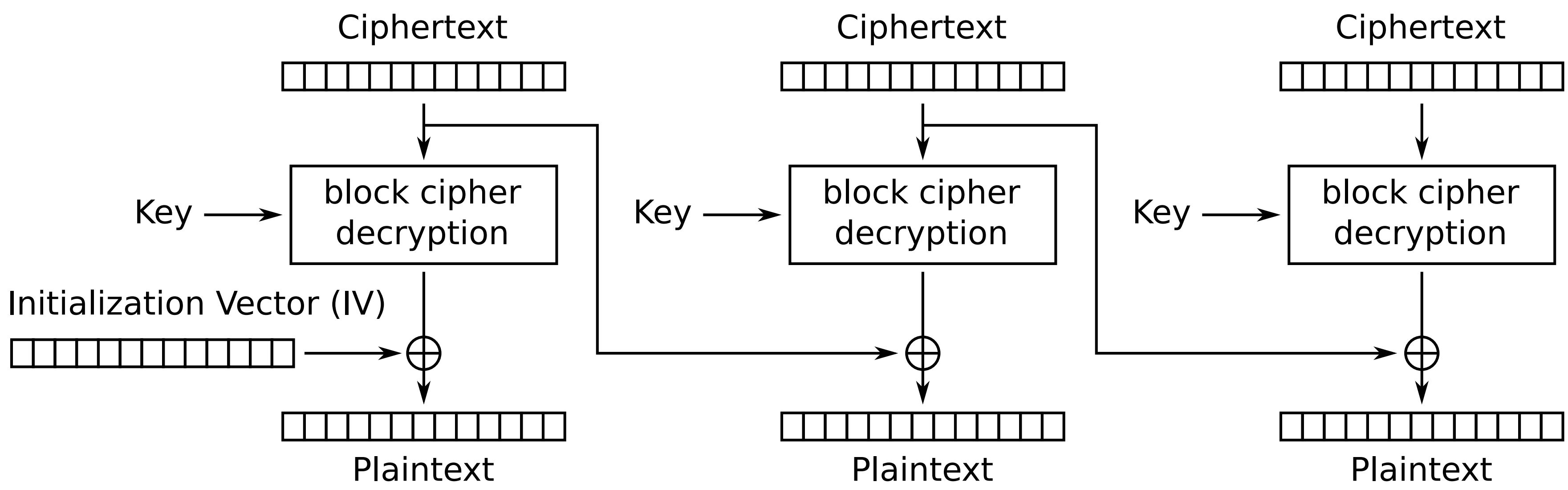
Plaintext



ECB

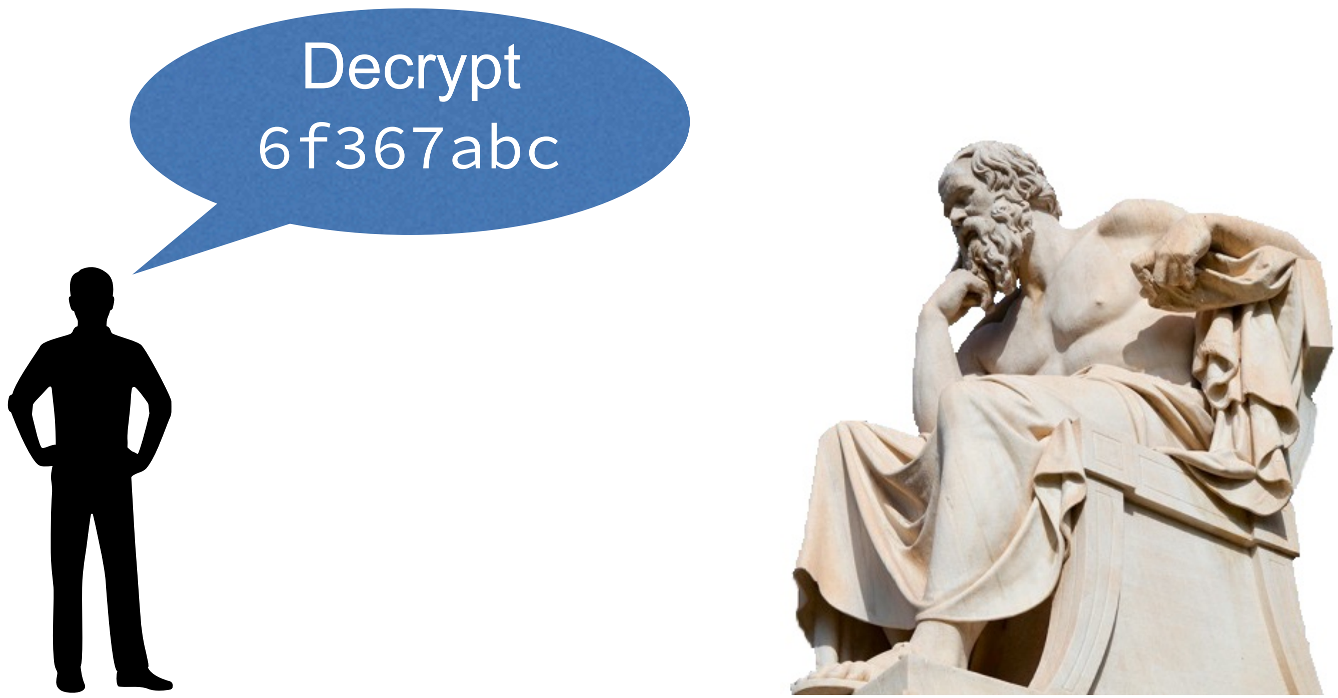
CBC

CBC Mode (Decryption)



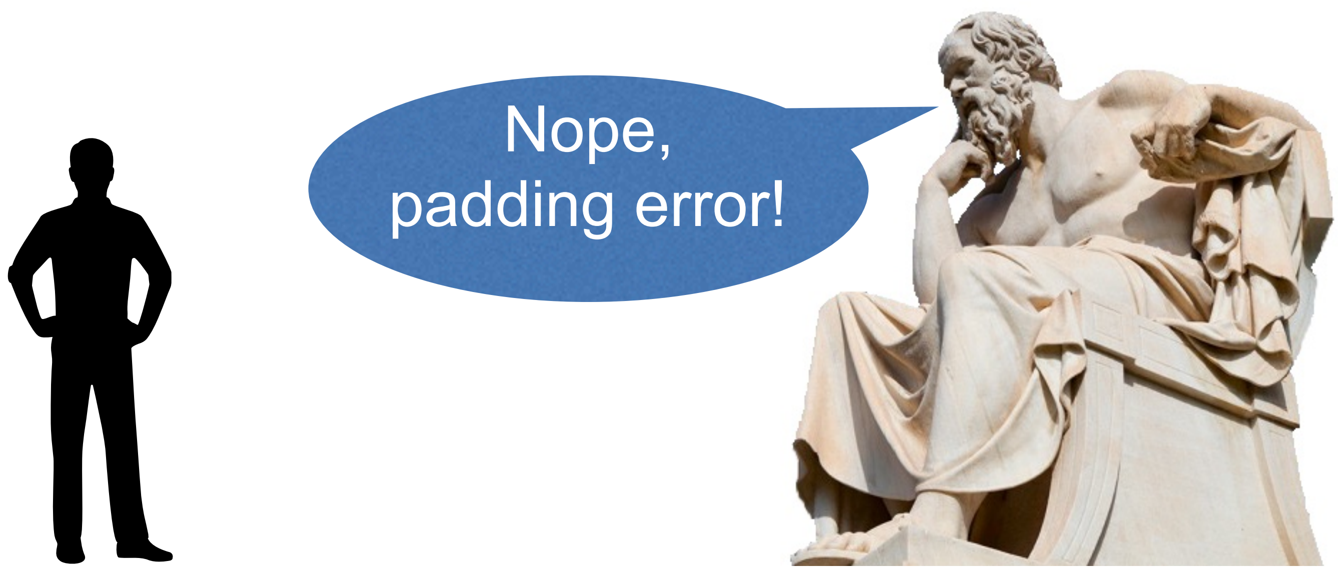
CBC Padding Oracle Attack

- Published by Serge Vaudenay in 2002
  - Later used against SSL, IPsec, Ruby on Rails, Steam…
- We can find the **plaintext** without knowing the key



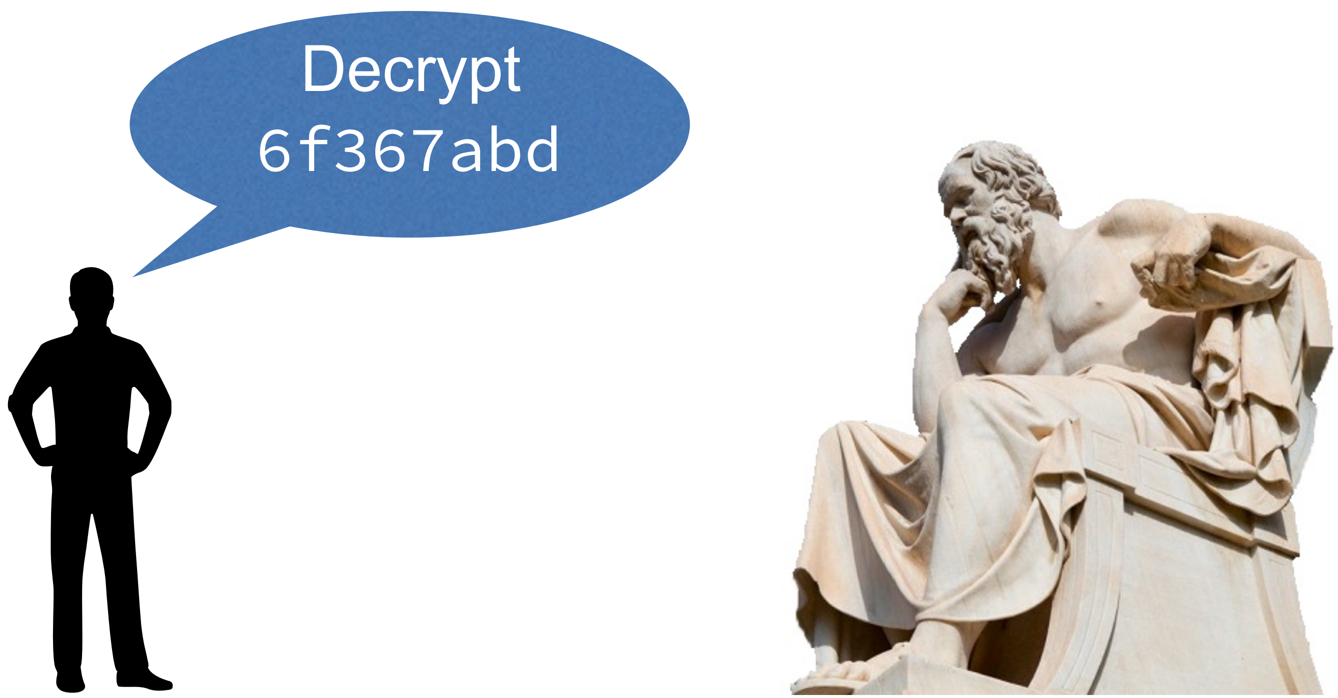
CBC Padding Oracle Attack

- Published by Serge Vaudenay in 2002
  - Later used against SSL, IPsec, Ruby on Rails, Steam…
- We can find the **plaintext** without knowing the key



CBC Padding Oracle Attack

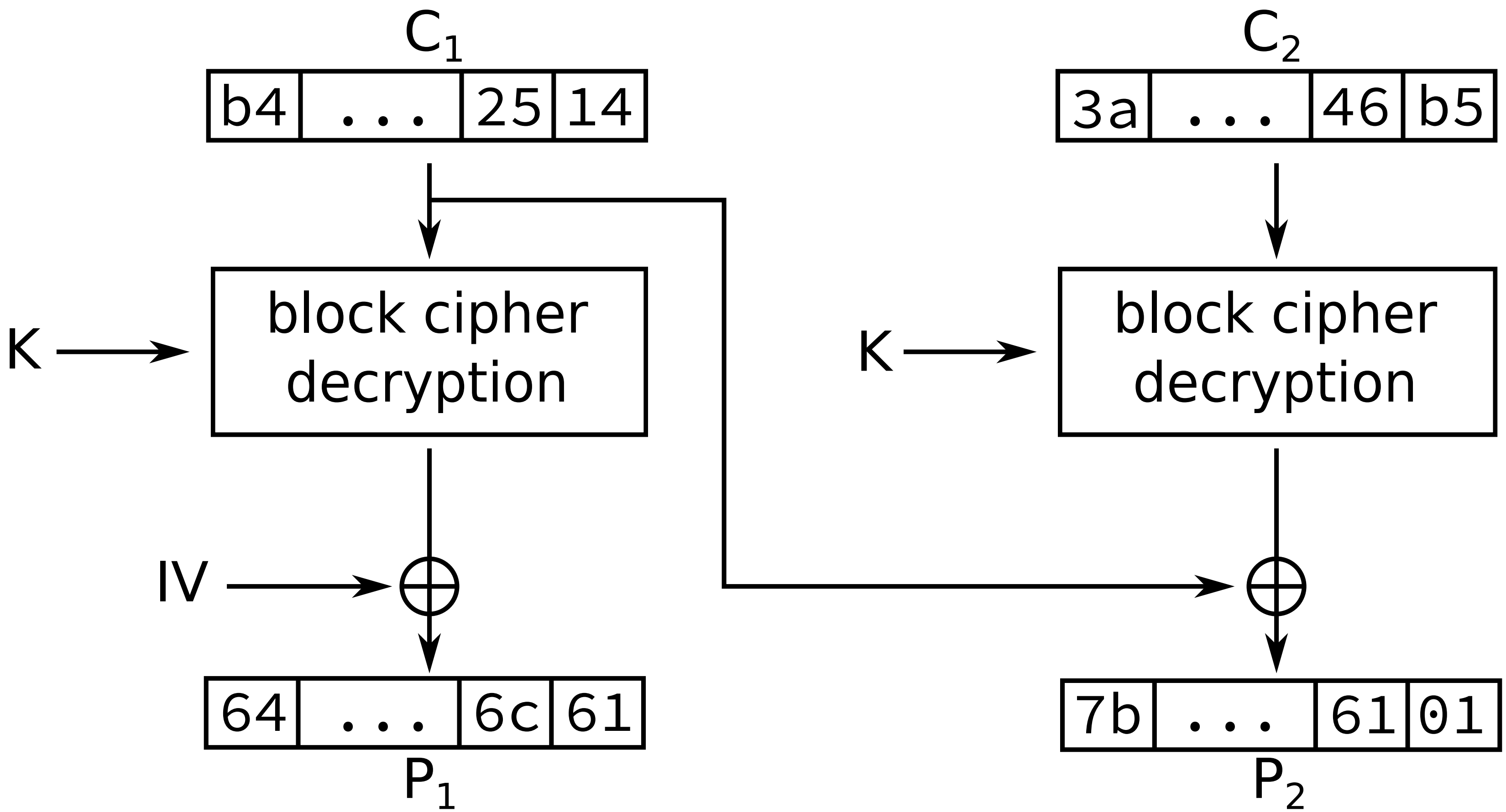
- Published by Serge Vaudenay in 2002
  - Later used against SSL, IPsec, Ruby on Rails, Steam…
- We can find the **plaintext** without knowing the key



CBC Padding Oracle Attack

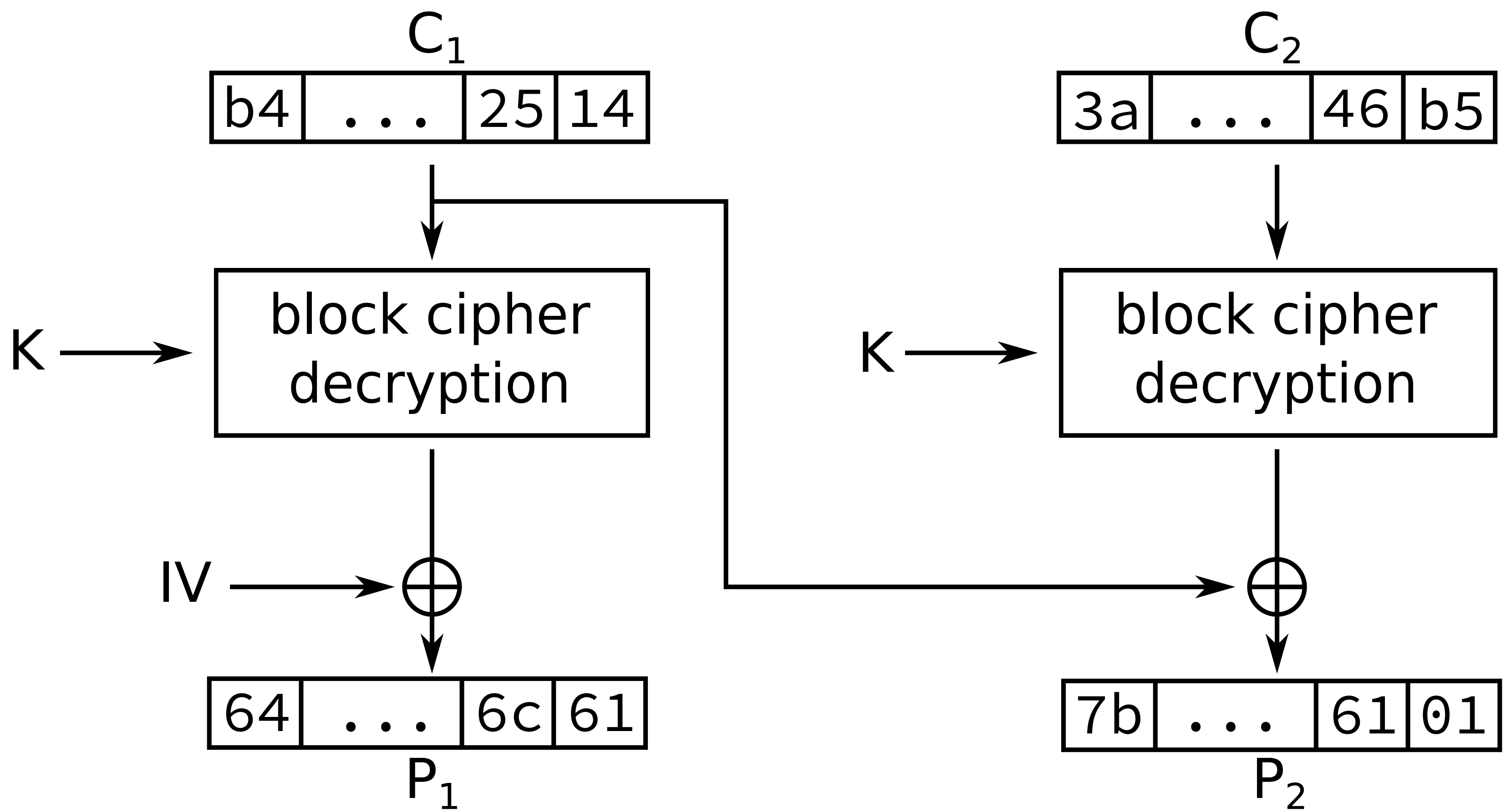
- Published by Serge Vaudenay in 2002
  - Later used against SSL, IPsec, Ruby on Rails, Steam…
- We can find the **plaintext** without knowing the key

Guessing the Last Byte of P_2



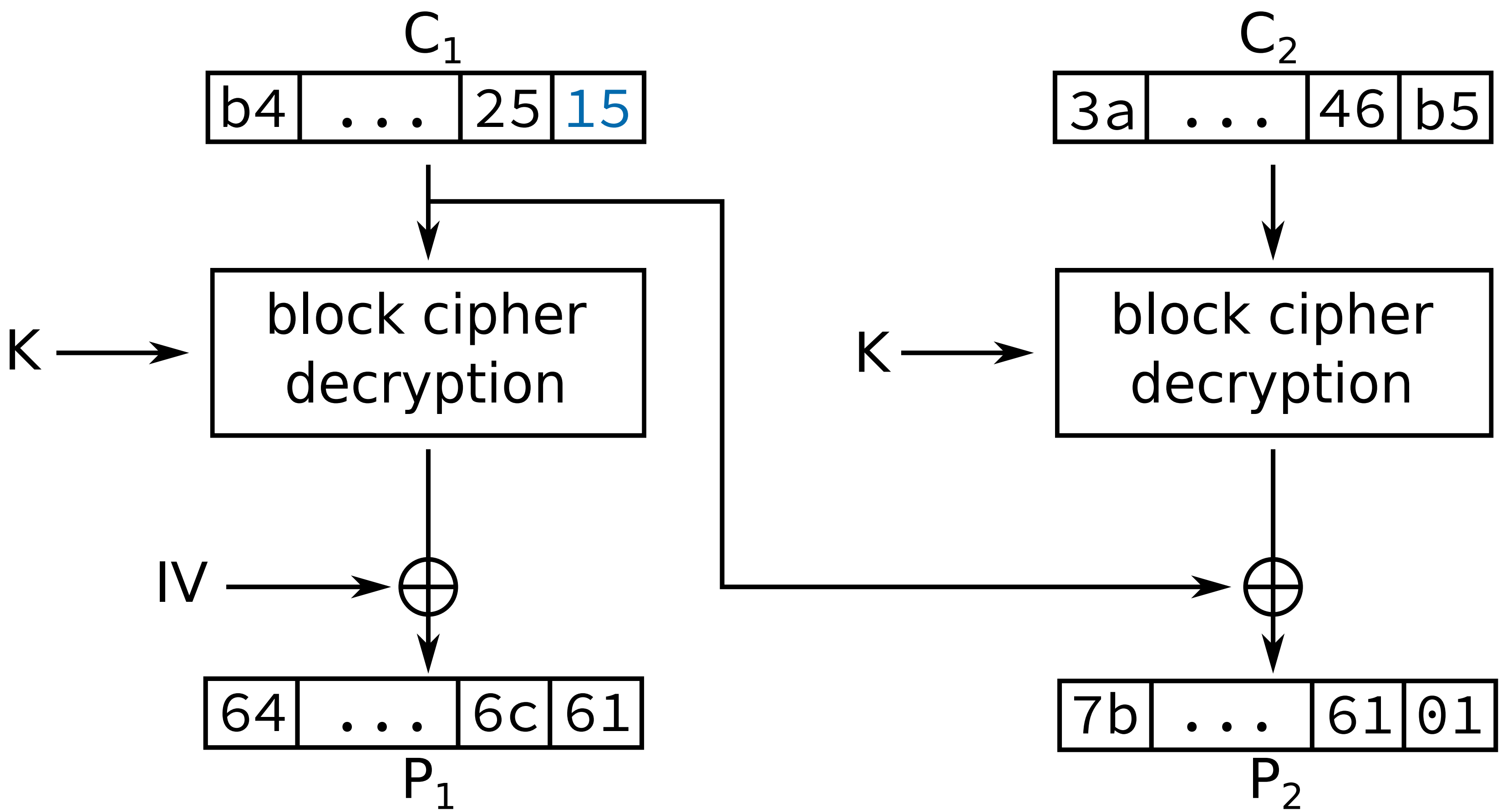
- We manipulate the last byte of C_1 to find the last byte of P_2
  - We XOR the last byte of C_1 with `01`$\oplus$G where G is our guess
  - If our guess is correct, we have no padding errors!

Guessing the Last Byte of P_2



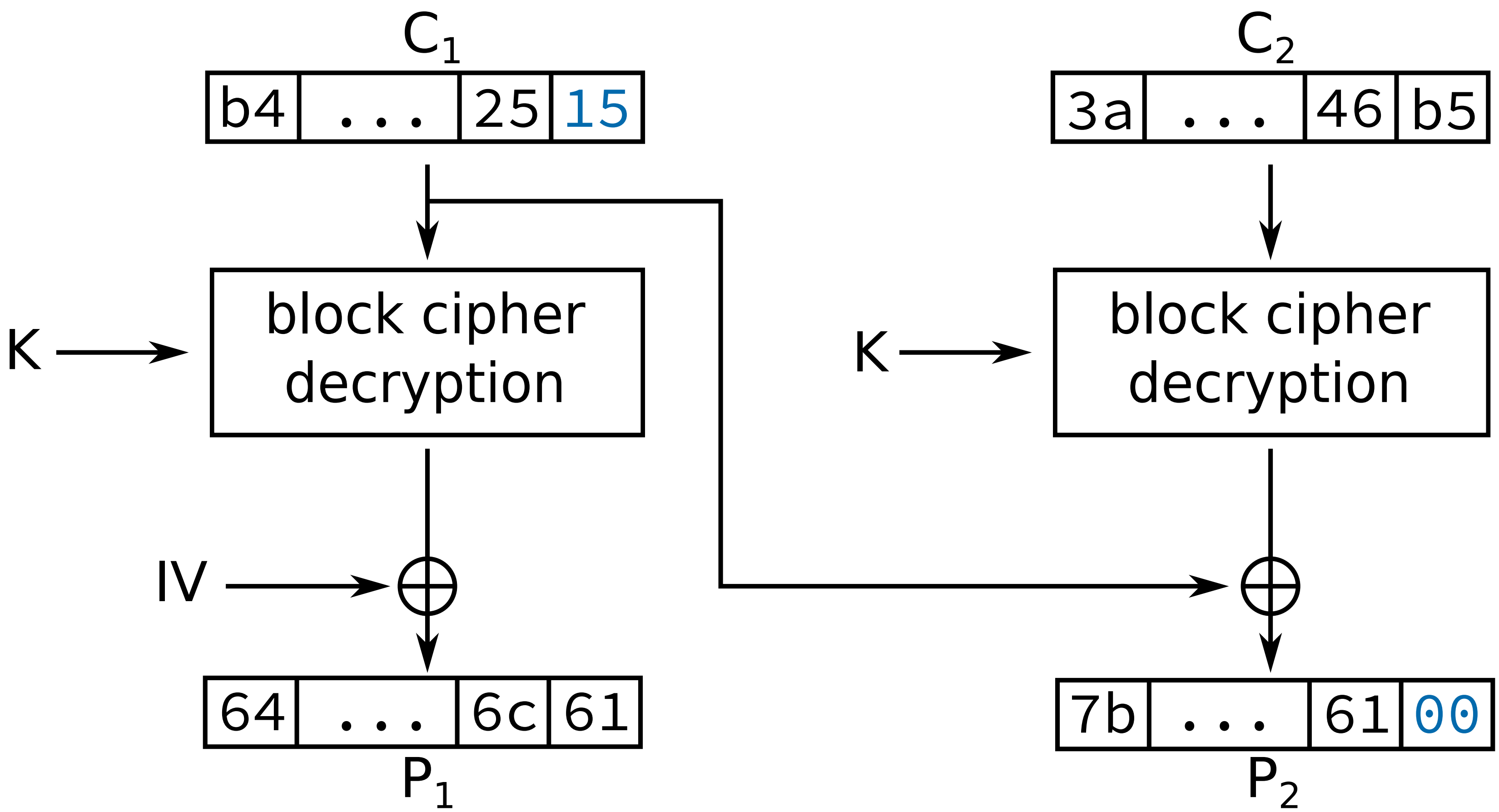
- Our guess is G = `00`

Guessing the Last Byte of P_2



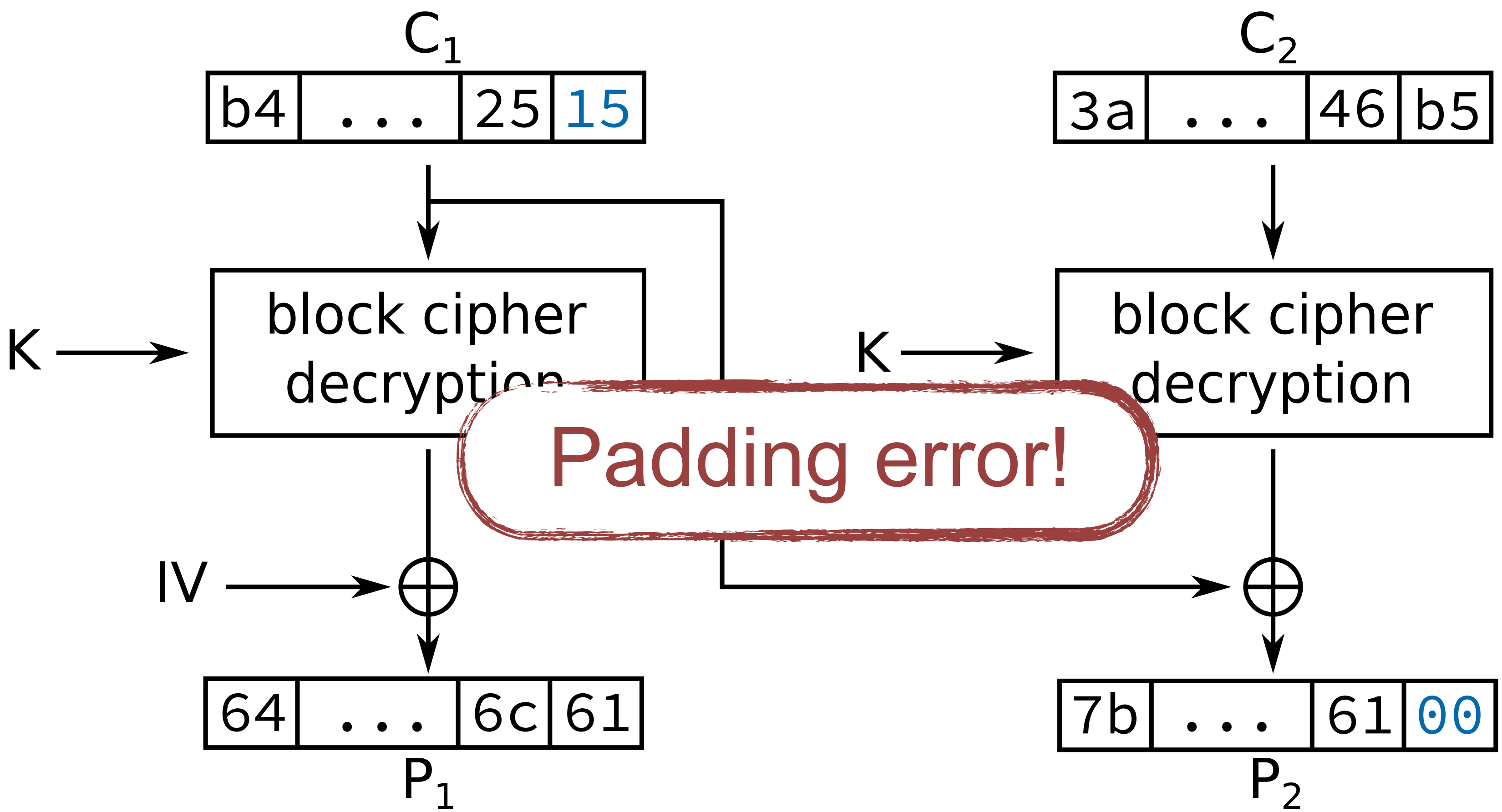
- Our guess is G = `00`
- Last byte of C_1 becomes $14 \oplus 00 \oplus 01 = 15$

Guessing the Last Byte of P_2



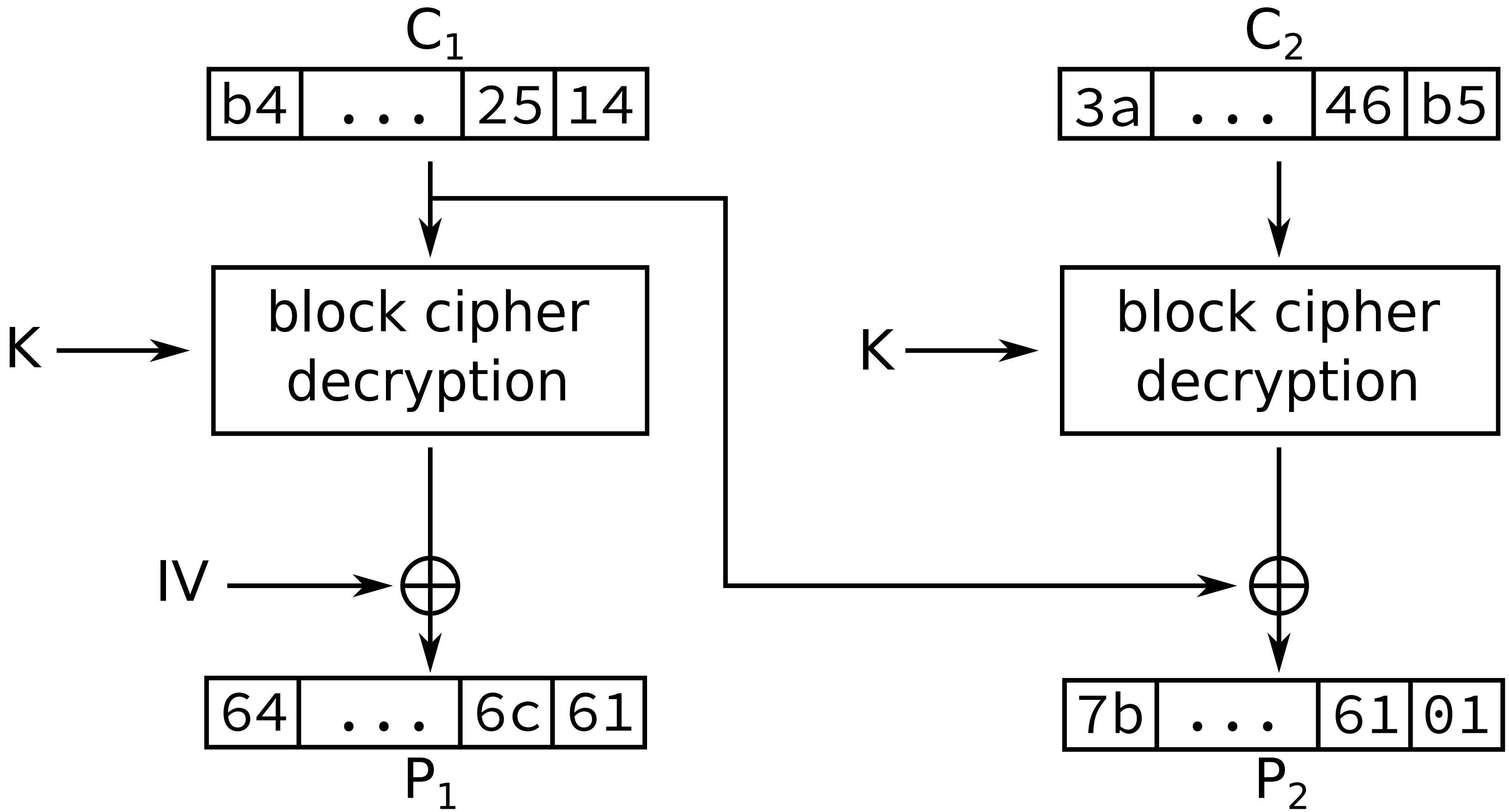
- Our guess is G = `00`
- Last byte of C_1 becomes $14 \oplus 00 \oplus 01 = 15$

Guessing the Last Byte of P_2



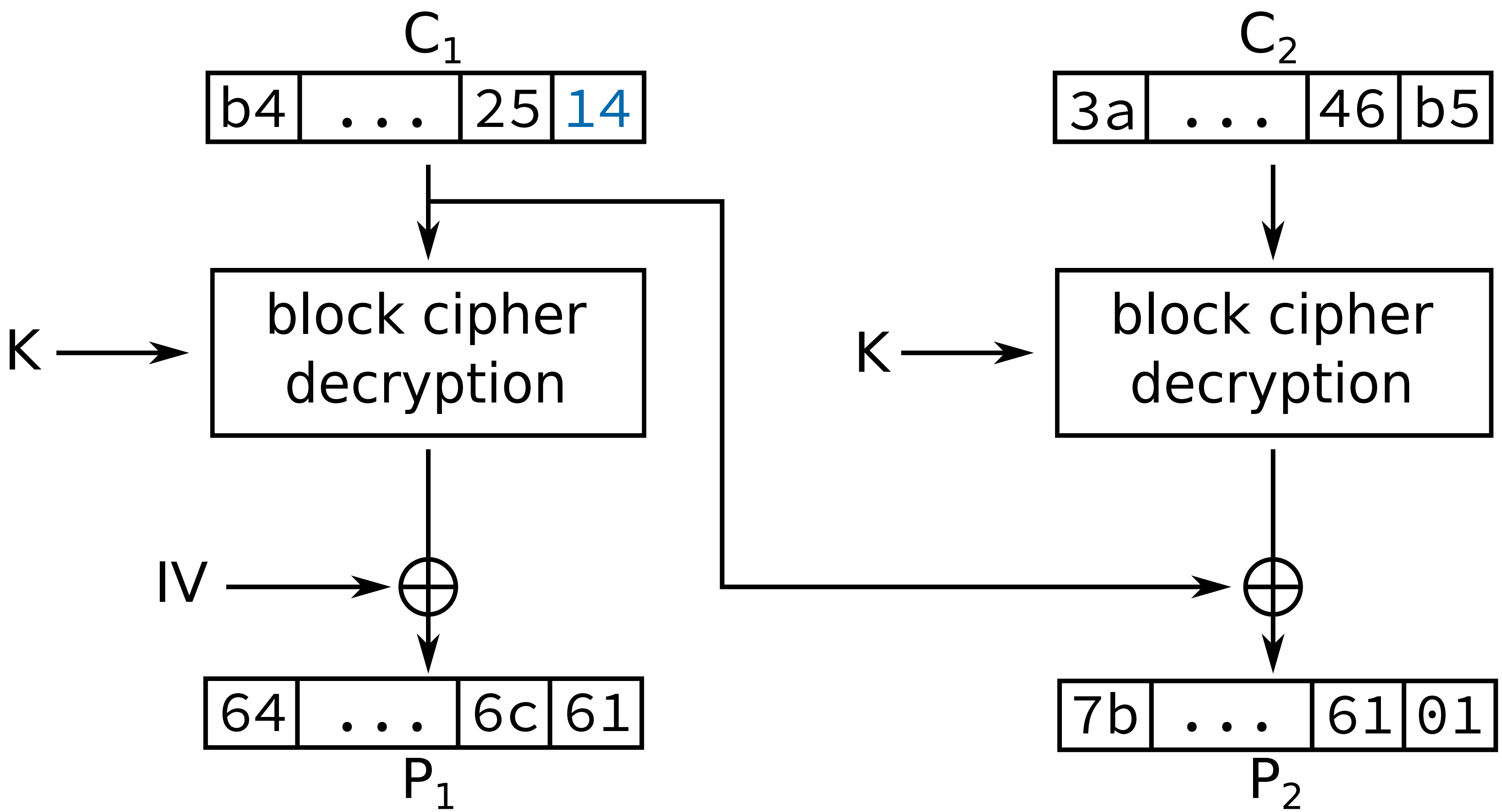
- Our guess is G = `00`
- Last byte of C_1 becomes `14` ⊕ `00` ⊕ `01` = `15`

Guessing the Last Byte of P_2



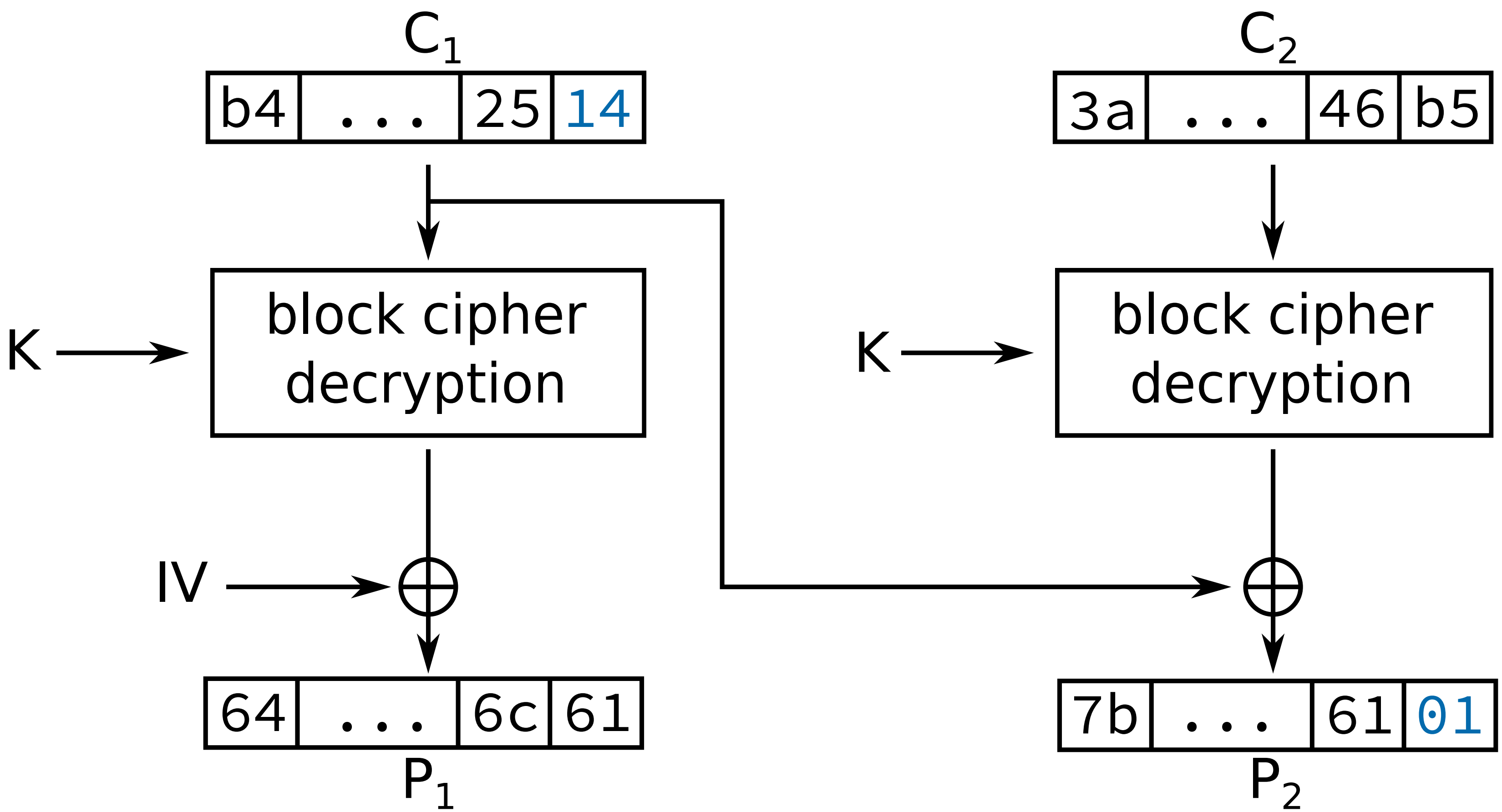
- Our guess is G = `01`

Guessing the Last Byte of P_2



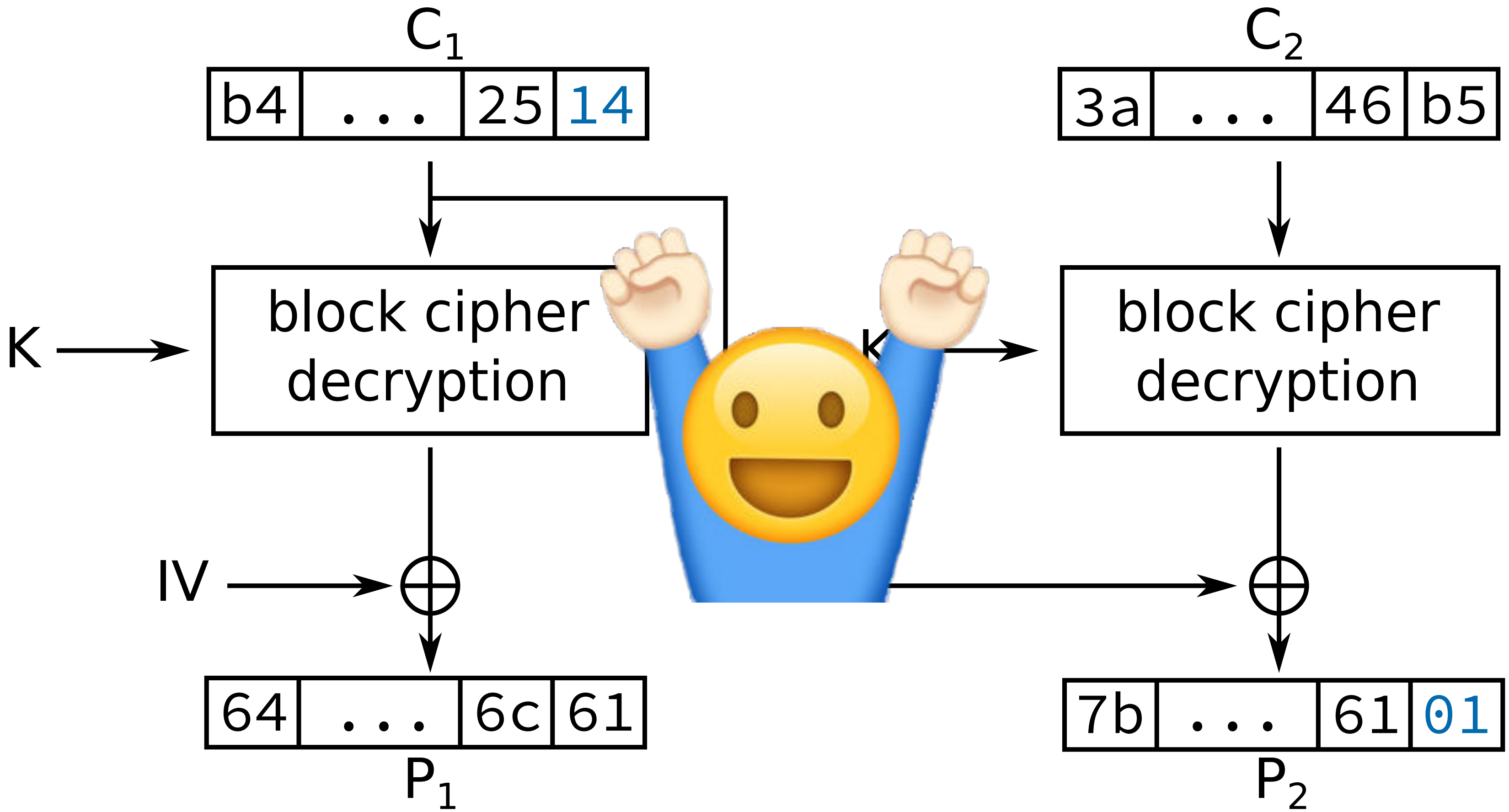
- Our guess is G = `01`
- Last byte of C_1 becomes $14 \oplus 01 \oplus 01 = 14$

Guessing the Last Byte of P_2



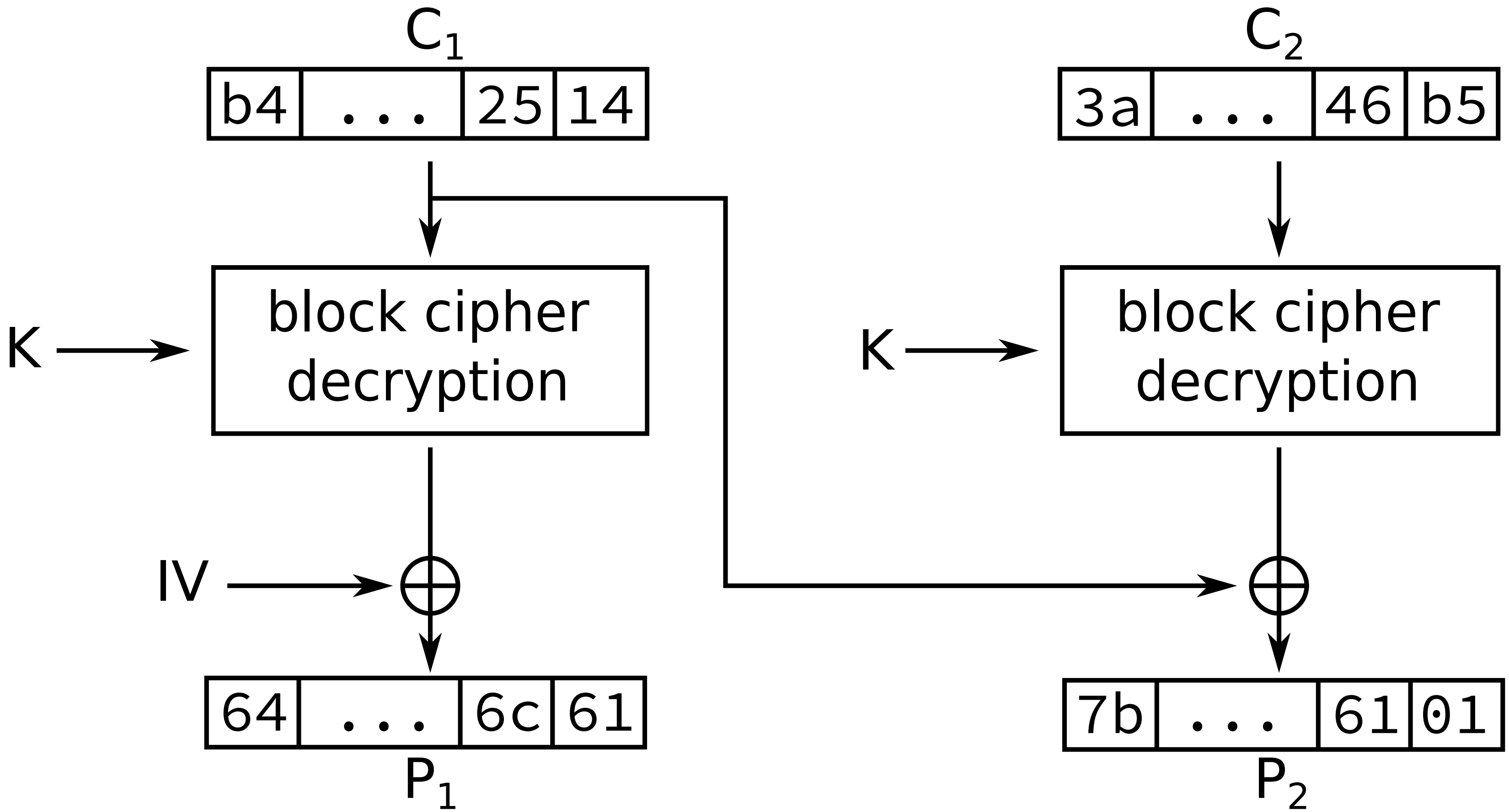
- Our guess is G = `01`
- Last byte of C_1 becomes $14 \oplus 01 \oplus 01 = 14$

Guessing the Last Byte of P_2



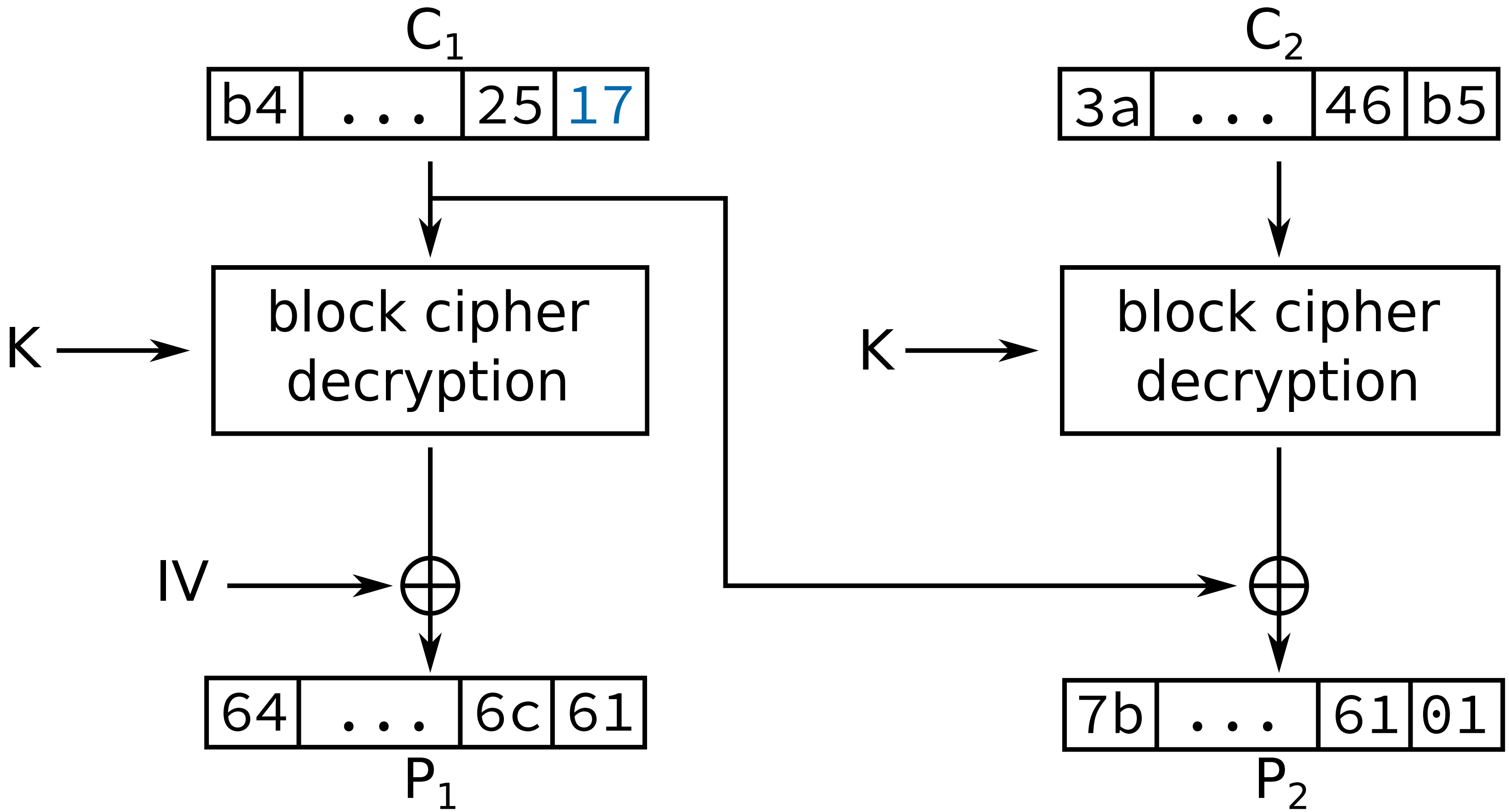
- Our guess is G = `01`
- Last byte of C_1 becomes $\texttt{14} \oplus \texttt{01} \oplus \texttt{01} = \texttt{14}$

Guessing the Penultimate Byte of P_2



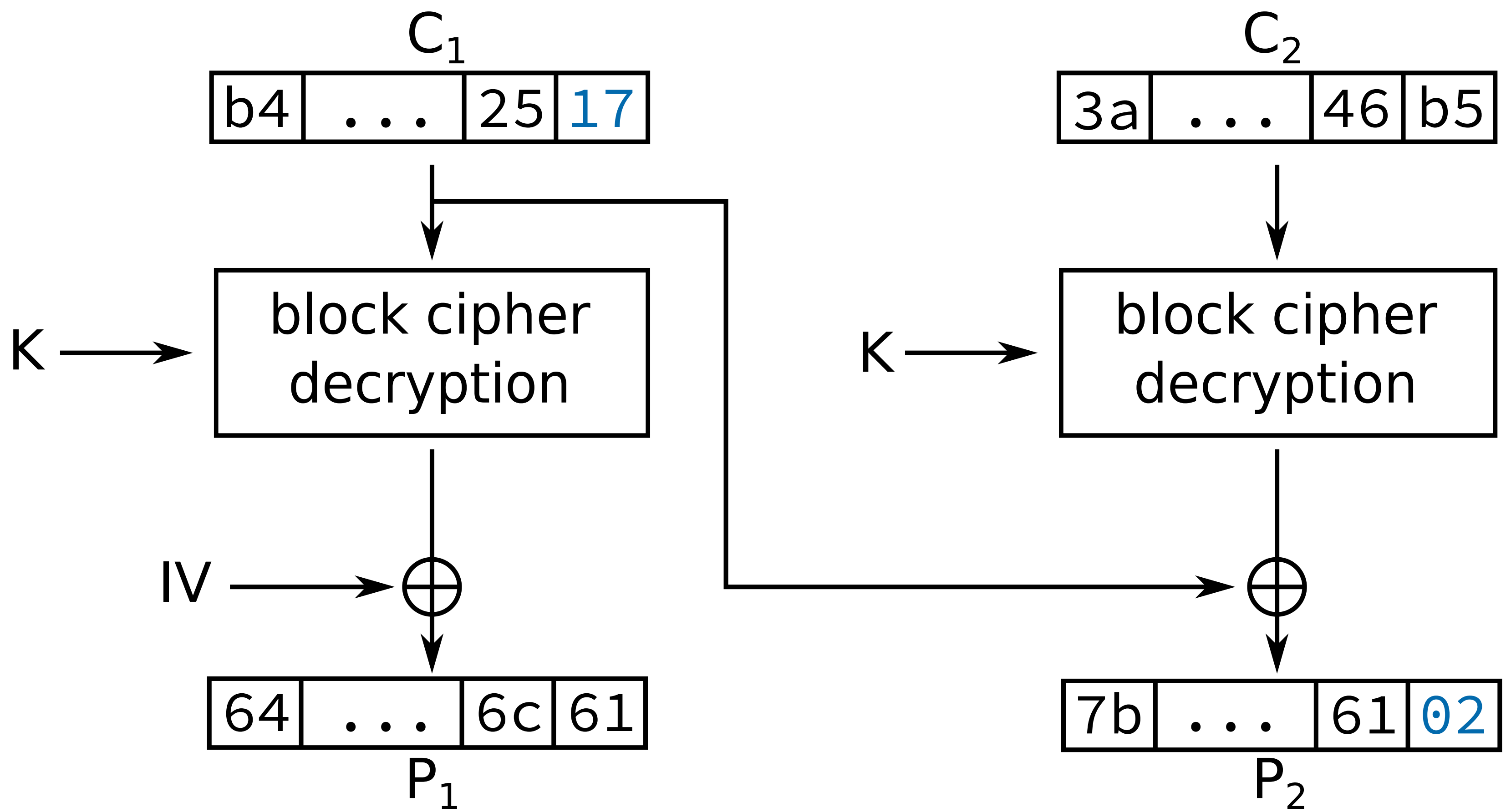
- Now we want the last byte of P2 to be `02`
  - We use the guess G = `01` from the previous step

Guessing the Penultimate Byte of P_2



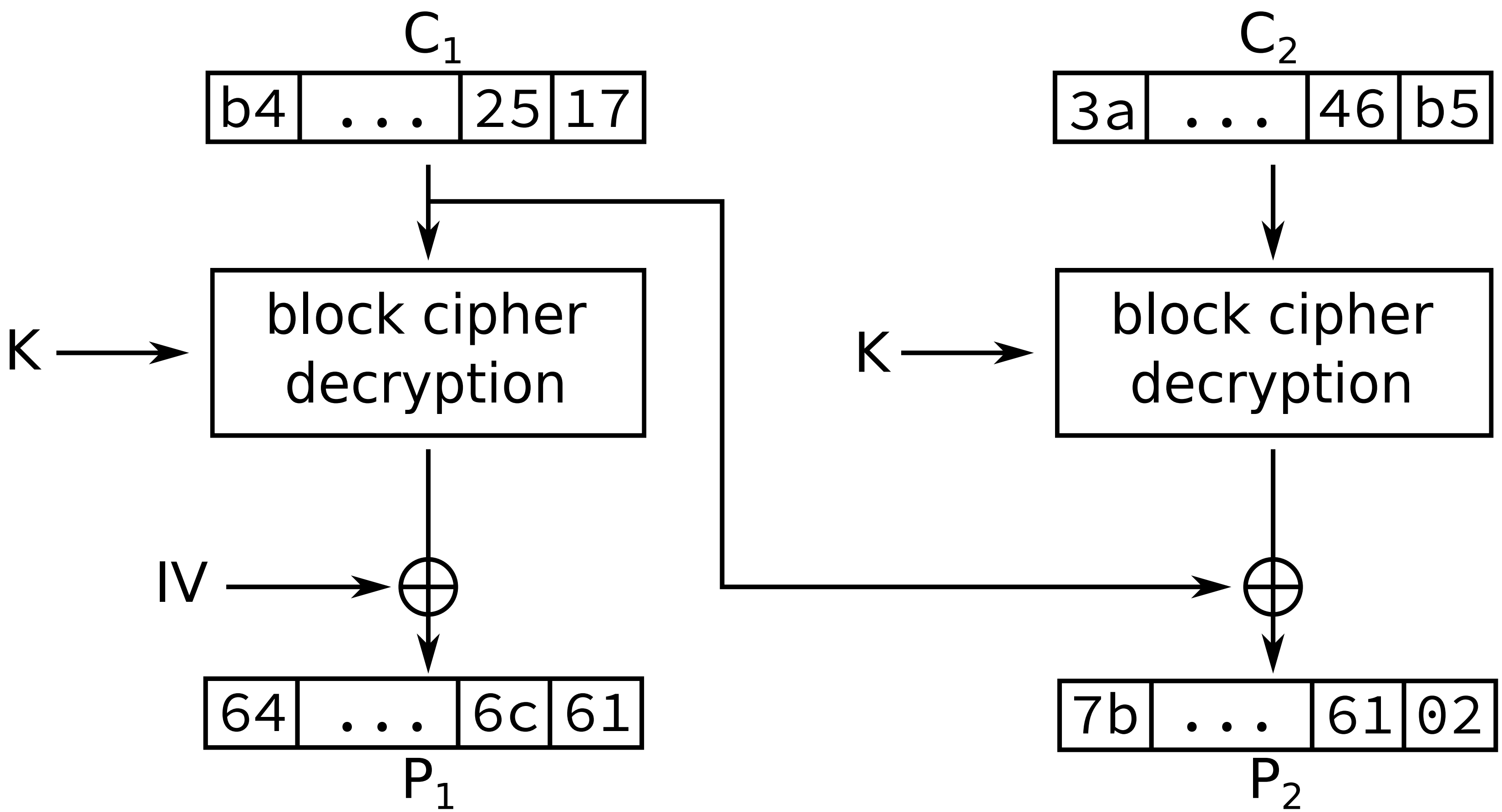
- Now we want the last byte of P2 to be `02`
  - We use the guess G = `01` from the previous step
- Last byte of C_1 becomes `14` $\oplus$ G $\oplus$ `02` = `17`

Guessing the Penultimate Byte of P_2



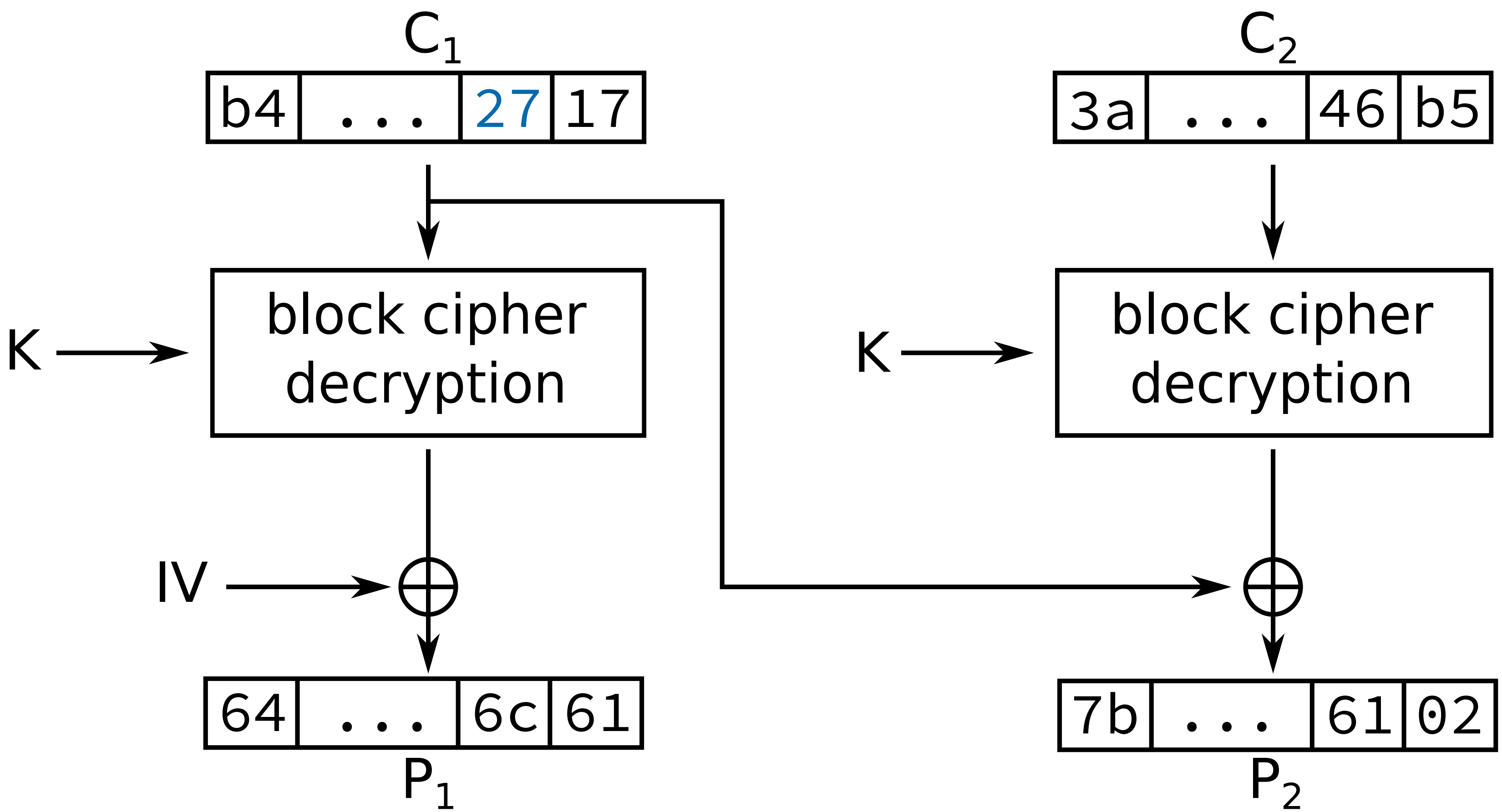
- Now we want the last byte of P2 to be `02`
  - We use the guess G = `01` from the previous step
- Last byte of C_1 becomes $\texttt{14} \oplus G \oplus \texttt{02} = \texttt{17}$

Guessing the Penultimate Byte of P_2



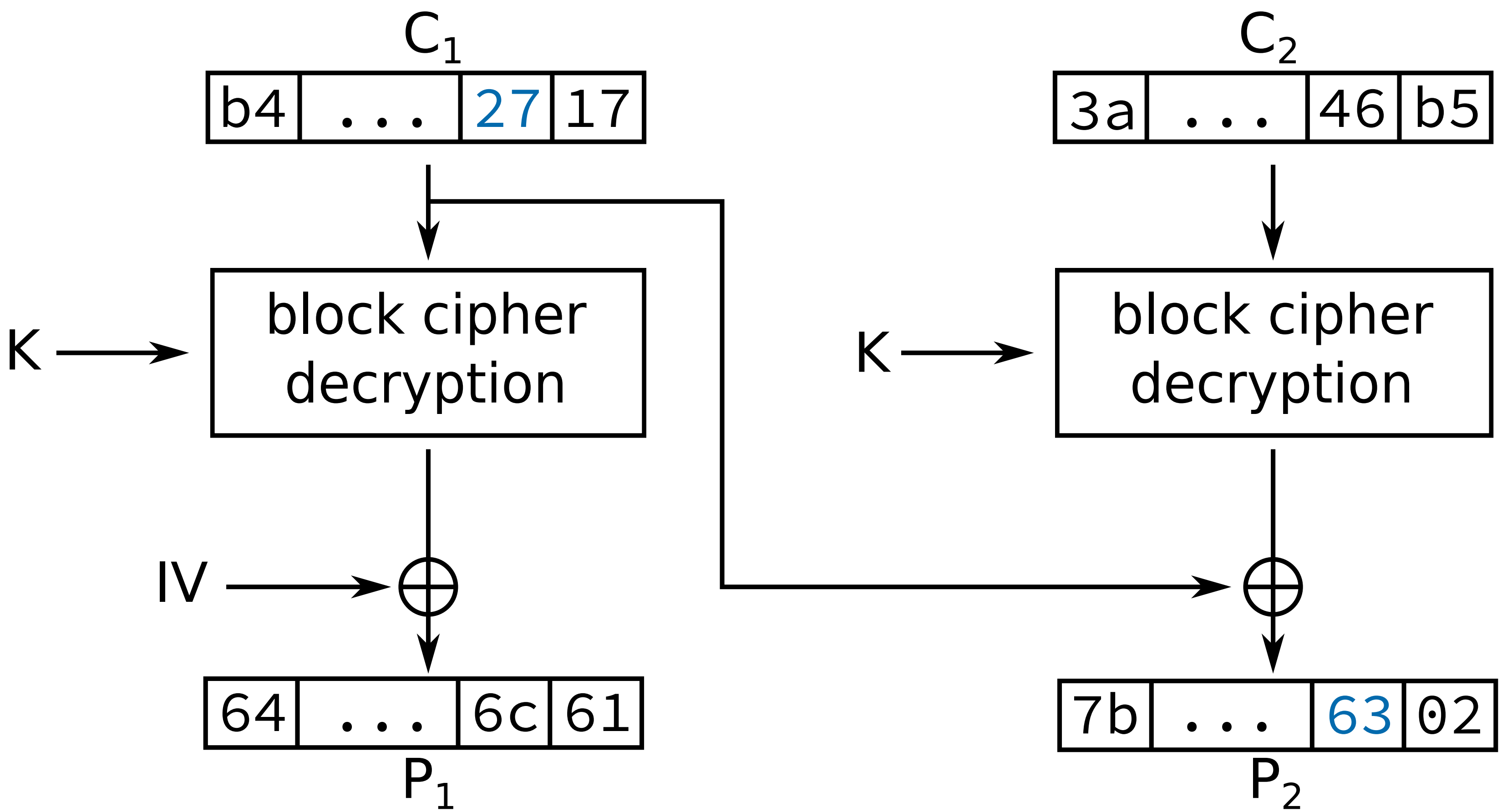
- Our guess is G = `00`

Guessing the Penultimate Byte of P_2



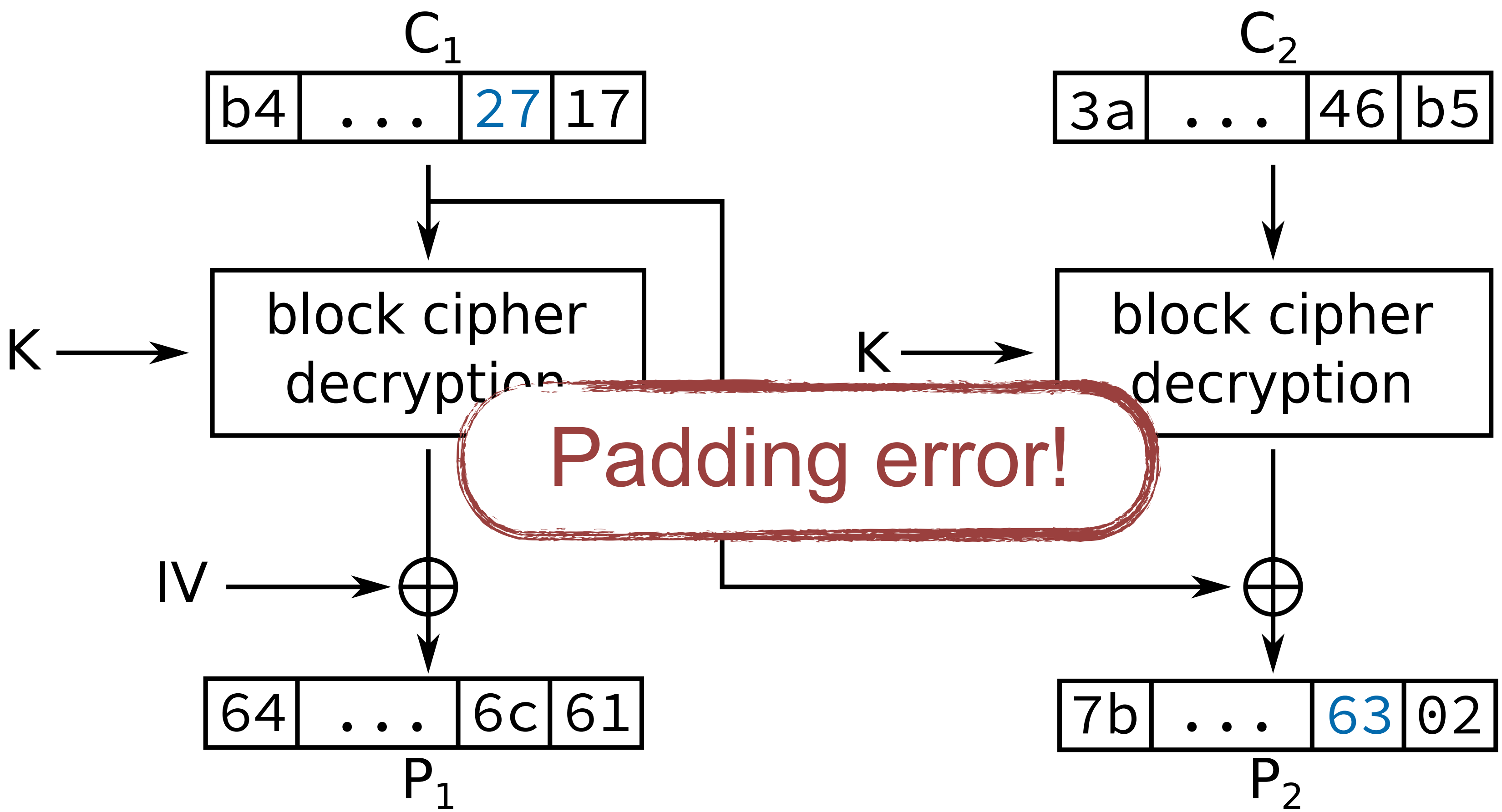
- Our guess is G = `00`
- Penultimate byte of C_1 becomes $25 \oplus 00 \oplus 02 = 27$

Guessing the Penultimate Byte of P_2



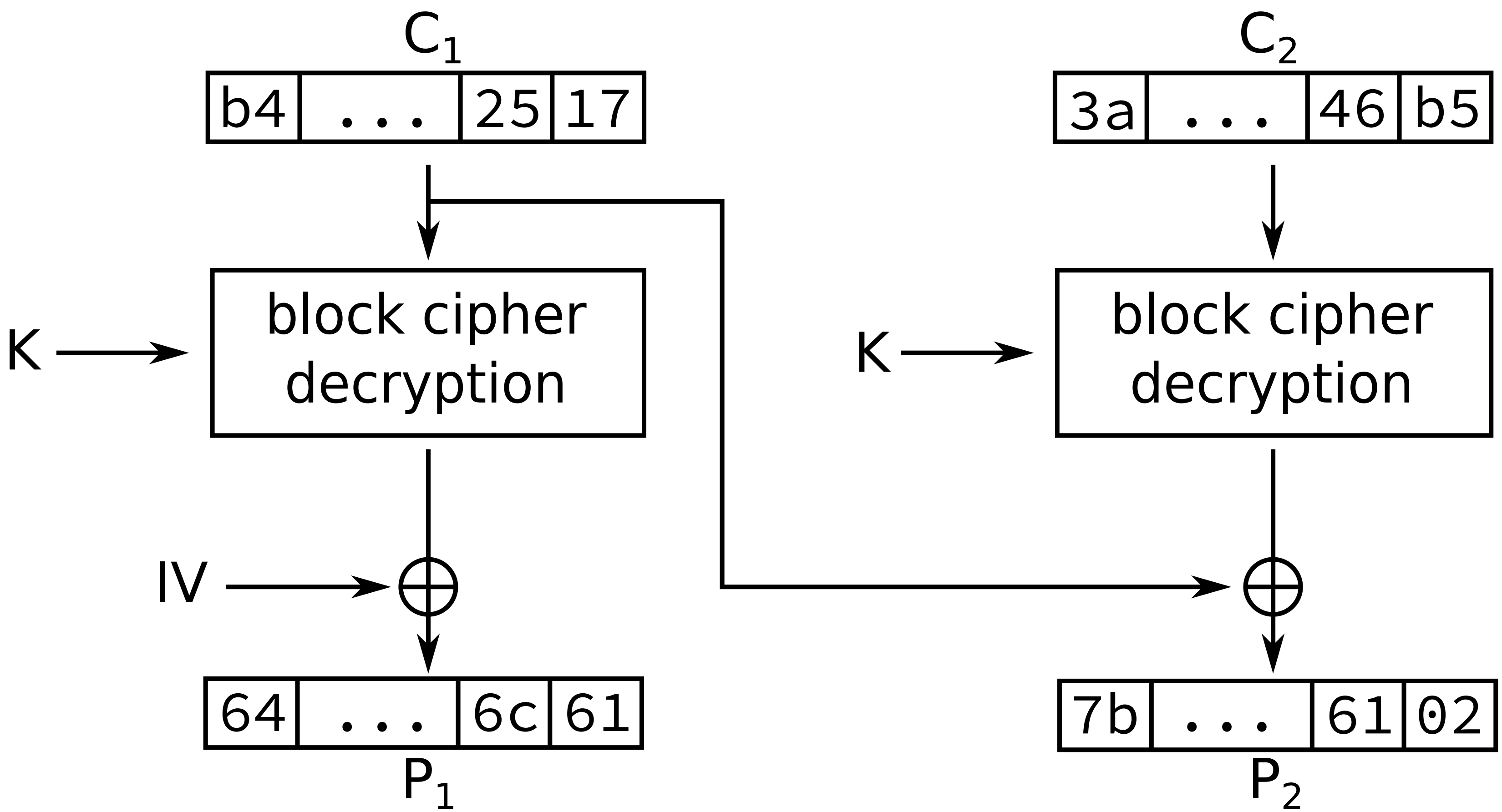
- Our guess is G = `00`
- Penultimate byte of C_1 becomes `25` ⊕ `00` ⊕ `02` = `27`

Guessing the Penultimate Byte of P_2



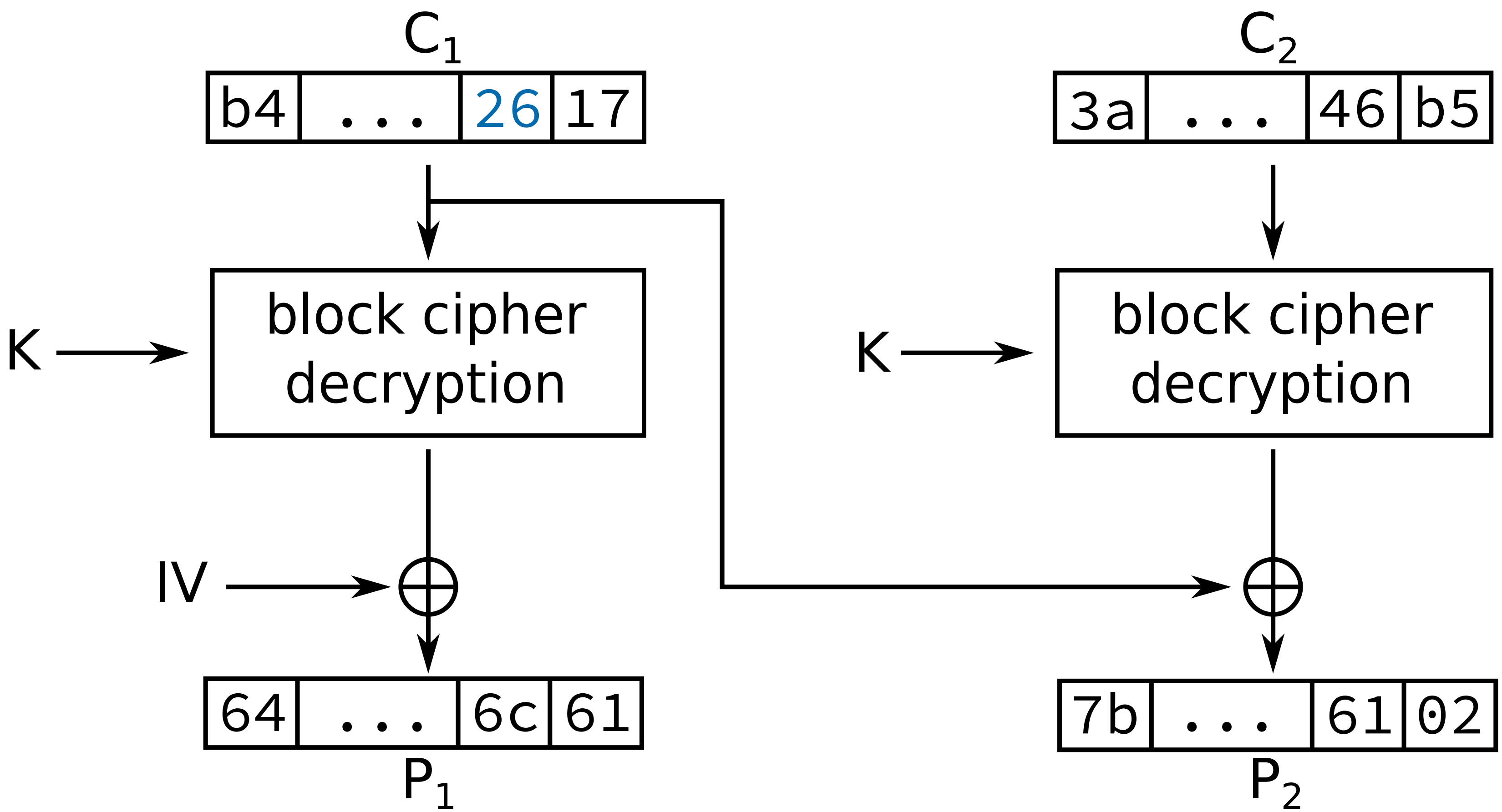
- Our guess is G = `00`
- Penultimate byte of C_1 becomes $25 \oplus 00 \oplus 02 = 27$

Guessing the Penultimate Byte of P_2



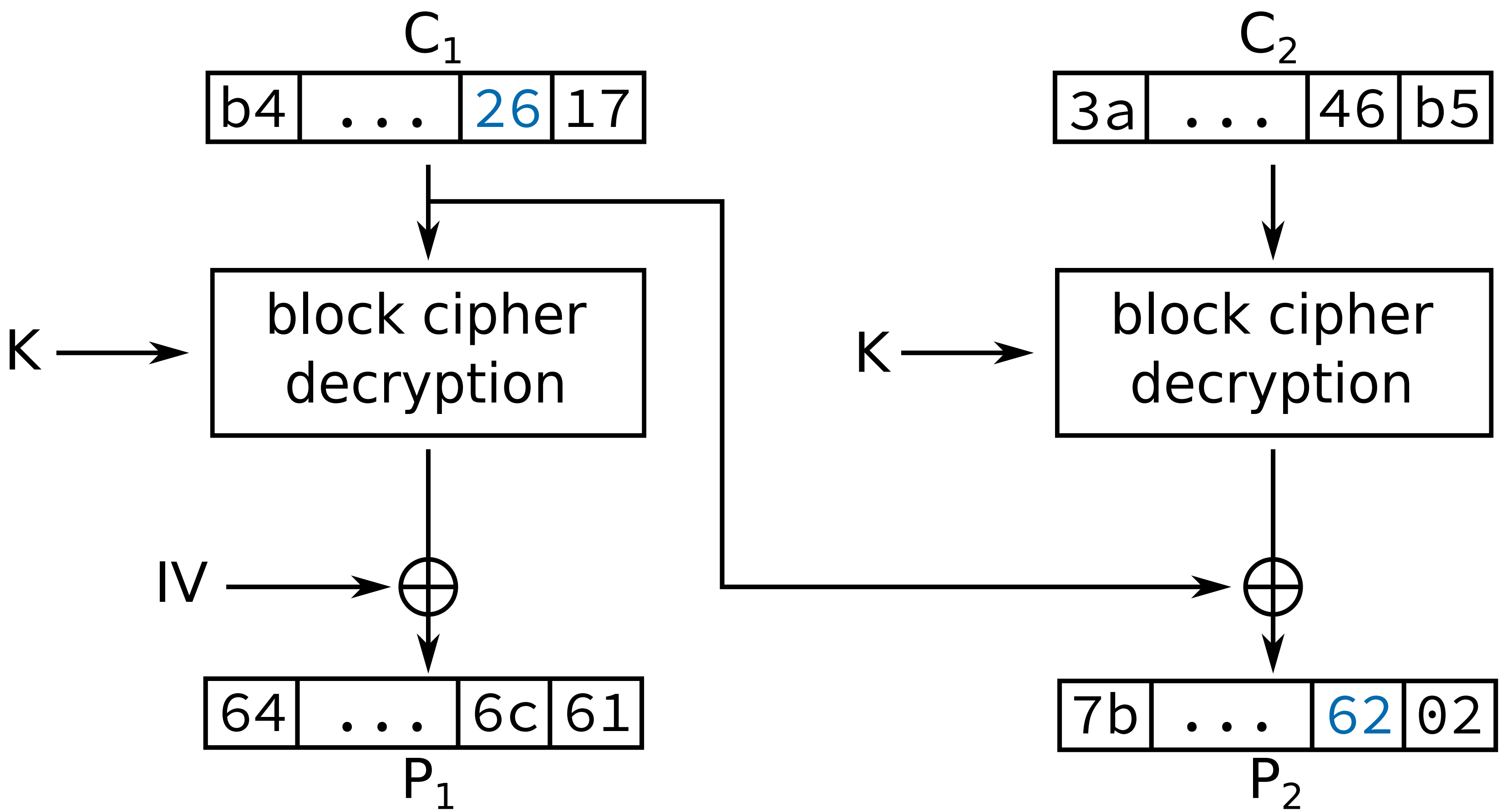
- Our guess is G = `01`

Guessing the Penultimate Byte of P_2



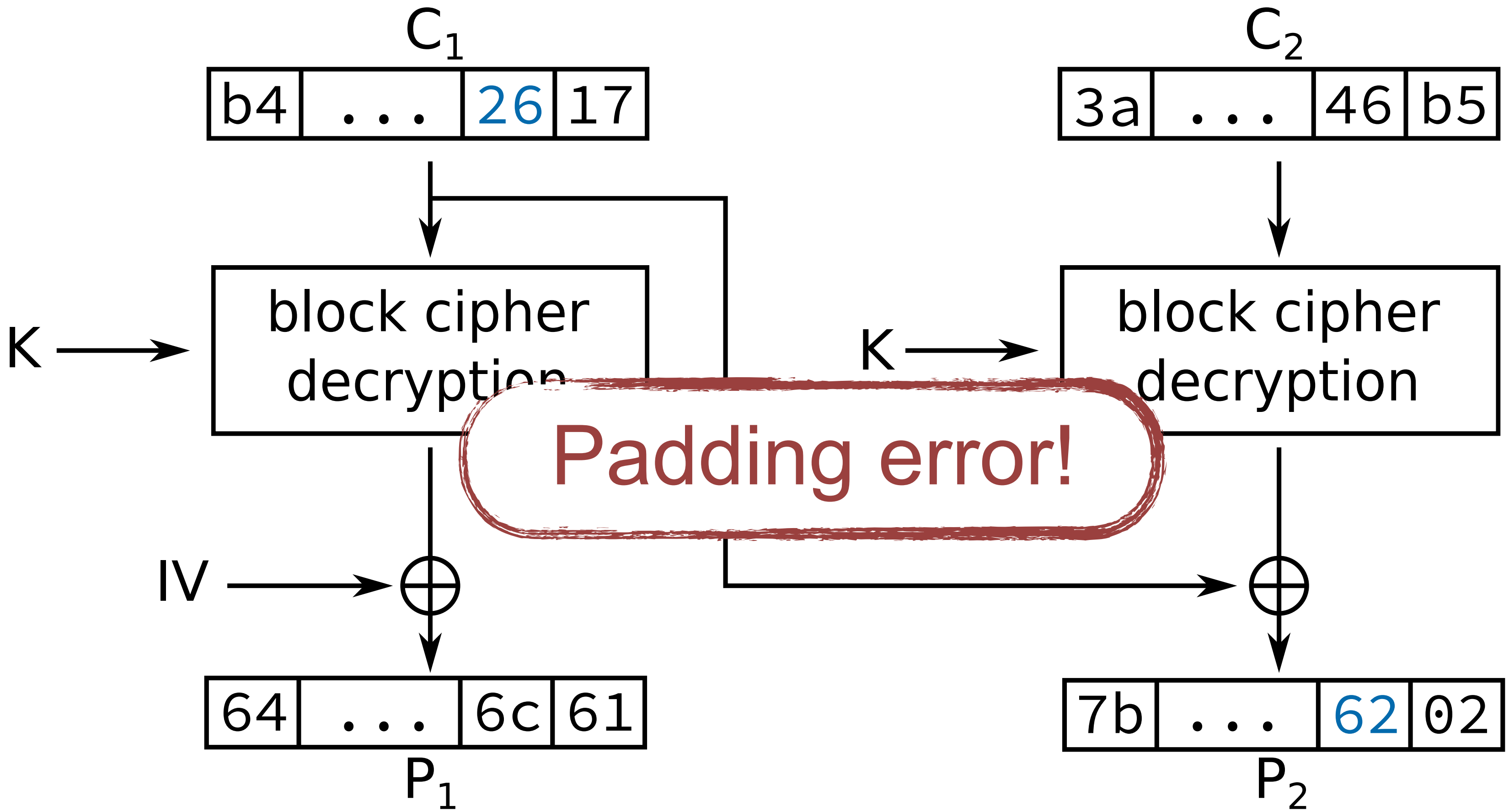
- Our guess is G = `01`
- Penultimate byte of C_1 becomes $25 \oplus 01 \oplus 02 = 26$

Guessing the Penultimate Byte of P_2



- Our guess is G = `01`
- Penultimate byte of C_1 becomes $\texttt{25} \oplus \texttt{01} \oplus \texttt{02} = \texttt{26}$

Guessing the Penultimate Byte of P_2

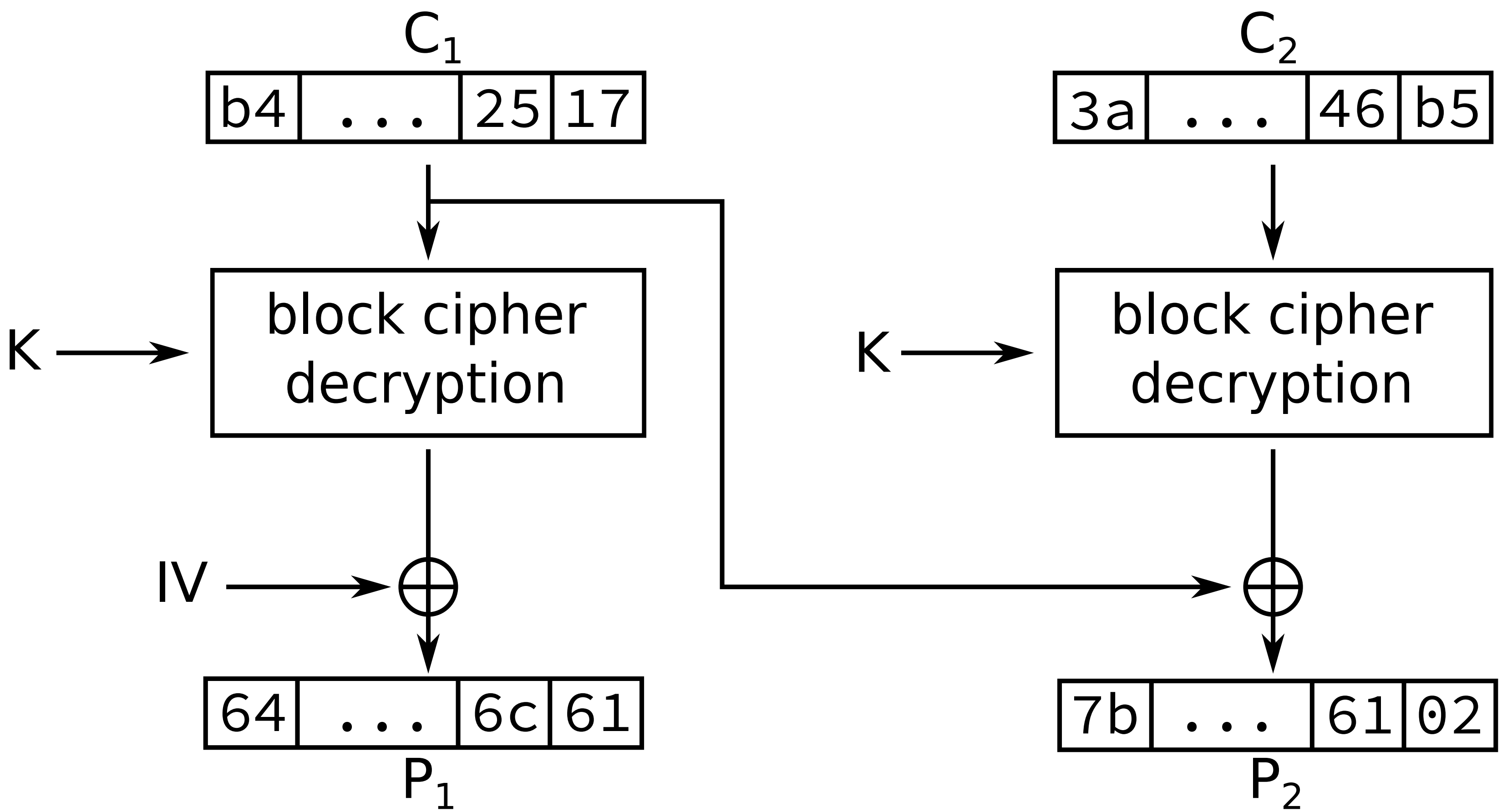


- Our guess is G = `01`
- Penultimate byte of C_1 becomes $25 \oplus 01 \oplus 02 = 26$

Guessing the Penultimate byte of P_2

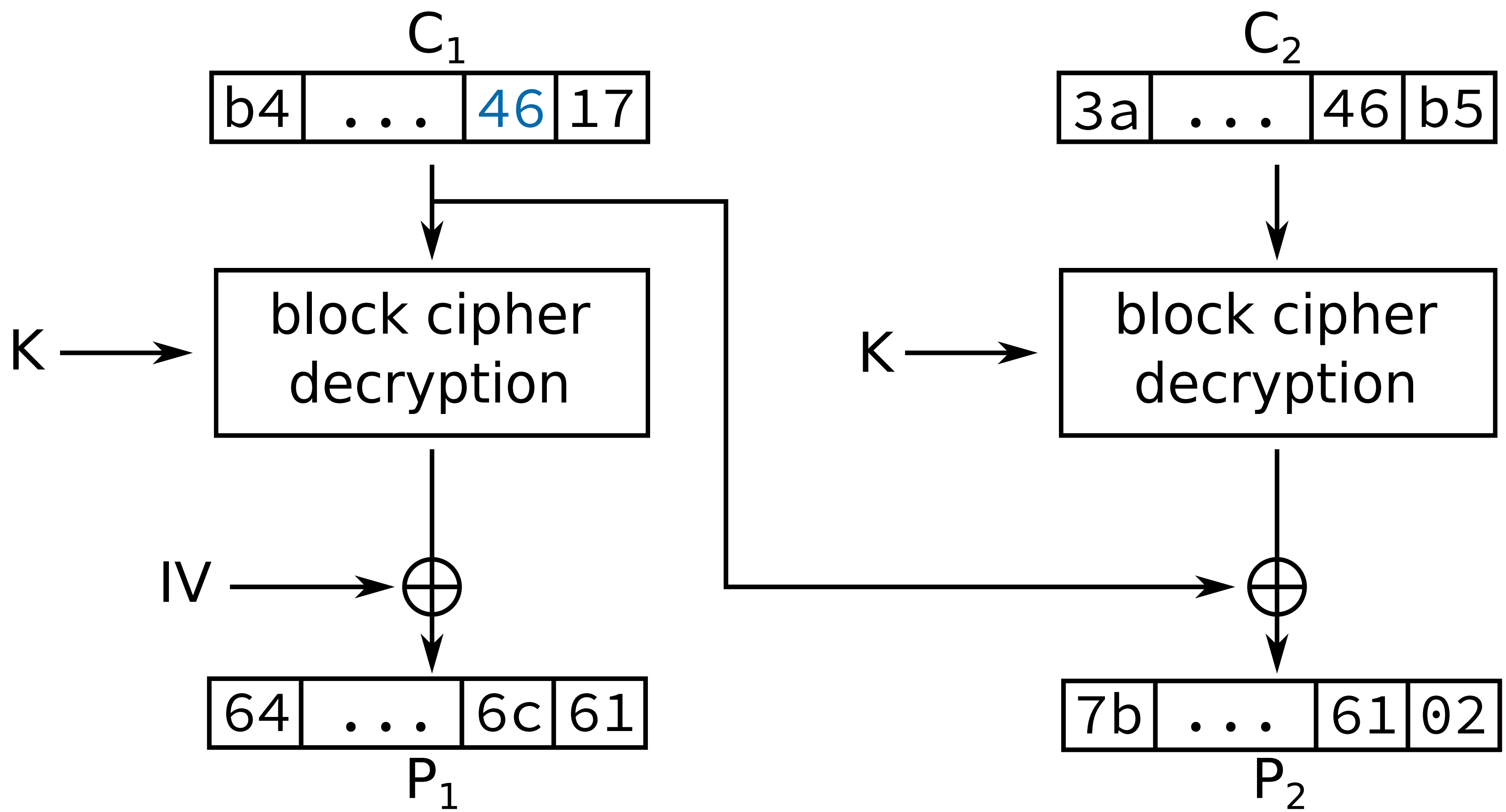
. . .

Guessing the Penultimate Byte of P_2



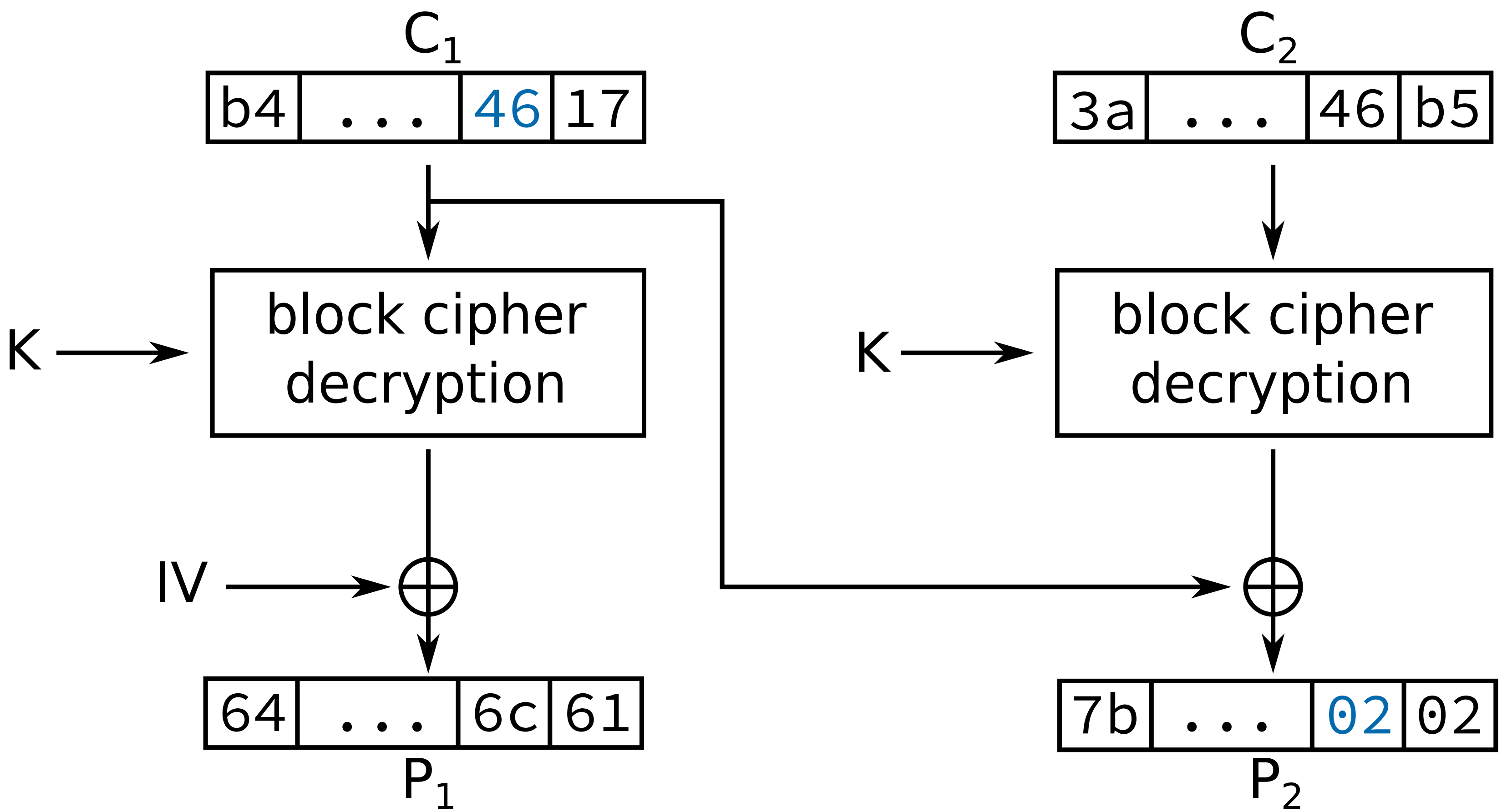
- Our guess is G = 61

Guessing the Penultimate Byte of P_2



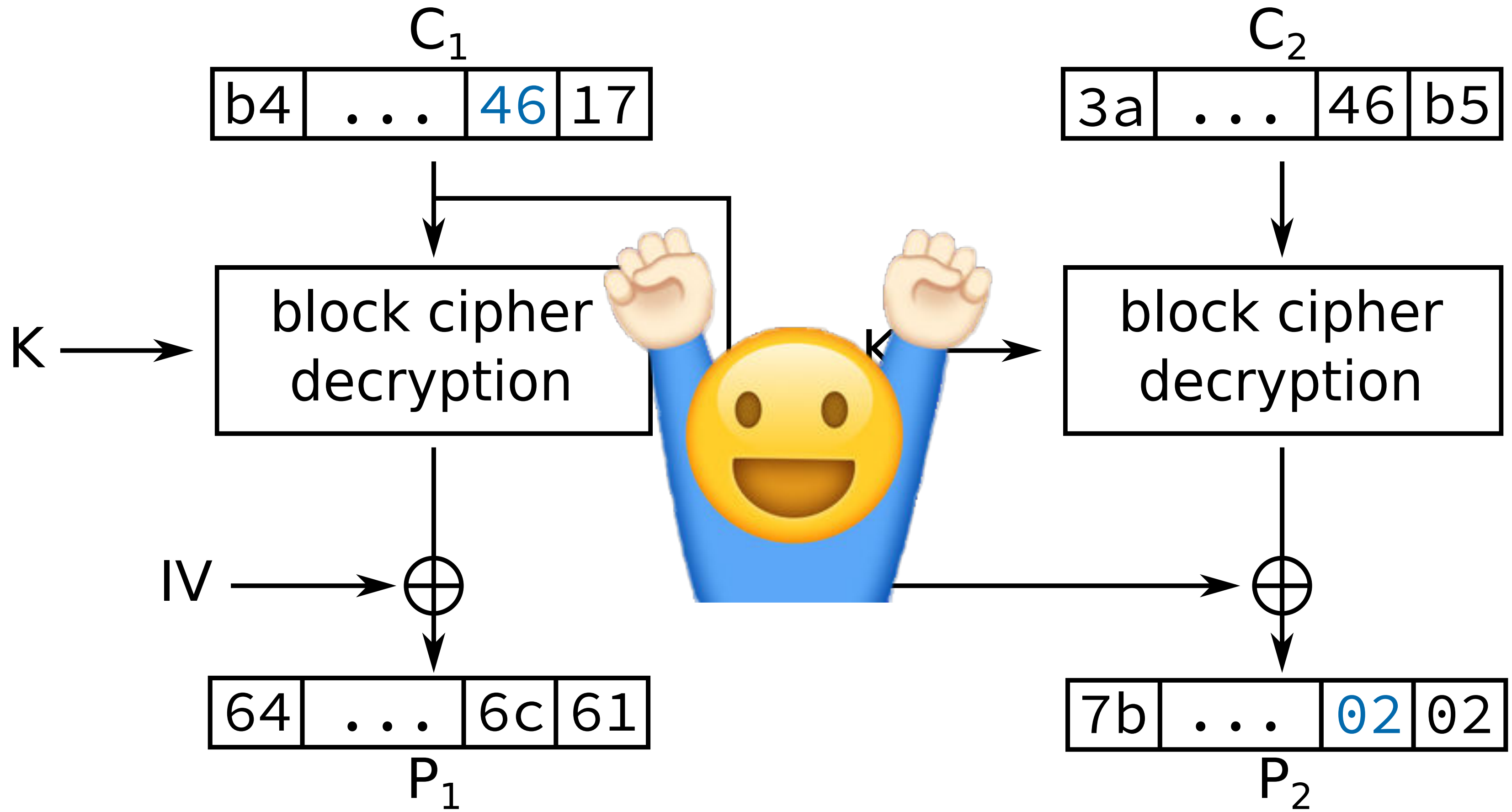
- Our guess is G = `61`
- Penultimate byte of C_1 becomes `25` ⊕ `61` ⊕ `02` = `46`

Guessing the Penultimate Byte of P_2



- Our guess is G = 61
- Penultimate byte of C_1 becomes $25 \oplus 61 \oplus 02 = 46$

Guessing the Penultimate Byte of P_2



- Our guess is G = 61
- Penultimate byte of C_1 becomes $25 \oplus 61 \oplus 02 = 46$

Final Remarks

- Proceed similarly to decrypt the rest of the block!
- To decrypt P_1 you can provide to the oracle the ciphertext $C_0||C_1$ where C_0 is a (random) block of bytes

- Sometimes you may have some false positives when decrypting the last byte of a block
  - Consider a block ending with two `02` bytes
  - Besides guess `02`, also `01` does not raise a padding error: the last byte of the plaintext will remain `02` thus you have a valid padding
- What to do?
  - If you pick a wrong guess, all attempts for the penultimate byte will result in a padding error
  - In that case, change guess for the last byte