Symmetric Cryptography

Kostas Papagiannopoulos
University of Amsterdam
kostaspap88@gmail.com // kpcrypto.net

Contents

Symmetric Encryption

Symmetric Encryption

The 5 core elements of a symmetric cryptosystem:

1. **Plaintext** P: the original message or data that we want to secure. The plaintext P is an input to the encryption algorithm.

Symmetric Encryption

The 5 core elements of a symmetric cryptosystem:

1. **Plaintext** P: the original message or data that we want to secure. The plaintext P is an input to the encryption algorithm.
2. **Secret key** K: a secret value that is also an input to the encryption algorithm. The key K is 'mixed' with the plaintext P.

Symmetric Encryption

The 5 core elements of a symmetric cryptosystem:

1. **Plaintext** P: the original message or data that we want to secure. The plaintext P is an input to the encryption algorithm.
2. **Secret key** K: a secret value that is also an input to the encryption algorithm. The key K is 'mixed' with the plaintext P.
3. **Encryption algorithm** $enc(P, K)$: an algorithm that 'mixes' the plaintext P and the key K. The encryption algorithm will produce a different output depending on the inputs P and K.

Symmetric Encryption

The 5 core elements of a symmetric cryptosystem:

1. **Plaintext** P: the original message or data that we want to secure. The plaintext P is an input to the encryption algorithm.
2. **Secret key** K: a secret value that is also an input to the encryption algorithm. The key K is 'mixed' with the plaintext P.
3. **Encryption algorithm** $enc(P, K)$: an algorithm that 'mixes' the plaintext P and the key K. The encryption algorithm will produce a different output depending on the inputs P and K.
4. **Ciphertext** C: the output of the encryption algorithm. The ciphertext C depends on both P and K and it should be unintelligible to an attacker.

$$C = enc(P, K)$$

"Encrypt plaintext P using key K and algorithm $enc(\cdot)$, outputing ciphertext C"

Symmetric Encryption

The 5 core elements of a symmetric cryptosystem:

1. **Plaintext** P: the original message or data that we want to secure. The plaintext P is an input to the encryption algorithm.
2. **Secret key** K: a secret value that is also an input to the encryption algorithm. The key K is 'mixed' with the plaintext P.
3. **Encryption algorithm** $enc(P, K)$: an algorithm that 'mixes' the plaintext P and the key K. The encryption algorithm will produce a different output depending on the inputs P and K.
4. **Ciphertext** C: the output of the encryption algorithm. The ciphertext C depends on both P and K and it should be unintelligible to an attacker.

$$C = enc(P, K)$$

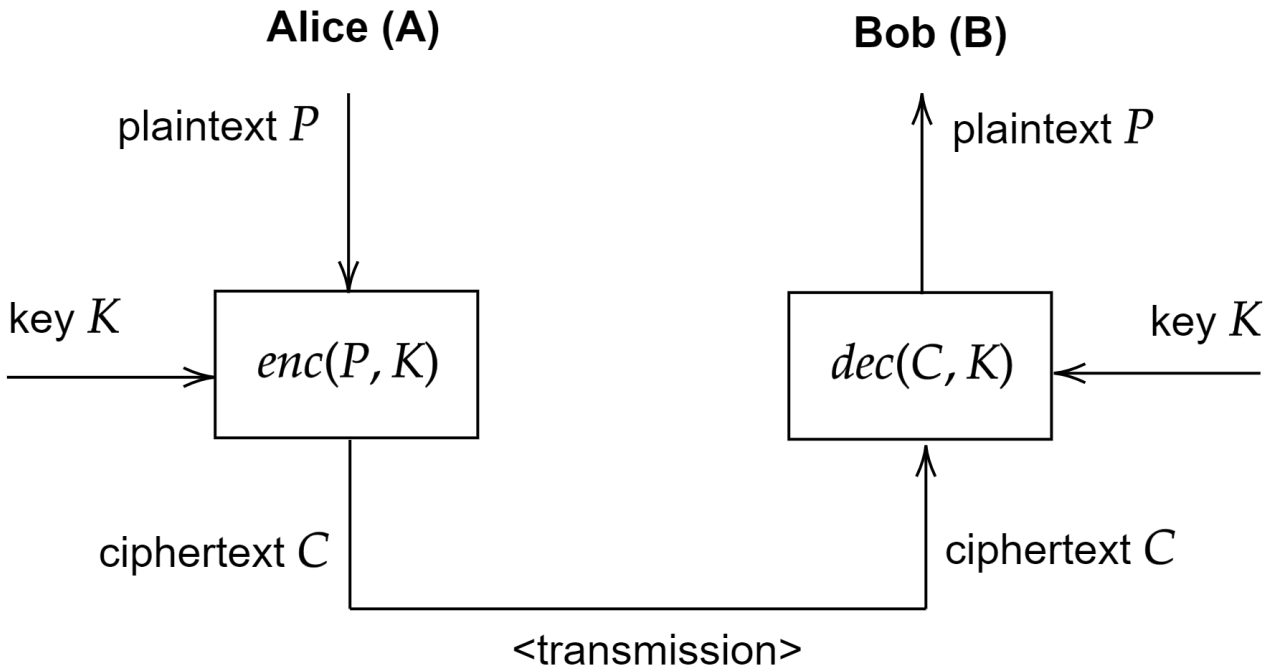
"Encrypt plaintext P using key K and algorithm $enc(\cdot)$, outputing ciphertext C"

5. **Decryption algorithm** $dec(C, K)$: The inverse of the encryption algorithm. The decryption algorithm has the ciphertext C and the key K as inputs and outputs the original plaintext P

$$P = dec(C, K)$$

Symmetric Encryption

- Symmetric encryption between Alice (party A) and Bob (party B)
- Alice and Bob share the same secret key K before $enc(\cdot), dec(\cdot)$ takes place



Symmetric Encryption

Symmetric cryptosystem requirements:

1. The two communicating parties A, B have already shared and stored the secret key K in a secure fashion. The same key K is used in both encryption and decryption, hence the name 'symmetric'.

Symmetric Encryption

Symmetric cryptosystem requirements:

1. The two communicating parties A, B have already shared and stored the secret key K in a secure fashion. The same key K is used in both encryption and decryption, hence the name 'symmetric'.
2. The cryptosystem is correct:

$$dec(enc(P, K), K) = P$$

Symmetric Encryption

Symmetric cryptosystem requirements:

1. The two communicating parties A, B have already shared and stored the secret key K in a secure fashion. The same key K is used in both encryption and decryption, hence the name 'symmetric'.
2. The cryptosystem is correct:

$$dec(enc(P, K), K) = P$$

3. The encryption algorithm is 'strong' i.e. an attacker should not be able to recover sensitive information like the plaintext P and/or the key K by observing the ciphertext C.

   Rephrasing, the *only* way to decrypt C is by knowing the key K.

Symmetric Encryption

Kerckhoffs's principle:

- Established by Dutch cryptographer and linguist Auguste Kerckhoffs (1835 -1903)
- The inner workings and specification of the encryption and decryption algorithms must be public information
- The security of the cryptosystem should rest on the secrecy of the key *K only*

Symmetric Encryption

Kerckhoffs's principle:

- Established by Dutch cryptographer and linguist Auguste Kerckhoffs (1835 -1903)
- The inner workings and specification of the encryption and decryption algorithms must be public information
- The security of the cryptosystem should rest on the secrecy of the key *K only*

模糊

Here is why we should avoid 'security by obscurity' at all times:

- Algorithms can leak or be reversed-engineered
  e.g. the proprietary Crypto1 cipher used by MIFARE classic chips in the OV-chipkaart was reversed, analyzed and finally cracked

- What makes a good encryption algorithm is several years of public scrutiny
  e.g. the Advanced Encryption Standard uses the Rijndael algorithm which remains secure after 20+ years of cryptanalysis

- The Kerckhoffs's principle extends to all security domains: more "eye balls" are more likely to expose flaws in network protocols, server configurations, privacy-enhancing techniques etc. thus we must follow the open design principle

One -Time Pad (OTP)

OTP

One-Time Pad (OTP):

- OTP is a symmetric cryptosystem developed by G. Vernam (1918) and later by J. Mauborgne
- It features a simple yet very secure encryption algorithm

OTP

One-Time Pad (OTP):

- OTP is a symmetric cryptosystem developed by G. Vernam (1918) and later by J. Mauborgne
- It features a simple yet very secure encryption algorithm

OTP encryption/decryption algorithm:

1. The OTP represents the plaintext P in bits

   e.g. the plaintext $p =$ `HELLO` is mapped using the ASCII table to:

   $p =$ `0x48, 0x45, 0x4c, 0x4c, 0x4f`

   and then is represented in binary:

   $p =$ `01001000 01000101 01001100 01001100 01001111` (40 bits)

OTP

2. The OTP generates a uniformly random key K that has equal bit-length with the plaintext P. The key K is shared between Alice (A) and Bob (B).

   e.g. $k =$11000011 10011110 01100001 01111010 11100101 (40 bits)

OTP

2. The OTP generates a uniformly random key K that has equal bit-length with the plaintext P. The key K is shared between Alice (A) and Bob (B).

   e.g. $k =$11000011 10011110 01100001 01111010 11100101 (40 bits)

3. The ciphertext C is generated by performing the **bitwise XOR** between the plaintext P and the key K

$$C(i) = enc(P(i), K(i)) = P(i) \oplus K(i) = (P(i) + K(i)) \bmod 2$$

	01001000	01000101	01001100	01001100	01001111
e.g. $\oplus$	11000011	10011110	01100001	01111010	11100101
	10001011	11011011	00101101	00110110	10101010

OTP

4. The decryption algorithm produces the plaintext P by performing the **bitwise XOR** between the ciphertext C and the key K

$$P(i) = dec(C(i), K(i)) = C(i) \oplus K(i) = (C(i) + K(i)) \bmod 2$$

		10001011	11011011	0010110	00110110	10101010
e.g.	$\oplus$	11000011	10011110	01100001	01111010	11100101
		01001000	01000101	01001100	01001100	01001111

Converting back to ASCII characters, we receive again $p =$ `HELLO`

OTP

4. The decryption algorithm produces the plaintext P by performing the bitwise XOR between the ciphertext C and the key K

$$P(i) = dec(C(i), K(i)) = C(i) \oplus K(i) = (C(i) + K(i)) \bmod 2$$

		10001011	11011011	0010110	00110110	10101010
e.g.	$\oplus$	11000011	10011110	01100001	01111010	11100101
		01001000	01000101	01001100	01001100	01001111

Converting back to ASCII characters, we receive again $p =$ `HELLO`

OTP correctness:

$$dec(enc(P, K), K) = enc(P, K) \oplus K = (P \oplus K) \oplus K = P$$

OTP

Consider the following scenario (aka threat model):

- The attacker knows the distribution of the plaintext P
- The attacker knows the encryption/decryption algorithms $enc(\cdot)$ and $dec(\cdot)$
- The attacker does not know the plaintext P or the key K but can eavesdrop the ciphertext C 窃听

In a **perfectly secret cryptosystem**, observing the ciphertext C should not reveal any information regarding the plaintext P

OTP

Consider the following scenario (aka threat model):

- The attacker knows the distribution of the plaintext P
- The attacker knows the encryption/decryption algorithms $enc(\cdot)$ and $dec(\cdot)$
- The attacker does not know the plaintext P or the key K but can eavesdrop the ciphertext C

In a **perfectly secret cryptosystem**, observing the ciphertext C should not reveal any information regarding the plaintext P

OTP Perfect Secrecy:

Def. Let $\mathcal{P}$ the set of all possible plaintexts, $\mathcal{C}$ the set of all possible ciphertexts and $\mathcal{K}$ the set of all possible keys.
An encryption is **perfectly secret** if for every plaintext $p \in \mathcal{P}$ and every ciphertext $c \in \mathcal{C}$ it holds that:

$$Pr(P = p|C = c) = Pr(P = p)$$

knowing c wont increase the prob of gessing p

OTP

Probability theory reminder:

- Conditional probability

$$Pr(X = x|Y = y) = \frac{Pr(X = x, Y = y)}{Pr(Y = y)}$$

- Marginal probability

  边缘概率

$$Pr(X = x) = \sum_{y} Pr(X = x, Y = y)$$

OTP

Probability theory reminder:

- Bayes' rule

$$Pr(X = x|Y = y) = \frac{Pr(X = x, Y = y)}{Pr(Y = y)} = \frac{Pr(Y = y|X = x) \cdot Pr(X = x)}{Pr(Y = y)} =$$

$$\frac{Pr(Y = y|X = x) \cdot Pr(X = x)}{\sum_{x^*} Pr(Y = y, X = x^*)} = \frac{Pr(Y = y|X = x) \cdot Pr(X = x)}{\sum_{x^*} Pr(Y = y|X = x^*) \cdot Pr(X = x^*)}$$

For uniformly distributed X: $Pr(X = x|Y = y) = \frac{Pr(Y = y|X = x)}{\sum_{x^*} Pr(Y = y|X = x^*)}$

prob of x=x1/x2/... same

OTP

Proof(1). We first show that $Pr(C = c|P = p) = 1/|\mathcal{K}|$

By the definition of encryption, of the conditional probability and of the OTP:

$$Pr(C = c|P = p) = Pr(enc(P, K) = c|P = p) = Pr(enc(p, K) = c) =$$

$$Pr(p \oplus K = c) = Pr(K = c \oplus p)$$

In the OTP, the key K is generated in a uniformly random manner thus $Pr(K = k) = 1/|\mathcal{K}|$. Hence:

$$Pr(K = c \oplus p) = Pr(K = k) = \frac{1}{|\mathcal{K}|}$$

OTP

Proof(2). We now show that $Pr(C = c) = 1/|\mathcal{K}|$

Using the definition of marginal probability, the Bayes rule and the previous result:

$$Pr(C = c) = \sum_{p \in \mathcal{P}} Pr(C = c, P = p) = \sum_{p \in \mathcal{P}} Pr(C = c | P = p) \cdot Pr(P = p) =$$

$$\sum_{p \in \mathcal{P}} \frac{1}{|\mathcal{K}|} \cdot Pr(P = p) = \frac{1}{|\mathcal{K}|} \cdot \sum_{p \in \mathcal{P}} Pr(P = p) = \frac{1}{|\mathcal{K}|}$$

OTP

Proof(3). We finally show that $Pr(P = p|C = c) = Pr(P = p)$ [**perfect secrecy**]

Using the Bayes theorem and the previous results:

$$Pr(P = p|C = c) = \frac{Pr(P = p, C = c)}{Pr(C = c)} = \frac{Pr(C = c|P = p) \cdot Pr(P = p)}{Pr(C = c)} =$$

$$\frac{(1/|\mathcal{K}|) \cdot Pr(P = p)}{(1/|\mathcal{K}|)} = Pr(P = p)$$

OTP

What does **perfect secrecy** imply in practice?

- It means that, given the ciphertext C, any plaintext can be generated by a suitable choice of key K
- All these possible generated plaintexts are equally likely

OTP

What does perfect secrecy imply in practice?

- It means that, given the ciphertext C, any plaintext can be generated by a suitable choice of key K
- All these possible generated plaintexts are equally likely

- For example, observe the ciphertext
  $c = \texttt{10001011 11011011 00101101 00110110 10101010}$
- This ciphertext originates from plaintext $p_1 = \texttt{HELLO}$ under the chosen key
  $k_1 = \texttt{11000011 10011110 01100001 01111010 11100101}$

OTP

What does perfect secrecy imply in practice?

- It means that, given the ciphertext C, any plaintext can be generated by a suitable choice of key K
- All these possible generated plaintexts are equally likely

- For example, observe the ciphertext
  $c = \texttt{10001011 11011011 00101101 00110110 10101010}$
- This ciphertext originates from plaintext $p_1 = \texttt{HELLO}$ under the chosen key
  $k_1 = \texttt{11000011 10011110 01100001 01111010 11100101}$
- Choosing a different key $k_2 = \texttt{11000010 10011111 1100100 1111001 11111110}$ will decrypt c into $p_2 = \texttt{IDIOT}$ (after ASCII mapping)
- As long as the correct key is unknown, the plaintexts `HELLO` and `IDIOT` are equally likely

OTP

What does perfect secrecy imply in practice?

- It means that, given the ciphertext C, any plaintext can be generated by a suitable choice of key K
- All these possible generated plaintexts are equally likely

- For example, observe the ciphertext
  $c =$ `10001011 11011011 00101101 00110110 10101010`
- This ciphertext originates from plaintext $p_1 =$ `HELLO` under the chosen key
  $k_1 =$ `11000011 10011110 01100001 01111010 11100101`
- Choosing a different key $k_2 =$ `11000010 10011111 1100100 1111001 11111110` will decrypt c into $p_2 =$ `IDIOT` (after ASCII mapping)
- As long as the correct key is unknown, the plaintexts `HELLO` and `IDIOT` are equally likely

Confidentiality. Using the OTP algorithm establishes the security property named **confidentiality** i.e. only the sender and intended receiver (A, B) should be able to understand the contents of the transmitted message.

OTP

Theorem. Let a perfectly secret cryptosystem with plaintext space $\mathcal{P}$ and key space $\mathcal{K}$. Then it must hold that:

$$|\mathcal{K}| \geq |\mathcal{P}|$$

OTP

Theorem. Let a perfectly secret cryptosystem with plaintext space $\mathcal{P}$ and key space $\mathcal{K}$. Then it must hold that:

$$|\mathcal{K}| \geq |\mathcal{P}|$$

Proof. We show that if $|\mathcal{K}| < |\mathcal{P}|$ then the cryptosystem cannot be perfectly secret.

Let uniformly distributed plaintext P in space $\mathcal{P}$

Let a certain ciphertext c that occurs with non-zero probability

Let set $\mathcal{S}$ that contains all possible decryptions of c i.e.

$$\mathcal{S} = \{\text{all } p \text{ such that } c = enc(p, k), \text{ for some } k \in \mathcal{K}\}$$

It holds that $|\mathcal{S}| \leq |\mathcal{K}|$ since decrypting c for all k can result in at most $|\mathcal{K}|$ plaintexts

However we assumed that $|\mathcal{K}| < |\mathcal{P}|$ thus there exists some $p' \in \mathcal{P}$ such that $p' \notin \mathcal{S}$

Then we can see that:

$$Pr(P = p'|C = c) = 0 \neq Pr(P = p')$$

Thus the cryptosystem is not perfectly secret

OTP

What does the theorem imply in practice?

- To maintain perfect secrecy the number of possible keys is at least as big as the number of possible plaintexts
- The bit-length of the key K must be greater or equal to the bit-length of the plaintext P
- The key should not be re-used if we want to maintain perfect secrecy

OTP Attacks

OTP Attacks

Two-Pad attack:

Consider the following scenario:

- Parties A and B are communicating using the OTP
- Due to an error, the OTP algorithm re-uses part of the key K

OTP Attacks

Two-Pad attack:

Consider the following scenario:

- Parties A and B are communicating using the OTP
- Due to an error, the OTP algorithm re-uses part of the key K

Attack: The attacker obtains ciphertexts C_1, C_2 that have been encrypted with the same key part K and will try to infer the respective plaintexts P_1, P_2.

$$C_1 \oplus C_2 = P_1 \oplus K \oplus P_2 \oplus K = P_1 \oplus P_2$$

K bitwise K=0

OTP Attacks

Two-Pad attack:

Consider the following scenario:

- Parties A and B are communicating using the OTP
- Due to an error, the OTP algorithm re-uses part of the key K

Attack: The attacker obtains ciphertexts C_1, C_2 that have been encrypted with the same key part K and will try to infer the respective plaintexts P_1, P_2.

$$C_1 \oplus C_2 = P_1 \oplus K \oplus P_2 \oplus K = P_1 \oplus P_2$$

- The attacker has obtained $P_1 \oplus P_2$ and if P_1 is known, then he can directly recover P_2
- The attacker can detect if the same plaintext is transmitted i.e. if $P_1 = P_2$ then $C_1 = C_2$ as well
- Observing multiple XORed plaintext pairs $P_1 \oplus P_2$ can eventually recover the plaintext in a natural language
- VENONA project (1940): Soviet spies used OTP to communicate. They never re-used the key but there was a flaw in the key generation process causing key repetitions and eventually a two-pad attack.

OTP Attacks

OTP integrity attack:

Consider the following scenario:

- Bank A transfers a salary to bank B
- The communication between A and B is encrypted using the OTP
- The salary amounts to 2700€ so the plaintext looks like:

  $p = 0000101010001100$ (16 bits)
- Encrypting with $k = 1010110000010000$ results in ciphertext $c = p \oplus k = 1010011010011100$ and nothing can be inferred about the original salary amount

OTP Attacks

OTP integrity attack:

Consider the following scenario:

- Bank A transfers a salary to bank B
- The communication between A and B is encrypted using the OTP
- The salary amounts to 2700€ so the plaintext looks like:

  $p = 0000101010001100$ (16 bits)
- Encrypting with $k = 1010110000010000$ results in ciphertext $c = p \oplus k = 1010011010011100$ and nothing can be inferred about the original salary amount

Attack: The attacker cannot recover the salary p but can intercept and modify the encrypted salary. The attacker can flip e.g. the 2nd bit of c producing c'. Subsequently, this 'man-in-the-middle' transmits c' to bank B instead of c.

拦截发送更改过的信息

$$c' = c \oplus 0100000000000000 = 1\underline{1}10011010011100$$

OTP Attacks

- Now bank B will decrypt the modified ciphertext c', resulting in a modified p'

$$p' = c' \oplus k = 1110011010011100 \oplus 1010110000010000 = 19084 \neq 2700$$

- Although the attacker does not know P, he managed to alter the transaction
- Although the OTP provided **confidentiality**, it did not provide **integrity**

OTP Attacks

- Now bank B will decrypt the modified ciphertext c', resulting in a modified p'

$$p' = c' \oplus k = 1110011010011100 \oplus 1010110000010000 = 19084 \neq 2700$$

- Although the attacker does not know P, he managed to alter the transaction
- Although the OTP provided **confidentiality**, it did not provide **integrity**

Integrity. The security property which ensures that the contents of communication are not altered, either maliciously or by accident, in transit from A to B.

OTP Attacks

OTP strengths:

- OTP has perfect secrecy i.e. if correctly implemented, it cannot be broken by observing the ciphertext, regardless of the attacker's computational capabilities (proof vs. alien invasions)
- Very simple and fast encryption and decryption process (just a bitwise XOR operation)

OTP Attacks

OTP strengths:

- OTP has perfect secrecy i.e. if correctly implemented, it cannot be broken by observing the ciphertext, regardless of the attacker's computational capabilities (proof vs. alien invasions)
- Very simple and fast encryption and decryption process (just a bitwise XOR operation)

OTP weaknesses:

- OTP needs very large keys since $|\mathcal{K}| \geq |\mathcal{P}|$. This is very inefficient during data-intensive communications.
- The lengthy keys must be generated by a random number generator that is suitable for cryptographic use. Such a procedure can be slow and error-prone.
- We need to distribute the large keys among parties in advance and make sure that they are well-protected and do not get compromised.
- If an attacker knows both P and C they can recover the key K since $K = P \oplus C$. Certain applications need to resist such attacks on the key.

Thus OTP is preferred for low-bandwidth communication with very high security requirements.

Stream Ciphers

Stream Ciphers

- The very long OTP keys makes the scheme unusable
- Stream ciphers provide a good alternative

Stream Ciphers

- The very long OTP keys makes the scheme unusable
- Stream ciphers provide a good alternative

Stream cipher encryption/decryption algorithm:

1. Alice (A) and Bob (B) share a reasonably-sized key K (128 bits or more).

Stream Ciphers

- The very long OTP keys makes the scheme unusable
- Stream ciphers provide a good alternative

Stream cipher encryption/decryption algorithm:

1. Alice (A) and Bob (B) share a reasonably-sized key K (128 bits or more).
2. Alice inputs the key K to the stream cipher. The stream cipher 'stretches' the key K into a long keystream S. The keystream S is as long as the bit-length of the plaintext P.

$$S = StreamCipher(K), \quad length(S) = length(P)$$

Stream Ciphers

- The very long OTP keys makes the scheme unusable
- Stream ciphers provide a good alternative

Stream cipher encryption/decryption algorithm:

1. Alice (A) and Bob (B) share a reasonably-sized key K (128 bits or more).
2. Alice inputs the key K to the stream cipher. The stream cipher 'stretches' the key K into a long keystream S. The keystream S is as long as the bit-length of the plaintext P.

$$S = StreamCipher(K), \quad length(S) = length(P)$$

3. We encrypt the plaintext P by XORing it with the generated keystream S (exactly like the OTP).

$$C = enc(P, K) = P \oplus S$$

Stream Ciphers

- The very long OTP keys makes the scheme unusable
- Stream ciphers provide a good alternative

Stream cipher encryption/decryption algorithm:

1. Alice (A) and Bob (B) share a reasonably-sized key K (128 bits or more).
2. Alice inputs the key K to the stream cipher. The stream cipher 'stretches' the key K into a long keystream S. The keystream S is as long as the bit-length of the plaintext P.

$$S = StreamCipher(K), \quad length(S) = length(P)$$

3. We encrypt the plaintext P by XORing it with the generated keystream S (exactly like the OTP).

$$C = enc(P, K) = P \oplus S$$

4. The decryption requires that Bob uses the shared key K to generate the same keystream S as Alice. The plaintext P is recovered by XORing the keystream S with the ciphertext C.

$$P = dec(C, K) = C \oplus S$$

Stream Ciphers

Initialization Vector (IV):

- Like in the OTP, parties A and B should avoid keystream repetitions
- This requires keeping track of the keystream S they used
- A lost message between A and B will 'desyncronize' the keystream

Stream Ciphers

Initialization Vector (IV):

- Like in the OTP, parties A and B should avoid keystream repetitions
- This requires keeping track of the keystream S they used
- A lost message between A and B will 'desyncronize' the keystream

To solve these issues stream ciphers use an Initialization Vector IV, a value that is public but should not be repeated under the same key. IV 'synchronises' the keystream.

Stream Ciphers

Initialization Vector (IV):

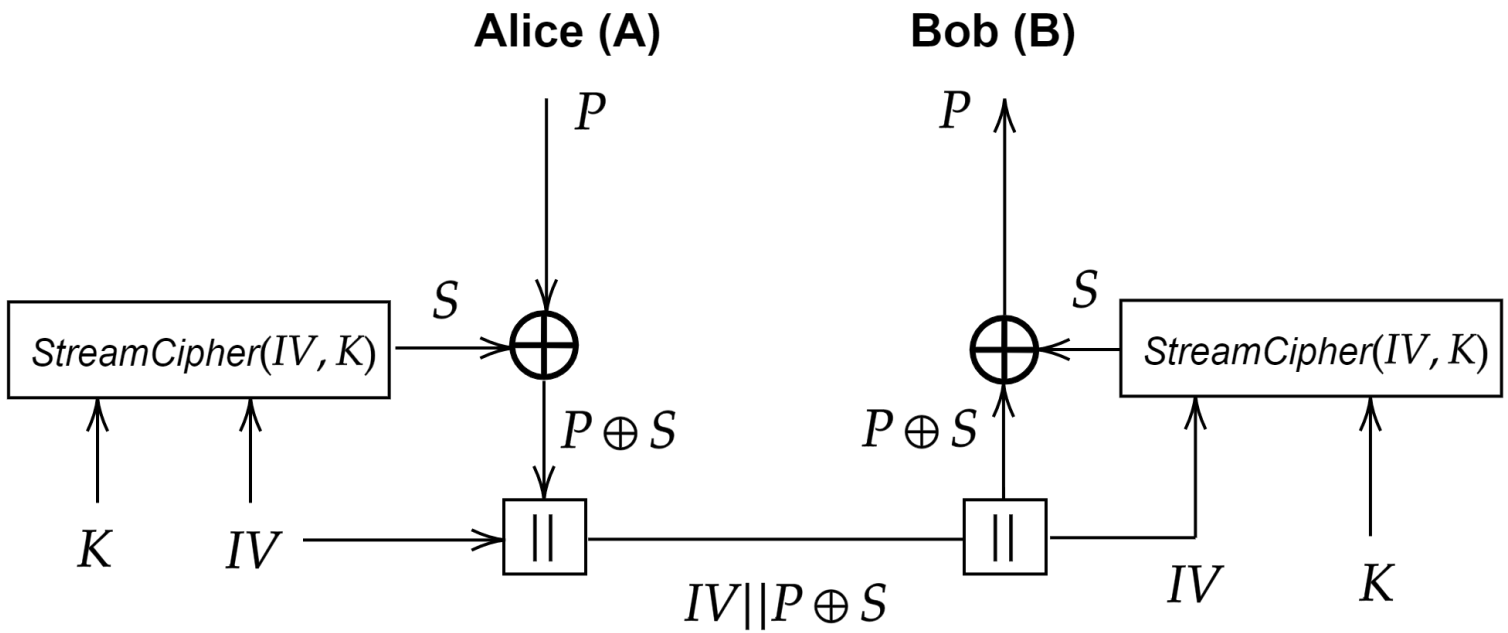
- Like in the OTP, parties A and B should avoid keystream repetitions
- This requires keeping track of the keystream S they used
- A lost message between A and B will 'desyncronize' the keystream

To solve these issues stream ciphers use an Initialization Vector IV, a value that is public but should not be repeated under the same key. IV 'synchronises' the keystream.

1. Both the IV and the key K are input to the stream cipher and result in the keystream S
$$S = StreamCipher(IV, K)$$
2. Every message between A and B has a different IV but the key K is the same
3. Now the transmission consists of the IV along with the ciphertext C
$$C = P \oplus S, \quad \text{transmit: } IV \,||\, C$$
4. Party B receives the transmission and detaches the IV. Both the IV and the shared key K are input to the $StreamCipher(\cdot)$, S is produced and decryption is carried out
$$S = StreamCipher(IV, K), \quad P = C \oplus S$$

Stream Ciphers

- Stream cipher encryption/decryption between Alice (A) and Bob (B)
- Alice and Bob share the same secret key K before $enc(\cdot), dec(\cdot)$ takes place



Stream Ciphers

Security of stream ciphers:

- Unlike the OTP that has perfect secrecy, stream ciphers have a weaker notion of security
- Stream ciphers are **computationally secure**, i.e. they are secure against attackers with bounded computational power

  e.g. if the key K of a stream cipher is 128 bits then an adversary that tries all possible keys (2^{128} keys) can crack it. This **brute-force attack** is impossible when using the OTP.
- Thus stream ciphers trade perfect secrecy for a reasonable key size $|\mathcal{K}|$
- The keystream S should be unpredictable and approximate a truly random number stream. This requires that the stream cipher passes various statistical tests.

Stream Ciphers

Final notes on stream ciphers:

- Stream ciphers are widely used, yet are less popular compared to block ciphers
- Useful in applications that require fast streaming of data: encrypting is just a XOR operation
- Useful in applications with limited hardware resources like IoT and RFID tags
- Less useful in applications were data are partitioned in blocks like file transfer, emails, databases
- The eSTREAM competition established several secure and efficient stream ciphers
  `https://www.ecrypt.eu.org/stream/`

WEP Attacks

WEP Attacks

- The 802.11 standard for wireless network communication included the **Wired Equivalent Privacy (WEP)** protocol
- WEP became a WiFi security standard in 1999, aiming to make wireless security as strong as wired security

WEP Attacks

- The 802.11 standard for wireless network communication included the **Wired Equivalent Privacy (WEP)** protocol
- WEP became a WiFi security standard in 1999, aiming to make wireless security as strong as wired security

The WEP protocol

- To establish the integrity property, WEP uses the CRC-32 integrity checksum algorithm. Before encrypting a message M, we compute its checksum and generate the following plaintext P

$$CRC = CRC32(M), \quad P = M \,||\, CRC$$

WEP Attacks

- The 802.11 standard for wireless network communication included the **Wired Equivalent Privacy (WEP)** protocol
- WEP became a WiFi security standard in 1999, aiming to make wireless security as strong as wired security

The WEP protocol

- To establish the integrity property, WEP uses the CRC-32 integrity checksum algorithm. Before encrypting a message M, we compute its checksum and generate the following plaintext P

$$CRC = CRC32(M), \quad P = M \,||\, CRC$$

- To establish confidentiality, WEP encrypts P with the symmetric stream cipher RC4, using pre-shared key K and initialization vector IV

$$S = RC4(IV, K), \quad C = P \oplus S$$

WEP Attacks

- The ciphertext C is concatenated with the initialization vector IV and transmitted

$$\text{transmit: } IV \,||\, C$$

WEP Attacks

- The ciphertext C is concatenated with the initialization vector IV and transmitted

$$\text{transmit: } IV \,||\, C$$

- The decryption process detaches IV, generates the same stream S and recovers the plaintext P, consisting of the message M and the checksum CRC

$$S = RC4(IV, K), \quad P = C \oplus S, \quad P = M \,||\, CRC$$

WEP Attacks

- The ciphertext C is concatenated with the initialization vector IV and transmitted

$$\text{transmit: } IV \,||\, C$$

- The decryption process detaches IV, generates the same stream S and recovers the plaintext P, consisting of the message M and the checksum CRC

$$S = RC4(IV, K), \quad P = C \oplus S, \quad P = M \,||\, CRC$$

- The CRC32 checksum on message M is re-computed (CRC') and compared to the received checksum CRC

$$CRC' = CRC32(M), \quad \text{check: } CRC' == CRC$$

- If the checksums CRC' and CRC match, then the message M is accepted

WEP Attacks

- The ciphertext C is concatenated with the initialization vector IV and transmitted

$$\text{transmit: } IV \,||\, C$$

- The decryption process detaches IV, generates the same stream S and recovers the plaintext P, consisting of the message M and the checksum CRC

$$S = RC4(IV, K), \quad P = C \oplus S, \quad P = M \,||\, CRC$$

- The CRC32 checksum on message M is re-computed (CRC') and compared to the received checksum CRC

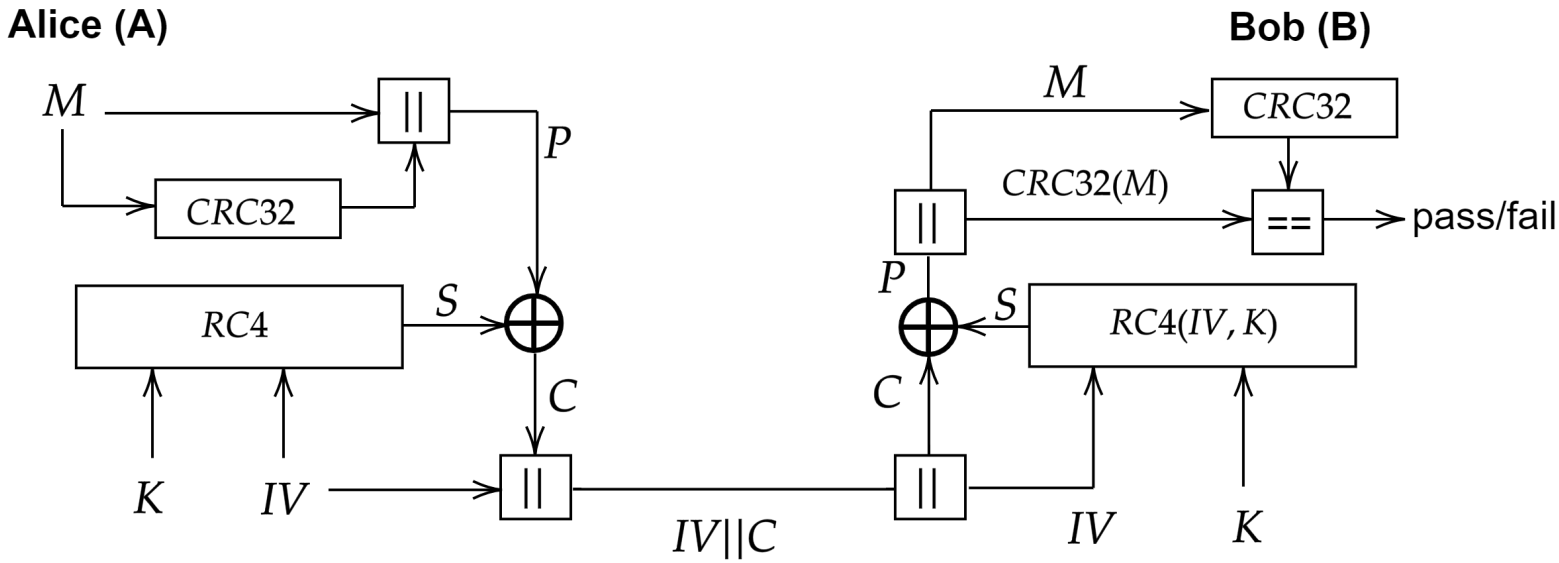
$$CRC' = CRC32(M), \quad \text{check: } CRC' == CRC$$

- If the checksums CRC' and CRC match, then the message M is accepted

The goal of WEP was to provide confidentiality and integrity, yet several cryptographic attacks resulted in a broken protocol with neither (and free neighbor's WiFi back in 2006).

WEP Attacks

- WEP protocol communication



WEP Attacks

Brute-force attack:

- WEP specified the cipher's key length as 40 bits
- Crypto Wars: US classified encryption algorithms as munition and controlled the export of cryptography (see also Bernstein v. United States court case)
- US limited the access of foreign nations to cryptographic technology
- NSA was actively undermining the security level of encryption algorithms e.g. by specifying small bit lengths for the key or promoting the usage of compromised algorithms

WEP Attacks

Brute-force attack:

- WEP specified the cipher's key length as 40 bits
- Crypto Wars: US classified encryption algorithms as munition and controlled the export of cryptography (see also Bernstein v. United States court case)
- US limited the access of foreign nations to cryptographic technology
- NSA was actively undermining the security level of encryption algorithms e.g. by specifying small bit lengths for the key or promoting the usage of compromised algorithms

- Implementing WEP with 40-bit keys is broken simply through a brute-force attack with modest computational resources
- Manufacturers extended the protocol to 104-bit keys to prevent an attack

WEP Attacks

IV reuse attack:

Idea: Say that WEP uses the same IV to encrypt two different plaintexts P_1 and P_2

$$C_1 = P_1 \oplus RC4(IV, K), \quad C_2 = P_2 \oplus RC4(IV, K)$$

$$C_1 \oplus C_2 = P_1 \oplus P_2$$

WEP Attacks

IV reuse attack:

Idea: Say that WEP uses the same IV to encrypt two different plaintexts P_1 and P_2

$$C_1 = P_1 \oplus RC4(IV, K), \quad C_2 = P_2 \oplus RC4(IV, K)$$

$$C_1 \oplus C_2 = P_1 \oplus P_2$$

- Reusing the IV in different plaintexts results in keystream repetition
- When the keystream is repeated, stream ciphers like RC4 are vulnerable to the **two-pad attack** (exactly like the OTP)
- WEP suggests to use a different IV in every transmission to prevent keystream repetitions from occurring but does not specify precisely how.

WEP Attacks

What causes *IV* repetitions in WEP?

Bad implementation:

- WEP suggests to change *IV*s but does not enforce it
- An implementation that always sends the same *IV* is still WEP-compliant

WEP Attacks

What causes IV repetitions in WEP?

Bad implementation:

- WEP suggests to change IVs but does not enforce it
- An implementation that always sends the same IV is still WEP-compliant

Device resets:

- Certain WEP implementations set $IV = 0$ every time they are reset and after every transmission they increase IV by 1
- Resets occur often thus low-valued IVs are repeated

WEP Attacks

What causes *IV* repetitions in WEP?

Bad implementation:

- WEP suggests to change *IV*s but does not enforce it
- An implementation that always sends the same *IV* is still WEP-compliant

Device resets:

- Certain WEP implementations set $IV = 0$ every time they are reset and after every transmission they increase *IV* by 1
- Resets occur often thus low-valued *IV*s are repeated

Limited *IV* length:

- WEP specifies the *IV* length as 24 bits
- Using incremental *IV*s, it takes $2^{24} \approx 17$ million packet transmissions to produce repetitions

  Assuming a packet size of 1500 bytes and bandwidth of 5Mbps will produce repetitions after $\frac{2^{24} * 1500}{5 * 10^6 * 8} \approx 12$ hours of transmission
- Using random *IV*s takes far less packet transmissions to produce repetitions, due to the birthday paradox

  After observing approx. $\sqrt{2^{24}} = 4096$ transmissions there is a 50% probability of *IV* repetition. It typically takes minutes to transmit such an number of packets.

WEP Attacks

How to exploit *IV* repetitions in WEP? We can use it to break protocol **confidentiality**

Recover the plaintext:

- When the *IV* is repeated, it holds that $C_1 \oplus C_2 = P_1 \oplus P_2$ and guessing correctly P_1 reveals P_2
- Protocols follow a strict, well-defined structure and the plaintexts are often easy to guess
- Alternatively we can send indirectly a message to the target. If the message is communicated to the target with a repeated *IV* then we can recover other messages under this *IV*.

WEP Attacks

How to exploit IV repetitions in WEP? We can use it to break protocol **confidentiality**

Recover the plaintext:

- When the IV is repeated, it holds that $C_1 \oplus C_2 = P_1 \oplus P_2$ and guessing correctly P_1 reveals P_2
- Protocols follow a strict, well-defined structure and the plaintexts are often easy to guess
- Alternatively we can send indirectly a message to the target. If the message is communicated to the target with a repeated IV then we can recover other messages under this IV.

Recover the keystream:

- Note that once P_1, P_2 are recovered, the keystream $RC4(IV, K)$ is also recovered
- Any plaintext encrypted with the same IV can be now decrypted
- The attacker can build a dictionary of keystreams for every IV

WEP Attacks

How to exploit IV repetitions in WEP? We can use it to break protocol **confidentiality**

Recover the plaintext:

- When the IV is repeated, it holds that $C_1 \oplus C_2 = P_1 \oplus P_2$ and guessing correctly P_1 reveals P_2
- Protocols follow a strict, well-defined structure and the plaintexts are often easy to guess
- Alternatively we can send indirectly a message to the target. If the message is communicated to the target with a repeated IV then we can recover other messages under this IV.

Recover the keystream:

- Note that once P_1, P_2 are recovered, the keystream $RC4(IV, K)$ is also recovered
- Any plaintext encrypted with the same IV can be now decrypted
- The attacker can build a dictionary of keystreams for every IV

Some WEP configurations used a single key for multiple users which causes even more IV repetitions and makes key replacement harder. We conclude that the WEP protocol does not manage keys in an effective manner.

WEP Attacks

Message modification attack:

Idea: The $CRC32(\cdot)$ checksum is a linear function i.e.

$$CRC32(X \oplus Y) = CRC32(X) \oplus CRC32(Y)$$

We will use this property to make controlled modifications to any ciphertext C, while passing the integrity check of the WEP protocol.

- We capture WEP ciphertext C that originates from an unknown message M
- We will modify C and construct C' that decrypts to M' such that $M' = M \oplus \Delta$ and Δ is our modification choice

  i.e. we will be able to alter any message M as we choose, just by modifying its ciphertext C

WEP Attack

We show how to construct C' from C such that it decrypts to $M' = M \oplus \Delta$ and passes the integrity check

1. We choose the Δ and compute the checksum $CRC32(\Delta)$
2. We XOR the ciphertext C with Δ and $CRC32(\Delta)$

$$C' = C \oplus [\Delta \mid\mid CRC32(\Delta)]$$

3. Using the definition of the WEP encryption

$$C' = RC4(IV, K) \oplus [M \mid\mid CRC32(M)] \oplus [\Delta \mid\mid CRC32(\Delta)] \iff$$

$$C' = RC4(IV, K) \oplus [M \oplus \Delta \mid\mid CRC32(M) \oplus CRC32(\Delta)]$$

4. Using the linearity of $CRC32(\cdot)$

$$C' = RC4(IV, K) \oplus [M \oplus \Delta \mid\mid CRC32(M \oplus \Delta)] \iff$$

$$C' = RC4(IV, K) \oplus [M' \mid\mid CRC32(M')]$$

WEP Attacks

- Decrypting the constructed C' yields the message M' and its checksum

  $$dec(C', K) = RC4(IV, K) \oplus RC4(IV, K) \oplus [M' \,||\, CRC32(M')] = [M' \,||\, CRC32(M')]$$

- The plaintext $[M' \,||\, CRC32(M')]$ will pass the WEP integrity check
- Thus any modifications on M will remain undetected and the attack breaks protocol **integrity**

WEP Attacks

- Decrypting the constructed C' yields the message M' and its checksum

  $$dec(C', K) = RC4(IV, K) \oplus RC4(IV, K) \oplus [M' \,||\, CRC32(M')] = [M' \,||\, CRC32(M')]$$

- The plaintext $[M' \,||\, CRC32(M')]$ will pass the WEP integrity check
- Thus any modifications on M will remain undetected and the attack breaks protocol **integrity**

- The root cause is the checksum CRC-32
- CRC-32 is a general-purpose error-detecting code that detects accidental transmission errors and cannot not always detect malicious tampering
- We need cryptographic-secure methods to check integrity such as hash functions and digital signatures

better not a linear checksum function

Block Ciphers

Block Ciphers

Block ciphers are the most popular symmetric encryption algorithms and are an alternative cipher class to stream ciphers. They are **computationally secure** encryption methods.

Block Ciphers

Block ciphers are the most popular symmetric encryption algorithms and are an alternative cipher class to stream ciphers. They are **computationally secure** encryption methods.

- Block ciphers split the plaintext P to fixed size **blocks**. Every block P_i has the same **blocklength** (block size in bits).

$$[P_1, P_2, \ldots, P_t] \leftarrow P$$

Block Ciphers

Block ciphers are the most popular symmetric encryption algorithms and are an alternative cipher class to stream ciphers. They are **computationally secure** encryption methods.

- Block ciphers split the plaintext P to fixed size **blocks**. Every block P_i has the same **blocklength** (block size in bits).

$$[P_1, P_2, \ldots, P_t] \leftarrow P$$

- A function $f(\cdot)$ that mixes the block P_i with the key K is applied r times. We call $f(\cdot)$ the **round function** and r the **number of rounds**.

$$C_i = enc(P_i, K) = \Big\{ f(P_i, K) \Big\}_r$$

Block Ciphers

Block ciphers are the most popular symmetric encryption algorithms and are an alternative cipher class to stream ciphers. They are **computationally secure** encryption methods.

- Block ciphers split the plaintext P to fixed size **blocks**. Every block P_i has the same **blocklength** (block size in bits).

$$[P_1, P_2, \ldots, P_t] \leftarrow P$$

- A function $f(\cdot)$ that mixes the block P_i with the key K is applied r times. We call $f(\cdot)$ the **round function** and r the **number of rounds**.

$$C_i = enc(P_i, K) = \Big\{ f(P_i, K) \Big\}_r$$

- The encryption is repeated in all blocks P_i and the result is fixed sized ciphertext blocks C_i

$$[C_1, C_2, \ldots, C_t] = [enc(P_1, K), enc(P_2, K), \ldots, enc(P_t, K)]$$

Block Ciphers

Feistel cipher encryption algorithm:

- Feistel cipher is not a specific cipher but a design principle for block ciphers
- The plaintext block P is split to left and right parts

$$[L_0, R_0] \leftarrow P$$

Block Ciphers

Feistel cipher encryption algorithm:

- Feistel cipher is not a specific cipher but a design principle for block ciphers
- The plaintext block P is split to left and right parts

$$[L_0, R_0] \leftarrow P$$

- Assume a round function $f(\cdot)$, key K and r rounds
  The Feistel cipher at round $i = 1, 2, \ldots, r$ operates as follows

$$L_i = R_{i-1}, \qquad R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$$

Block Ciphers

Feistel cipher encryption algorithm:

- Feistel cipher is not a specific cipher but a design principle for block ciphers
- The plaintext block P is split to left and right parts

$$[L_0, R_0] \leftarrow P$$

- Assume a round function $f(\cdot)$, key K and r rounds
  The Feistel cipher at round $i = 1, 2, \ldots, r$ operates as follows

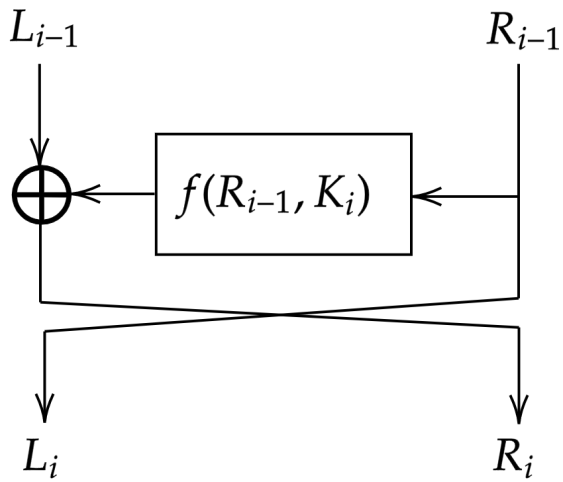
$$L_i = R_{i-1}, \qquad R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$$

- K_i is the roundkey for round i, a quantity derived from the cipher key K using a **key schedule** algorithm

$$K_i = \mathit{schedule}(K, i), \quad i = 1, 2, \ldots, r$$

Block Ciphers

- Feistel cipher operating on round i with roundkey K_i



- The ciphertext block C is the output of the final round

$$L_r = R_{r-1}, \qquad R_r = L_{r-1} \oplus f(R_{r-1}, K_r)$$

$$C = [L_r, R_r]$$

Block Ciphers

Feistel cipher decryption algorithm:

- The Feistel cipher can easily decrypt by running the process backwards

$$R_{i-1} = L_i, \qquad L_{i-1} = R_i \oplus f(R_{i-1}, K_i)$$

- The plaintext block P is recovered for $i = 0$

$$P = [L_0, R_0]$$

- The decryption process works irrespective of the round function $f(\cdot)$. Note that the security of any Feistel structure lies on the security of $f(\cdot)$

Block Ciphers

DES cipher: To address the need for commercial cryptography, the Data Encryption Standard (DES) was developed in 1975 by IBM and in part by the NSA.

- DES is a Feistel cipher with $r = 16$ rounds and round function $f(\cdot)$
- The blocklength is 64 bits and the keylength is 56 bits
- The length of each roundkey K_i is 48 bits

Block Ciphers

DES cipher encryption algorithm:

1. The input to DES is a 64-bit plaintext block P. We index the bits of P left-to-right with $i = 1, 2, \ldots, 64$.

   The DES initial permutation $IP(\cdot)$ is permuting the bits of P in the following manner

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$IP(i)$	58	50	42	34	26	18	10	2	60	52	44	36	28	20	12	4
i	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
$IP(i)$	62	54	46	38	30	22	14	6	64	56	48	40	32	24	16	8
i	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
$IP(i)$	57	49	41	33	25	17	9	1	59	51	43	35	27	19	11	3
i	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
$IP(i)$	61	53	45	37	29	21	13	5	63	55	47	39	31	23	15	7

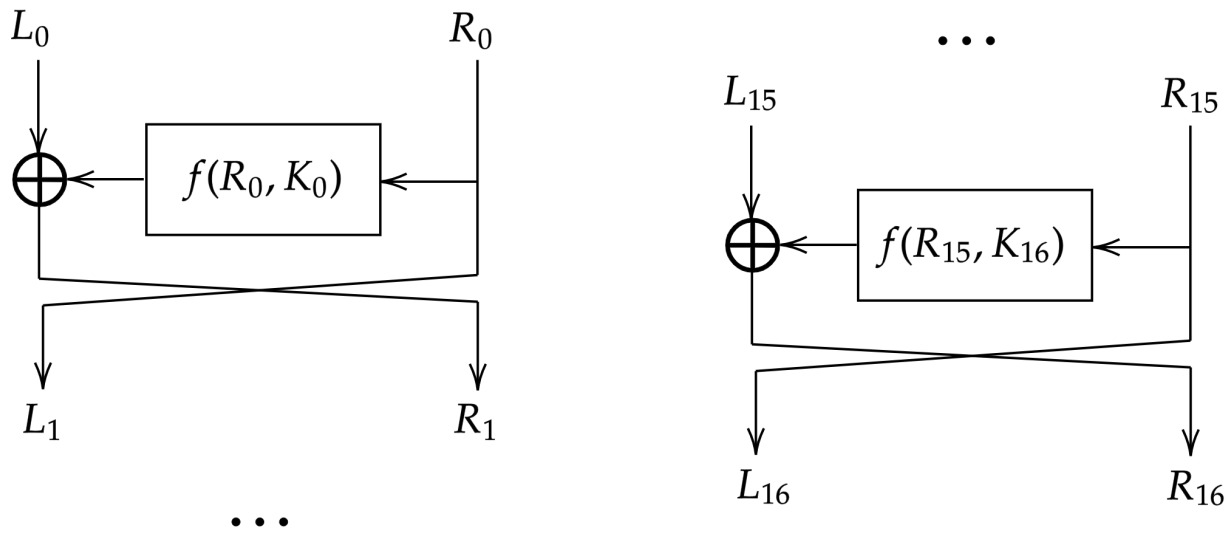
   e.g. the 7th bit of value $IP(P)$ is equal to the 64th bit of plaintext P

Block Ciphers

2. The output of $IP(\cdot)$ is split to 32-bit left and right parts L_0 and R_0

$$[L_0, R_0] \leftarrow IP(P)$$

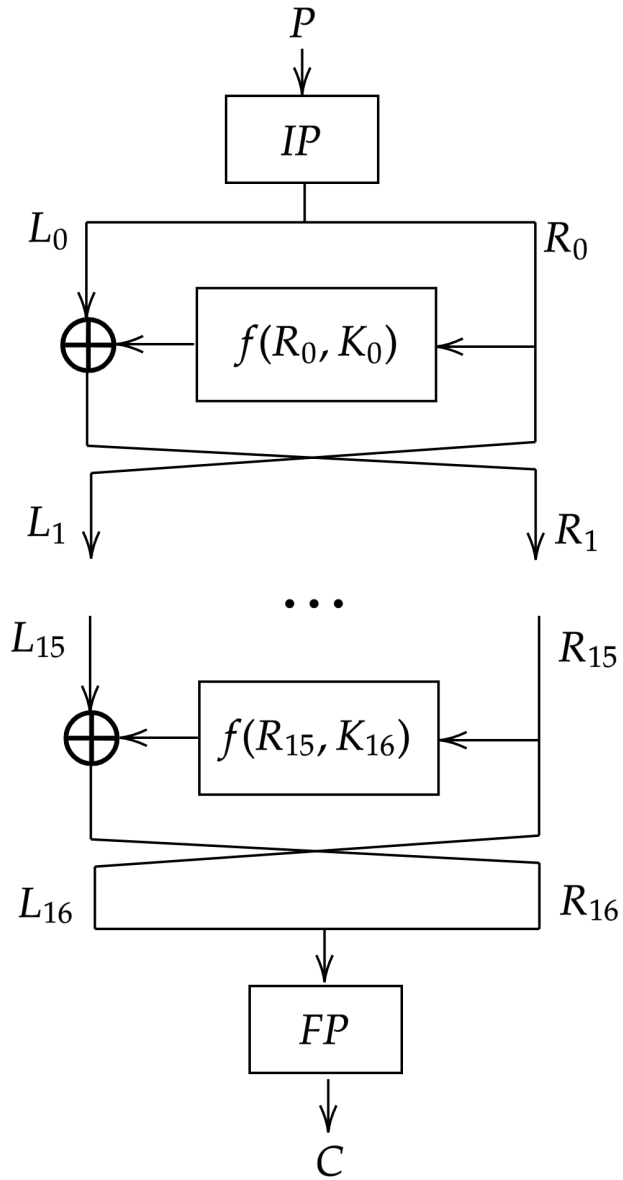
3. DES performs $r = 16$ Feistel rounds using the DES round function $f(\cdot)$ and the roundkey K_i, generated by the DES key schedule algorithm



Block Ciphers

4. A final permutation $FP(\cdot)$ is applied to the output of the Feistel round 16 and produces the 64-bit ciphertext. $FP(\cdot)$ is the inverse of $IP(\cdot)$

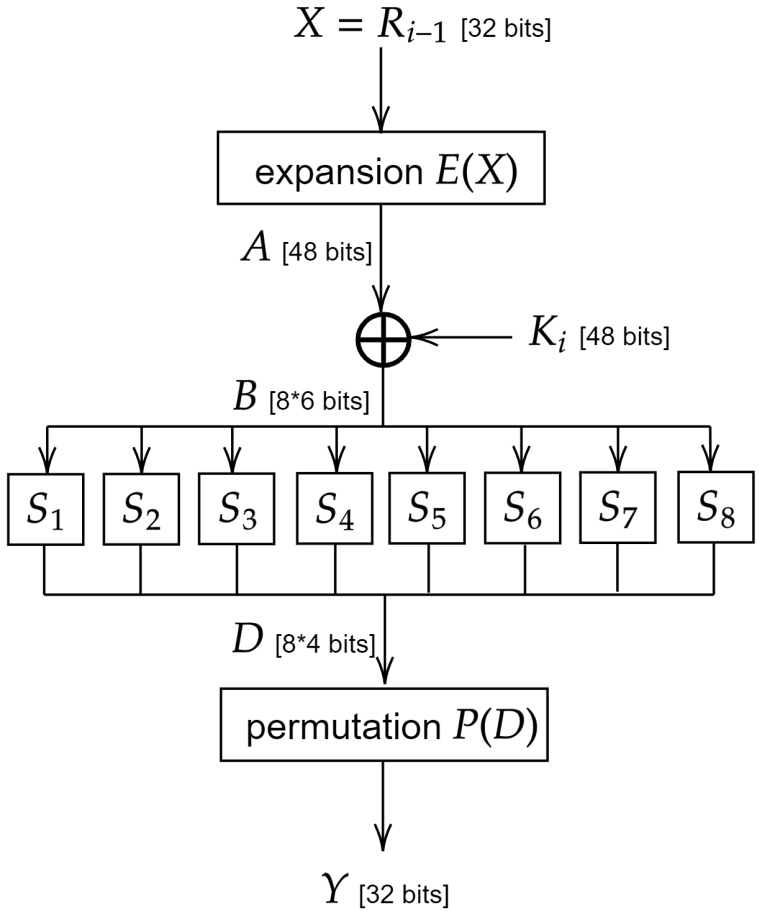
$$C = FP([L_{16}, R_{16}]), \qquad FP(IP(X)) = X$$



Block Ciphers

Inside the DES round function $f(R_i, K_i)$

1. The 32-bit input X is expanded to 48-bit value $E(X)$
2. The 48-bit value $E(X)$ is XORed with the roundkey K_i
3. The 48-bit value $E(X)$ is split to 8 $\times$ 6-bit values
4. Every 6-bit value is input to the respective DES sbox $S_1(\cdot)$ till $S_8(\cdot)$
5. The sbox maps 6-bit inputs to 4-bit outputs
6. The 8 $\times$ 4-bit sbox outputs are permuted with permutation $P(\cdot)$



Block Ciphers

- The expansion $E(X)$ permutes and expands the 32-bit input X to 48-bit value A

$$A = E(X)$$

- The expansion table shows the bit permutations and repetitions

bit j of A	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
bit i of X	32	1	2	3	4	5	4	5	6	7	8	9	8	9	10	11
bit j of A	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
bit i of X	12	13	12	13	14	15	16	17	16	17	18	19	20	21	20	21
bit j of A	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
bit i of X	22	23	24	25	24	25	26	27	28	29	28	29	30	31	32	1

The table reads as: "bit j of A is equal to bit i of X"

e.g. the 1st bit of A is equal to the 32nd bit of X

Block Ciphers

- The value B is the result of the addition of A with the roundkey K_i and is split to 8×6-bit values

$$B = A \oplus K_i, \qquad [B_1, B_2, \ldots, B_8] \leftarrow B$$

- The DES cipher applies 8 different sboxes that map 6-bit values to 4-bit values

$$[D_1, D_2, \ldots, D_8] = [S_1(B_1), S_2(B_2), \ldots, S_8(B_8)]$$

Block Ciphers

- The value B is the result of the addition of A with the roundkey K_i and is split to 8×6-bit values

$$B = A \oplus K_i, \qquad [B_1, B_2, \ldots, B_8] \leftarrow B$$

- The DES cipher applies 8 different sboxes that map 6-bit values to 4-bit values

$$[D_1, D_2, \ldots, D_8] = [S_1(B_1), S_2(B_2), \ldots, S_8(B_8)]$$

- Below we show sbox S_1 of DES

map from 6bit input to 4bit output

	x0000x	x0001x	x0010x	x0011x	x0100x	x0101x	x0110x	x0111x
0yyyy0	14	4	13	1	2	15	11	8
0yyyy1	0	15	7	4	14	2	13	1
1yyyy0	4	1	14	8	13	6	2	11
1yyyy1	15	12	8	2	4	9	1	7
	x1000x	x1001x	x1010x	x1011x	x1100x	x1101x	x1110x	x1111x
0yyyy0	3	10	6	12	5	9	0	7
0yyyy1	10	6	12	11	9	5	3	8
1yyyy0	15	12	9	7	3	10	5	0
1yyyy1	5	11	3	14	10	0	6	13

The table reads as: "We receive a 6-bit sbox input $x = x_0x_1x_2x_3x_4x_5$ in binary. Bits x_0, x_5 specify the row and bits x_1, x_2, x_3, x_4 specify the column"

e.g for input $x = 42 = 010101$ we have $x_0, x_5 = 01$ (row 2 of 4) and $x_1, x_2, x_3, x_4 = 1010$ (column 11 of 16), thus the sbox output is 12

Block Ciphers

- The 32-bit sbox outputs D are permuted using the round permutation $P(\cdot)$ and compute the 32-bit output of the round function $f(\cdot)$

$$D = [D_1, D_2, \ldots, D_8], \qquad Y = P(D)$$

The round permutation $P(\cdot)$ is permuting bits in the following manner

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$P(i)$	16	7	20	21	29	12	28	17	1	15	23	26	5	18	31	10
i	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
$P(i)$	2	8	24	14	32	27	3	9	19	13	30	6	22	11	4	25

The table reads as: "the bit in position i is re-wired to position $P(i)$"

e.g. the 2nd bit of value Y is equal to the 17th bit of value D

DES Attacks

DES Attacks

Bruteforce attack:

- The DES keylength is 56 bits which is not enough to prevent bruteforce attacks
- Specialized hardware was developed to crack DES efficiently
  `https://www.copacobana.org/`

DES Attacks

Bruteforce attack:

- The DES keylength is 56 bits which is not enough to prevent bruteforce attacks
- Specialized hardware was developed to crack DES efficiently
  `https://www.copacobana.org/`

Solution: Applying the DES 3 times, with 3×56-bit keys K_1, K_2, K_3 resulted in the 3DES standard.

$$C = enc(dec(enc(P, K_1), K_2), K_3)$$

$$P = dec(enc(dec(C, K_3), K_2), K_1)$$

This 3DES structure provides a key length of 168 bits

DES Attacks

DES design issues:

- The sboxes $S_1, S_2, \ldots, S_8$ presented are *not* the original sboxes developed by IBM but the ones suggested by NSA

  "We sent the [DES] S-boxes off to Washington. They came back and were all different"

- The DES design was opaque and many feared NSA backdoors
- It appears that the NSA secured the sboxes against differential cryptanalysis – an attack publicly unknown at that time
- Edward Snowden revealed that the NSA proposed the Dual_EC_DRBG algorithm while being aware of its vulnerabilities

DES Attacks

DES design issues:

- The sboxes $S_1, S_2, \ldots, S_8$ presented are *not* the original sboxes developed by IBM but the ones suggested by NSA

  "We sent the [DES] S-boxes off to Washington. They came back and were all different"

- The DES design was opaque and many feared NSA backdoors
- It appears that the NSA secured the sboxes against differential cryptanalysis – an attack publicly unknown at that time
- Edward Snowden revealed that the NSA proposed the Dual_EC_DRBG algorithm while being aware of its vulnerabilities

Solution: A new standard was developed, after a lengthy competition. The Advanced Encryption Standard (AES) was established, using the Rijndael algorithm by Joan Daemen and Vincent Rijmen in KU Leuven.

AES-128 has a blocklength of 128 bits and keylength of 128 bits

Block Cipher Modes

Block Cipher Modes

- Stream ciphers we can generate a keystream of arbitrary length to XOR with the plaintext
- Block ciphers work with fixed block size, do how do we encrypt long plaintexts?
- We use a certain **block cipher mode** to encrypt multiple blocks with the same block cipher

Cipher Modes

Electronic Code-Book mode (ECB):

- ECB encryption splits the plaintext P to t blocks P_i of b bits each and applies the block cipher $enc(\cdot)$ to every block, under the same key K

$$[P_1, P_2, \ldots, P_t] \leftarrow P$$

$$[C_1, C_2, \ldots, C_t] = [enc(P_1, K), enc(P_2, K), \ldots, enc(P_t, K)]$$

Cipher Modes

Electronic Code-Book mode (ECB):

- ECB encryption splits the plaintext P to t blocks P_i of b bits each and applies the block cipher $enc(\cdot)$ to every block, under the same key K

$$[P_1, P_2, \ldots, P_t] \leftarrow P$$

$$[C_1, C_2, \ldots, C_t] = [enc(P_1, K), enc(P_2, K), \ldots, enc(P_t, K)]$$

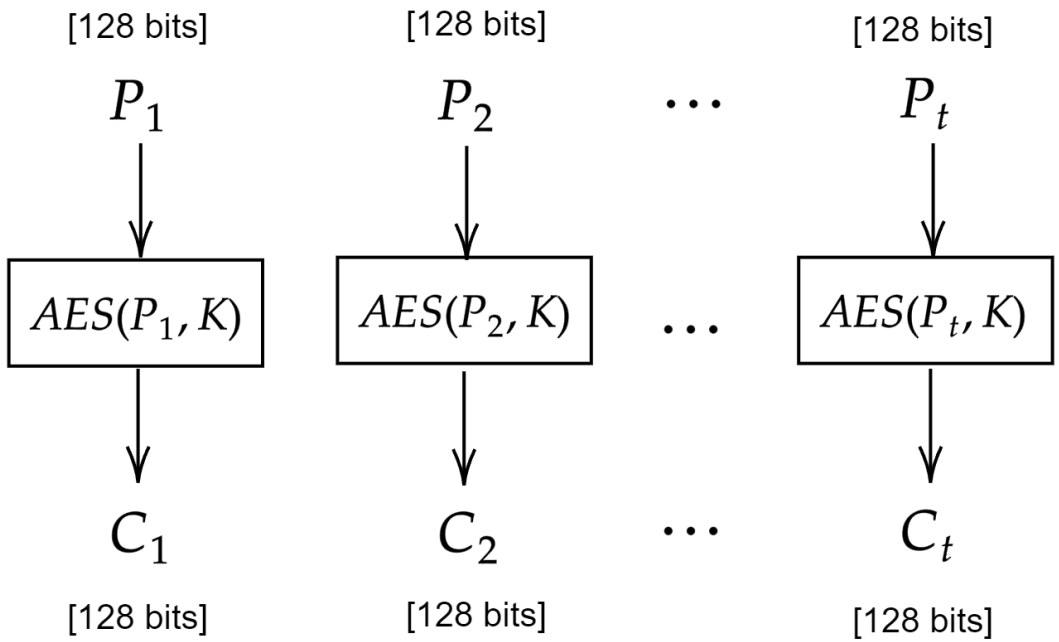
- Decryption is straightforward

$$[P_1, P_2, \ldots, P_t] = [dec(C_1, K), dec(C_2, K), \ldots, dec(C_t, K)]$$

$$P = [P_1, P_2, \ldots, P_t]$$

Block Cipher Modes

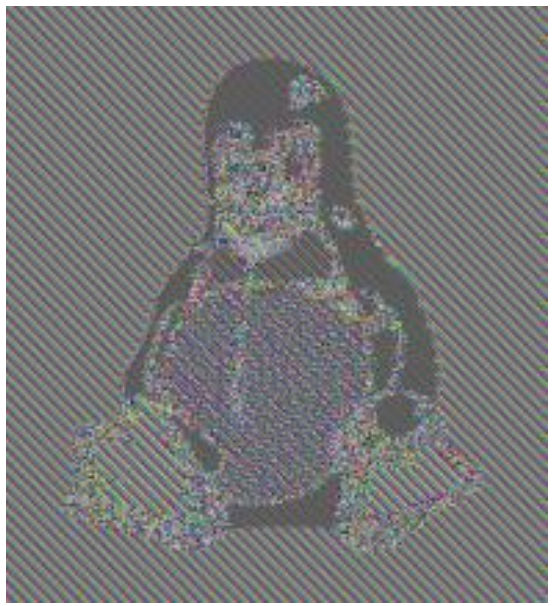
- ECB mode schematic for AES-128



Block Cipher Modes

- ECB uses the same key K in all blocks i.e. if block $P_i = P_j$ then $C_i = C_j$
- If the attacker knows block P_i they can also learn block P_j
- Encrypting an image in ECB mode results (approximately) in the same image

original image



encrypted image in ECB mode

Cipher Modes

Cipher Block Chaining mode (CBC):

- CBC encryption uses the ciphertext of a plaintext block to obscure the next plaintext block

$$[P_1, P_2, \ldots, P_t] \leftarrow P$$

$$C_i = enc(P_i \oplus C_{i-1}, K), \quad \text{for } i = 1, 2, \ldots, t$$

Cipher Modes

Cipher Block Chaining mode (CBC):

- CBC encryption uses the ciphertext of a plaintext block to obscure the next plaintext block

$$[P_1, P_2, \ldots, P_t] \leftarrow P$$

$$C_i = enc(P_i \oplus C_{i-1}, K), \quad \text{for } i = 1, 2, \ldots, t$$

$$[C_1, C_2, \ldots, C_t] = [enc(P_1 \oplus C_0, K), enc(P_2 \oplus C_1, K), \ldots, enc(P_t \oplus C_{t-1}, K)]$$

where C_0 is initialization vector IV i.e. $C_1 = enc(P_1 \oplus IV, K)$

Cipher Modes

Cipher Block Chaining mode (CBC):

- CBC encryption uses the ciphertext of a plaintext block to obscure the next plaintext block

$$[P_1, P_2, \ldots, P_t] \leftarrow P$$

$$C_i = enc(P_i \oplus C_{i-1}, K), \quad \text{for } i = 1, 2, \ldots, t$$

$$[C_1, C_2, \ldots, C_t] = [enc(P_1 \oplus C_0, K), enc(P_2 \oplus C_1, K), \ldots, enc(P_t \oplus C_{t-1}, K)]$$

where C_0 is initialization vector IV i.e. $C_1 = enc(P_1 \oplus IV, K)$

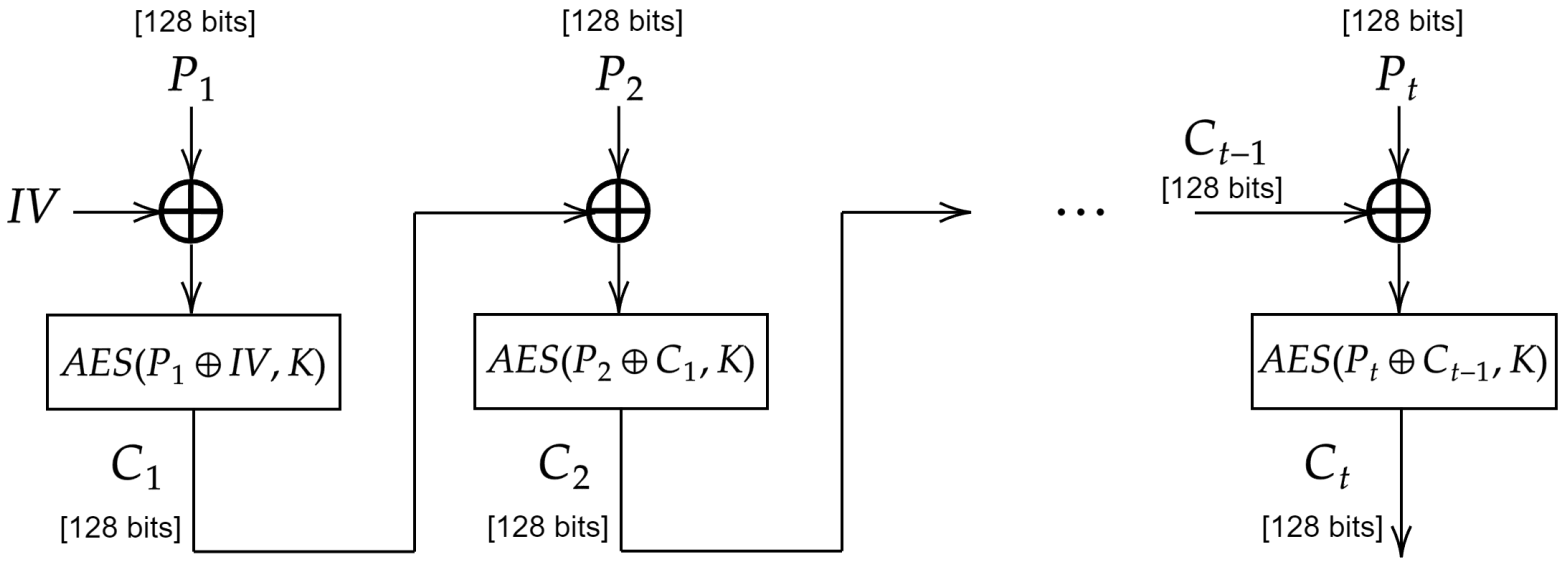
- CBC decryption formula

$$P_i = dec(C_i, K) \oplus C_{i-1}, \quad \text{for } i = 1, 2, \ldots, t$$

decryption start w C_1

Block Cipher Modes

- CBC mode schematic for AES-128



Block Cipher Modes

Establishing integrity with CBC encryption:

1. We construct a **message authentication code (*MAC*)** to ensure data integrity by encrypting in CBC mode and keeping only the final ciphertext block

$$C_1 = enc(P_1 \oplus IV, K), C_2 = enc(P_2 \oplus C_1, K), \ldots, MAC = enc(P_t \oplus C_{t-1}, K)$$

2. We transmit the plaintext P, the IV and the MAC
3. The receiver repeats the same CBC encryption and computes MAC'
4. If the received MAC is equal to the computed MAC' then the data has not been tampered with

Block Cipher Modes

Establishing integrity with CBC encryption:

1. We construct a **message authentication code (*MAC*)** to ensure data integrity by encrypting in CBC mode and keeping only the final ciphertext block

$$C_1 = enc(P_1 \oplus IV, K), C_2 = enc(P_2 \oplus C_1, K), \ldots, MAC = enc(P_t \oplus C_{t-1}, K)$$

2. We transmit the plaintext P, the IV and the MAC here no confidentiality
3. The receiver repeats the same CBC encryption and computes MAC'
4. If the received MAC is equal to the computed MAC' then the data has not been tampered with

- A change to any plaintext block P_i will propagate to all subsequent blocks and finally to the MAC
- Unlike the CRC, we have stronger guarantees when using actual cryptographic constructs
- Integrity can also provided through other cryptographic blocks such as hash functions (HMAC)

Padding Oracle Attack

Padding Oracle Attack

- To encrypt a variable-length plaintext using a block cipher we need a cipher mode of operation (ECB, CBC, etc.) and a **padding method**

Padding Oracle Attack

- To encrypt a variable-length plaintext using a block cipher we need a cipher mode of operation (ECB, CBC, etc.) and a **padding method**
- e.g. AES-128 has a fixed input of 16 bytes (128 bits) but we want to encrypt the following plaintext with length of 27 bytes:

```
plaintext = 0x06 0x10 0xff 0x21 0x42 0x3a 0x99 0x00 0x01 0x99 0x6c
0xdd 0x00 0x01 0xe3 0xd0 0x11 0x15 0x8f 0x8d 0x1e 0x72 0x09 0x62
0x00 0x22 0xef
```

Padding Oracle Attack

- To encrypt a variable-length plaintext using a block cipher we need a cipher mode of operation (ECB, CBC, etc.) and a **padding method**
- e.g. AES-128 has a fixed input of 16 bytes (128 bits) but we want to encrypt the following plaintext with length of 27 bytes:

  `plaintext = 0x06 0x10 0xff 0x21 0x42 0x3a 0x99 0x00 0x01 0x99 0x6c 0xdd 0x00 0x01 0xe3 0xd0 0x11 0x15 0x8f 0x8d 0x1e 0x72 0x09 0x62 0x00 0x22 0xef`

- We need to make sure that the plaintext length is a multiple of the cipher input i.e. we must pad it with 5 bytes to reach length $32 = 2 * 16$

  `plaintext_withpadding = 0x06 0x10 0xff 0x21 0x42 0x3a 0x99 0x00 0x01 0x99 0x6c 0xdd 0x00 0x01 0xe3 0xd0 0x11 0x15 0x8f 0x8d 0x1e 0x72 0x09 0x62 0x00 0x22 0xef PAD PAD PAD PAD PAD`

Padding Oracle Attack

- We then we split the padded plaintext to 16-byte blocks and apply e.g. the CBC cipher mode

  $P_1 =$ `0x06 0x10 0xff 0x21 0x42 0x3a 0x99 0x00 0x01 0x99 0x6c 0xdd 0x00 0x01 0xe3 0xd0`

  $P_2 =$ `0x11 0x15 0x8f 0x8d 0x1e 0x72 0x09 0x62 0x00 0x22 0xef PAD PAD PAD PAD PAD`

  $C_1 = enc(P_1 \oplus IV, K)$

  $C_2 = enc(P_2 \oplus C_1, K)$

Padding Oracle Attack

A commonly used padding algorithm is PKCS#7 [specified in RFC 5652]

- Let plaintext with length n bytes and let block cipher with input b bytes
- Pad the plaintext with m bytes such that $m = b - (n \bmod b)$
- All the padding bytes have the same value, equal to $m = b - (n \bmod b)$

Padding Oracle Attack

A commonly used padding algorithm is PKCS#7 [specified in RFC 5652]

- Let plaintext with length n bytes and let block cipher with input b bytes
- Pad the plaintext with m bytes such that $m = b - (n \bmod b)$
- All the padding bytes have the same value, equal to $m = b - (n \bmod b)$

- In the previous example:
  `plaintext = 0x06 0x10 0xff 0x21 0x42 0x3a 0x99 0x00 0x01 0x99 0x6c 0xdd 0x00 0x01 0xe3 0xd0 0x11 0x15 0x8f 0x8d 0x1e 0x72 0x09 0x62 0x00 0x22 0xef`

- We must pad it with $m = 16 - (27 \bmod 16) = 16 - 11 = 5$ bytes

Padding Oracle Attack

A commonly used padding algorithm is PKCS#7 [specified in RFC 5652]

- Let plaintext with length n bytes and let block cipher with input b bytes
- Pad the plaintext with m bytes such that $m = b - (n \bmod b)$
- All the padding bytes have the same value, equal to $m = b - (n \bmod b)$

- In the previous example:
  `plaintext = 0x06 0x10 0xff 0x21 0x42 0x3a 0x99 0x00 0x01 0x99 0x6c 0xdd 0x00 0x01 0xe3 0xd0 0x11 0x15 0x8f 0x8d 0x1e 0x72 0x09 0x62 0x00 0x22 0xef`
- We must pad it with $m = 16 - (27 \bmod 16) = 16 - 11 = 5$ bytes
- The value of the all the padding bytes is equal to $m = 5$
  `plaintext_withpadding = 0x06 0x10 0xff 0x21 0x42 0x3a 0x99 0x00 0x01 0x99 0x6c 0xdd 0x00 0x01 0xe3 0xd0 0x11 0x15 0x8f 0x8d 0x1e 0x72 0x09 0x62 0x00 0x22 0xef 0x05 0x05 0x05 0x05 0x05`

Padding Oracle Attack

- With $b = 16$ these are all the valid PKCS#7 paddings:

```
0x01
0x02 0x02
0x03 0x03 0x03
...
0x0f 0x0f ...  0x0f
```

Padding Oracle Attack

- With $b = 16$ these are all the valid PKCS#7 paddings:

```
0x01
0x02 0x02
0x03 0x03 0x03
...
0x0f 0x0f ...  0x0f
```

- This padding method is well-defined for input size $b < 256$ bytes
- When the original plaintext length n is already a multiple of the block cipher input b, we add a full extra block to encrypt (case `0x0f 0x0f ... 0x0f`)

Padding Oracle Attack

- With $b = 16$ these are all the valid PKCS#7 paddings:

```
0x01
0x02 0x02
0x03 0x03 0x03
...
0x0f 0x0f ...  0x0f
```

- This padding method is well-defined for input size $b < 256$ bytes
- When the original plaintext length n is already a multiple of the block cipher input b, we add a full extra block to encrypt (case `0x0f 0x0f ... 0x0f`)

- The decryption process decrypts the ciphertext blocks, then checks that the plaintext padding is valid and removes it
  - If all is ok it returns message: `<decryption_ok>`
  - If the padding is invalid then it returns an error: `<incorrect_padding>`

Padding Oracle Attack

Threat Model:

- Say we captured 2 ciphertexts C_1, C_2 from server S to client A
- We don't know the server key K thus we cannot decrypt and recover P_1, P_2
- We can however send C_1, C_2 to the server for decryption
- Note that sending C_1, C_2 is not dangerous: the server will decrypt them and reply with message: `<decryption_ok>`

Padding Oracle Attack

Threat Model:

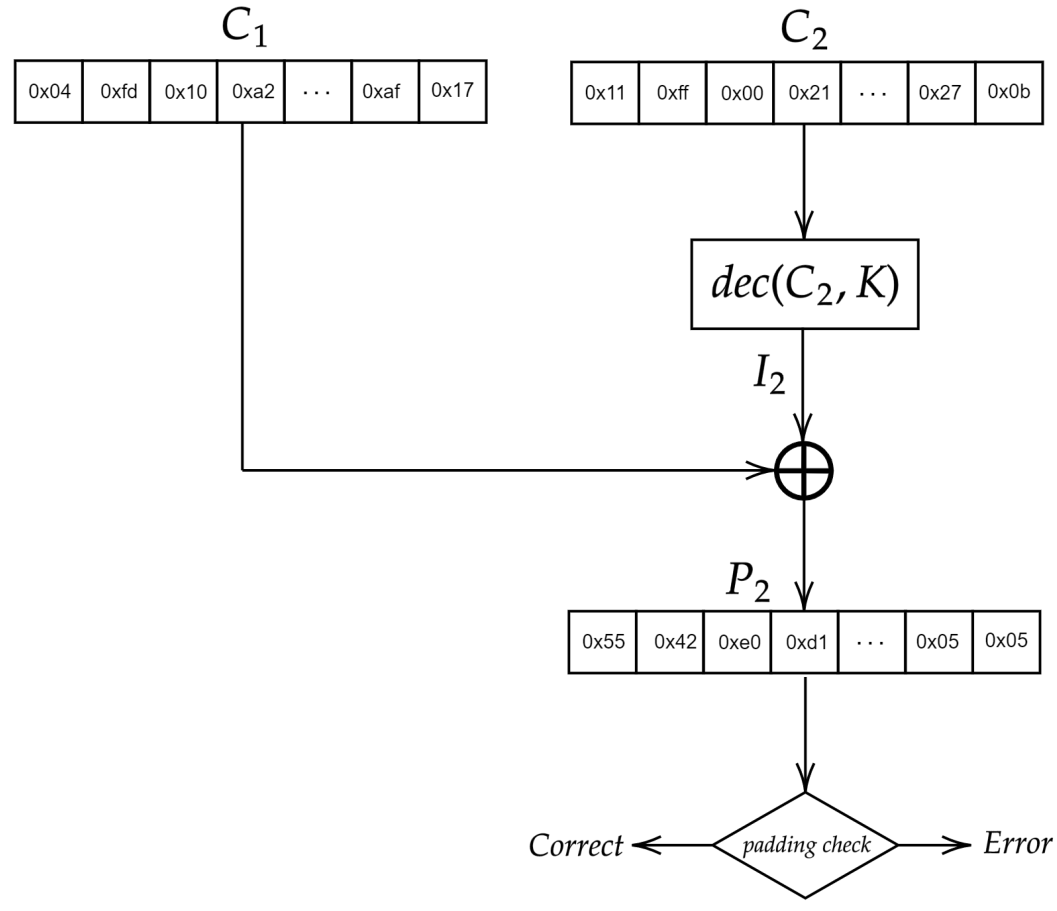
- Say we captured 2 ciphertexts C_1, C_2 from server S to client A
- We don't know the server key K thus we cannot decrypt and recover P_1, P_2
- We can however send C_1, C_2 to the server for decryption
- Note that sending C_1, C_2 is not dangerous: the server will decrypt them and reply with message: `<decryption_ok>`

Attack Idea:

- We will keep block C_2 constant
- We will modify block C_1 in order to to cause a padding error in the server!
- The goal is to recover P_1, P_2

Padding Oracle Attack

- Server workflow with the original ciphertext blocks C_1, C_2



- C_2 gets decrypted producing I_2 and $P_2 = I_2 \oplus C_1$
- The padding is checked returning potential error messages

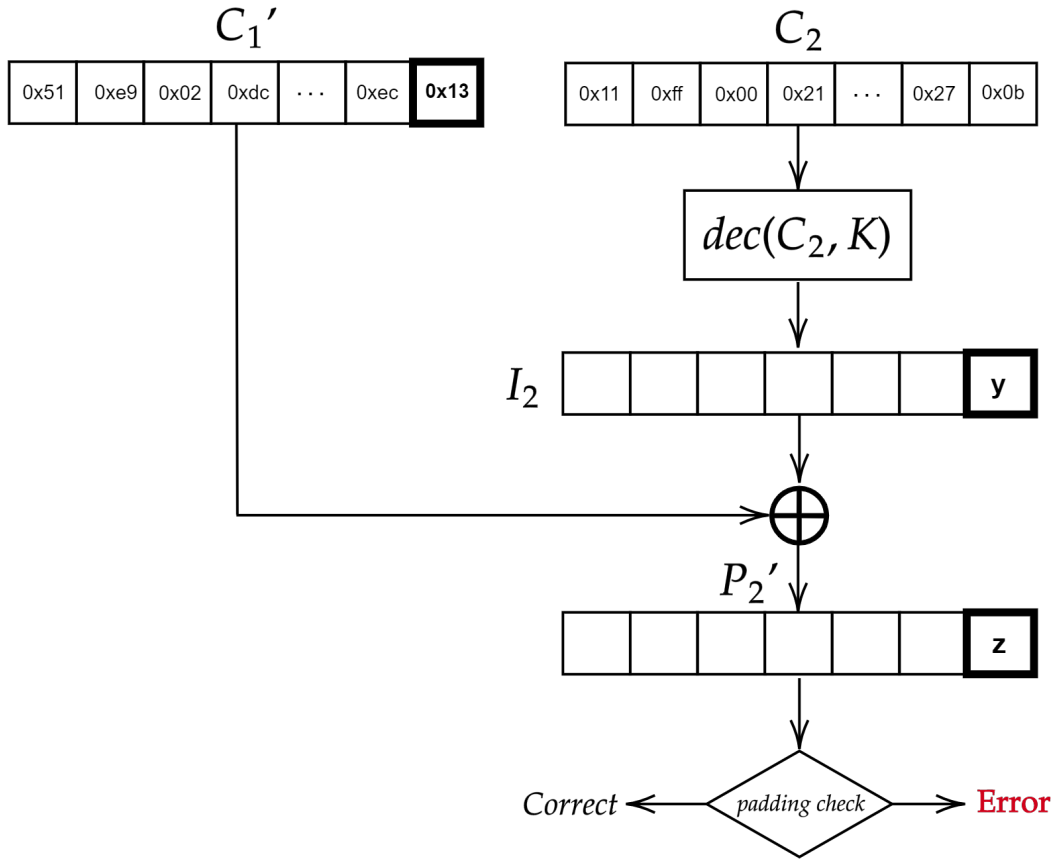
Padding Oracle Attack

- This attack scenario offers oracle access to the server
- The oracle offers a single bit of information about the decryption
  e.g. padding correct `<decryption_ok>` or paddding error `<incorrect_padding>`

$$oracle(IV, C_1, C_2) = \begin{cases} 0, & \text{if there is a padding error} \\ 1, & \text{if there is no padding error} \end{cases}$$

Padding Oracle Attack

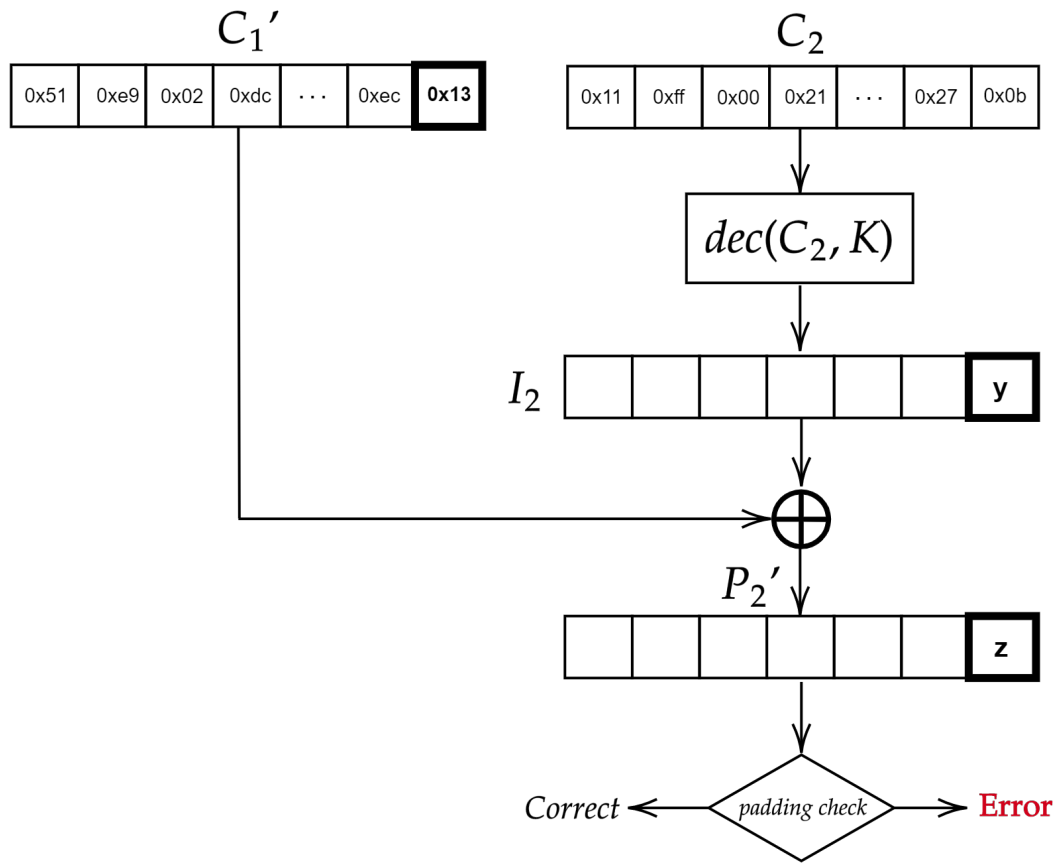
- We now replace C_1 with a random value C_1'



- I_2 remains the constant because C_2 is constant
- P_2 is replaced by P_2' because $P_2' = I_2 \oplus C_1'$

Padding Oracle Attack

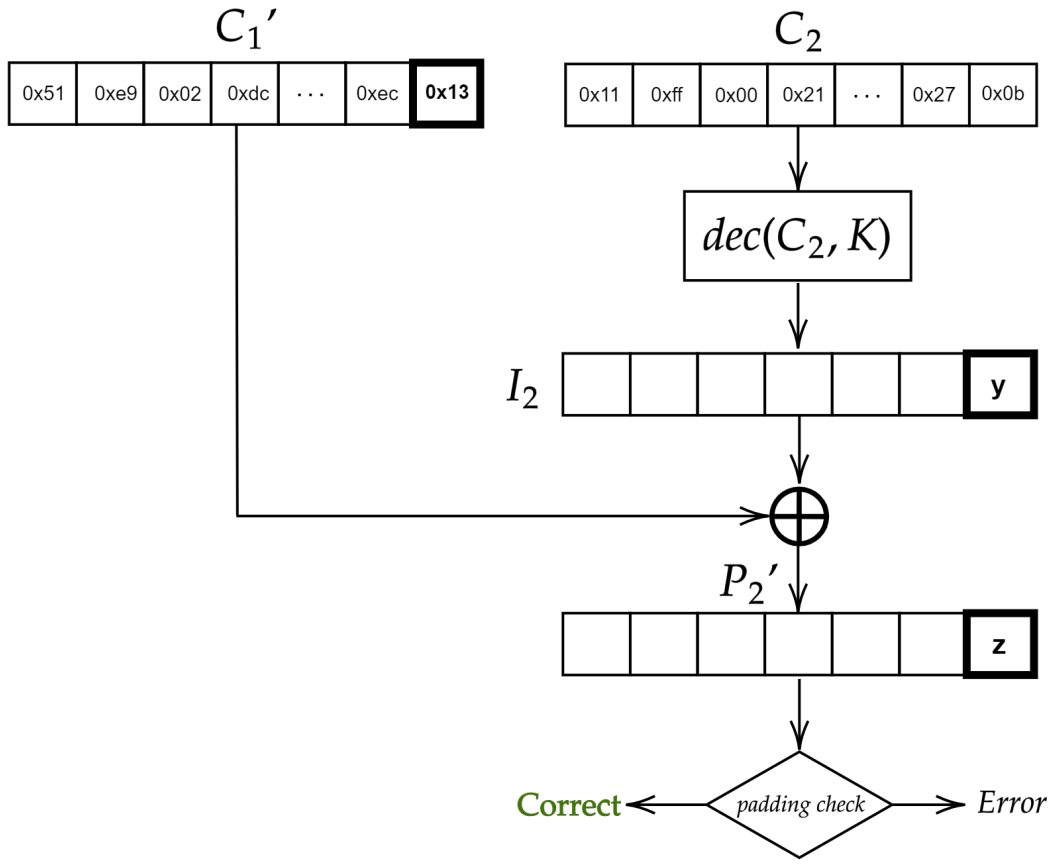
- Can we learn the value of byte z when the oracle returns 0 (error)?



- No! The value z cannot be substantially narrowed down

Padding Oracle Attack

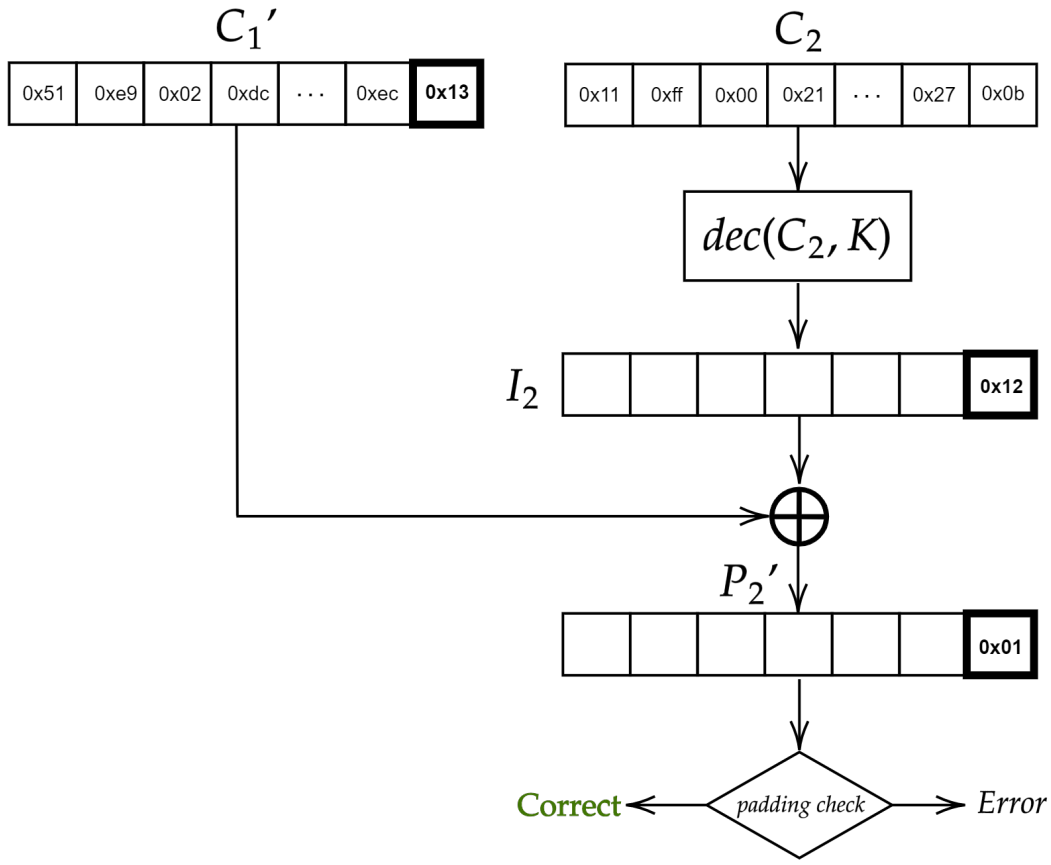
- Can we learn the value of byte z when the oracle returns 1 (correct)?



- Yes! The value z must be part of one of the valid paddings:
  i.e. z =0x01 or z =0x02 or ... or z =0x0f

Padding Oracle Attack

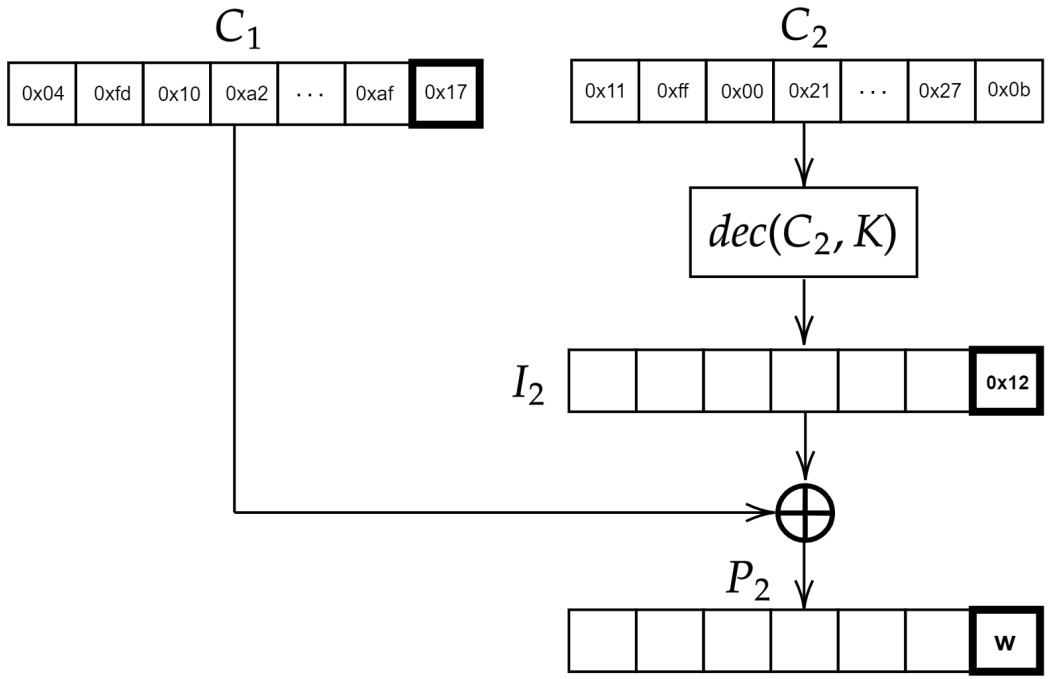
- Let's assume that z =0x01 (single-byte padding)
  This is one of the valid paddings (we will cover the other cases later)



- We can now easily compute the last byte of I_2
  $y = 0x13 \oplus 0x01 = 0x12$

Padding Oracle Attack

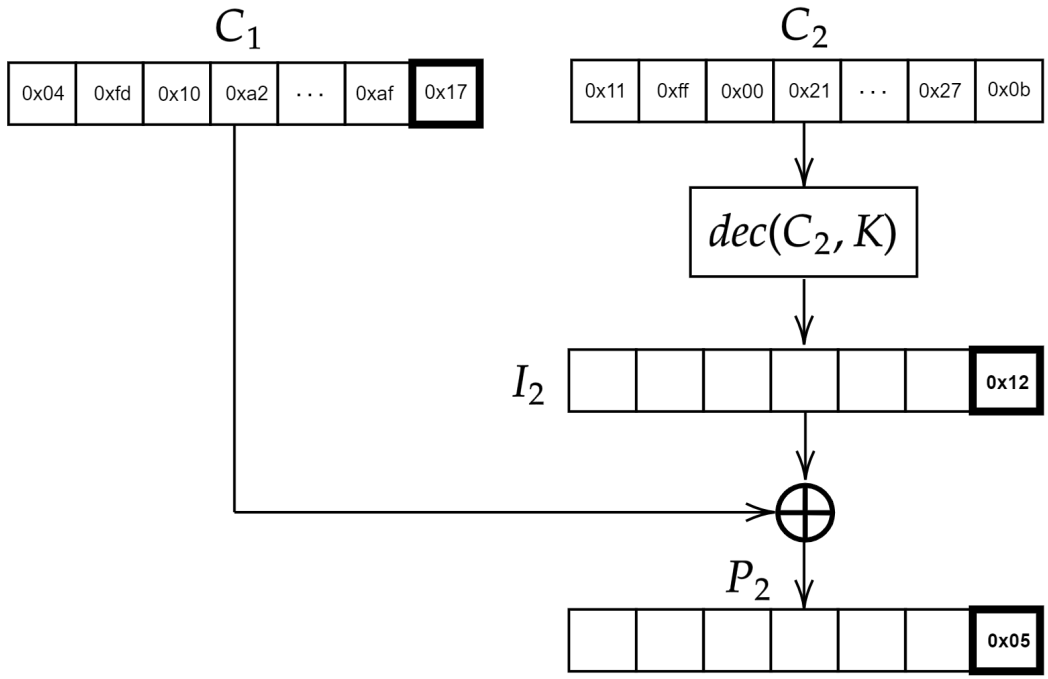
- We now return to the original ciphertext blocks C_1, C_2
  i.e. we no longer use the modified block C_1'



- Notice that we have already found the last byte of I_2
- Notice that I_2 does not depend on C_1 or C_1'
- We can now compute the last byte of P_2
  $w = \texttt{0x17} \oplus \texttt{0x12} = \texttt{0x05}$

Padding Oracle Attack

- We have now recovered the last byte of P_2



Padding Oracle Attack

Last-Byte Oracle:

Input: Ciphertext blocks C_1, C_2 and IV, cipher block size b (in bytes)
We denote with $X(i)$ the ith byte of block X, $i = 1, 2, \ldots, b$
Output: The last byte of plaintext block P_2 i.e. $P_2(b)$

```
1  C'_1 = generate_random_bytes(b) // create a random b-byte block C'_1
2  for x = 0 until 255 do
3  |   C'_1(b) = x   // set the last byte of C'_1 to x
4  |   reply = oracle([IV, C'_1, C_2])   // call the oracle
5  |   if reply == 1 then
6  |   |   x_correct = x   // no padding error
7  |   |   break
8  |   end
9  end
10 I_2(b) = x_correct ⊕ 0x01   // recover the last byte of I_2
11 P_2(b) = I_2(b) ⊕ C_1(b)   // use the original C_1(b) to recover P_2(b)
```

Padding Oracle Attack

Last-Byte Oracle (extra check)

- When the oracle replies with 1, the padding is correct and in line 10 we assumed that the correct padding is `0x01`
- Although `0x01` is the most likely one (due to randomly selecting C_1'), we can also check for case `[0x02 0x02]` or case `[0x03 0x03 0x03]` etc.

Padding Oracle Attack

Last-Byte Oracle (extra check)

- When the oracle replies with 1, the padding is correct and in line 10 we assumed that the correct padding is `0x01`
- Although `0x01` is the most likely one (due to randomly selecting C_1'), we can also check for case `[0x02 0x02]` or case `[0x03 0x03 0x03]` etc.

- We have found that the padding is correct when $C_1' = [C_1'(1), C_1'(2), \ldots, C_1'(b-1), x_{correct}]$
- How do we check that the correct padding is e.g. `[0x02 0x02]`?

Padding Oracle Attack

Last-Byte Oracle (extra check)

- When the oracle replies with 1, the padding is correct and in line 10 we assumed that the correct padding is `0x01`
- Although `0x01` is the most likely one (due to randomly selecting C_1'), we can also check for case `[0x02 0x02]` or case `[0x03 0x03 0x03]` etc.

- We have found that the padding is correct when $C_1' = [C_1'(1), C_1'(2), \ldots, C_1'(b-1), x_{correct}]$
- How do we check that the correct padding is e.g. `[0x02 0x02]`?
- We flip a bit in $C_1'(b-1)$ and send $[IV, C_1', C_2]$ to the oracle
  - If the oracle replies with 1, then altering $C_1'(b-1)$ doesn't affect the padding. Thus the padding was `0x01` after all.
  - If the oracle replies with 0, then $C_1'(b-1)$ affects the padding. Thus the padding can be `[0x02 0x02]` or `[0x03 0x03 0x03]` etc.

Padding Oracle Attack

Last-Byte Oracle (extra check)

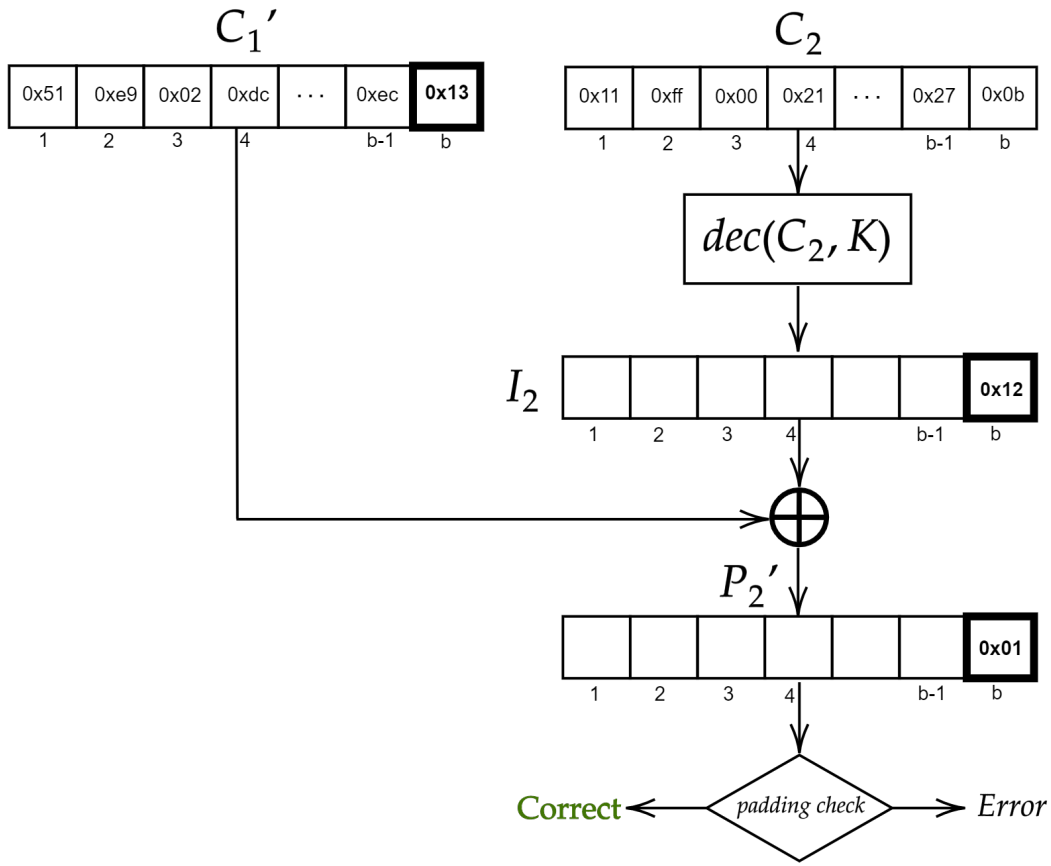
- When the oracle replies with 1, the padding is correct and in line 10 we assumed that the correct padding is `0x01`
- Although `0x01` is the most likely one (due to randomly selecting C_1'), we can also check for case `[0x02 0x02]` or case `[0x03 0x03 0x03]` etc.

- We have found that the padding is correct when $C_1' = [C_1'(1), C_1'(2), \ldots, C_1'(b-1), x_{correct}]$
- How do we check that the correct padding is e.g. `[0x02 0x02]`?
- We flip a bit in $C_1'(b-1)$ and send $[IV, C_1', C_2]$ to the oracle
  - If the oracle replies with 1, then altering $C_1'(b-1)$ doesn't affect the padding. Thus the padding was `0x01` after all.
  - If the oracle replies with 0, then $C_1'(b-1)$ affects the padding. Thus the padding can be `[0x02 0x02]` or `[0x03 0x03 0x03]` etc.
- To exclude the padding `[0x03 0x03 0x03]`, we flip a bit in $C_1'(b-2)$ and check the oracle again
- The process is repeated until the oracle replies with 1 or we reach and check $C_1'(1)$

Padding Oracle Attack

- The Last-Byte Oracle chooses the last byte of C_1' such that the `0x01` padding appears in the last byte of P_2' and then recovers the last byte of I_2

  i.e. it chooses $C_1'(b)$ such
  that $P_2'(b) = \texttt{0x01}$ and recovers $I_2(b)$ where b is the input size of the block cipher

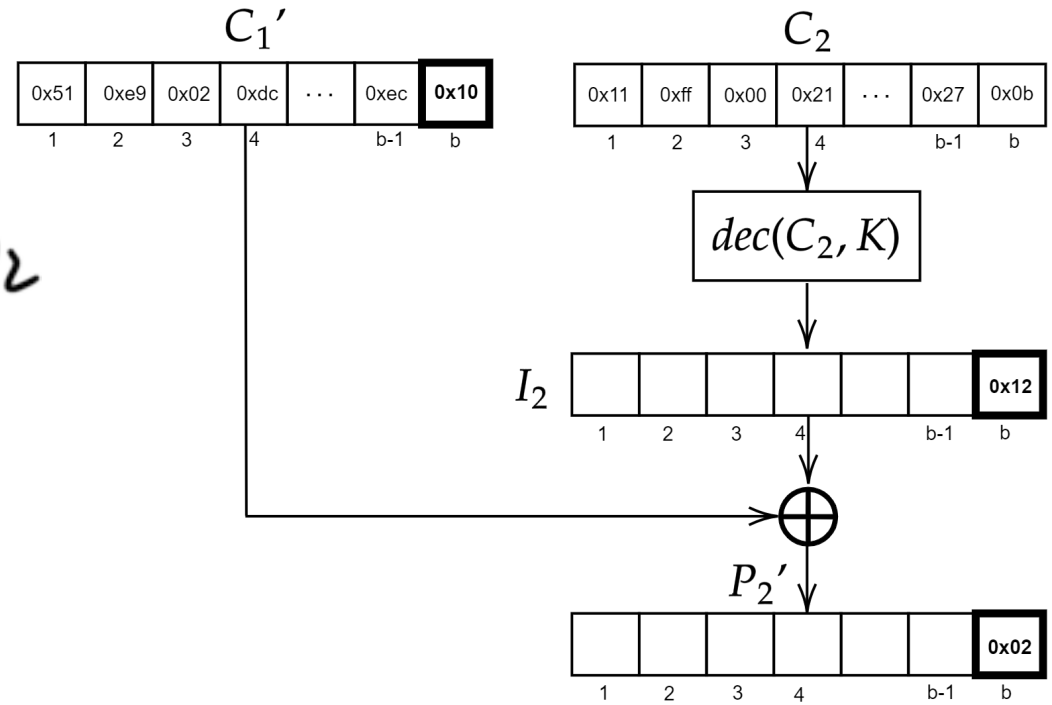


Padding Oracle Attack

- Since we now know $I_2(b)$, we can choose $C_1'(b)$ such that $P_2'(b) = \texttt{0x02}$

$$P_2'(b) = \texttt{0x02} \iff I_2(b) \oplus C_1'(b) = \texttt{0x02} \iff C_1'(b) = \texttt{0x02} \oplus \texttt{0x12} \iff C_1'(b) = \texttt{0x10}$$

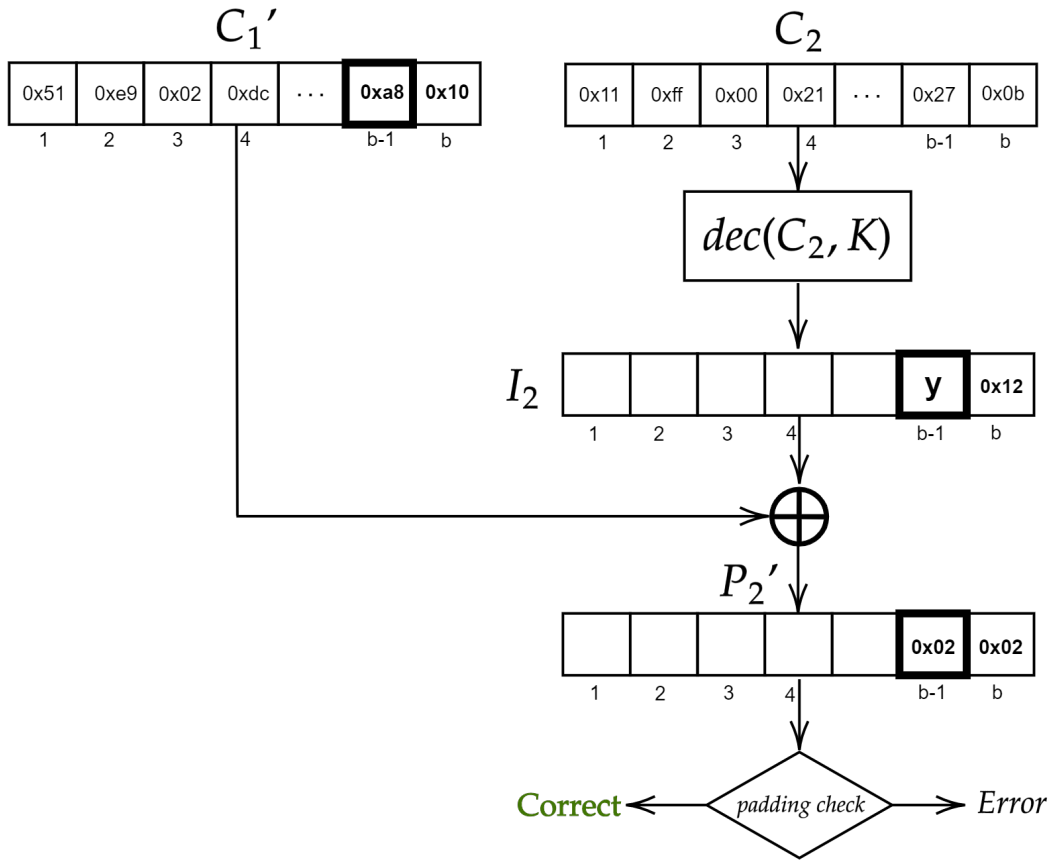
test from making up 0x01 → 0x02 0x02 ... until fault.



Padding Oracle Attack

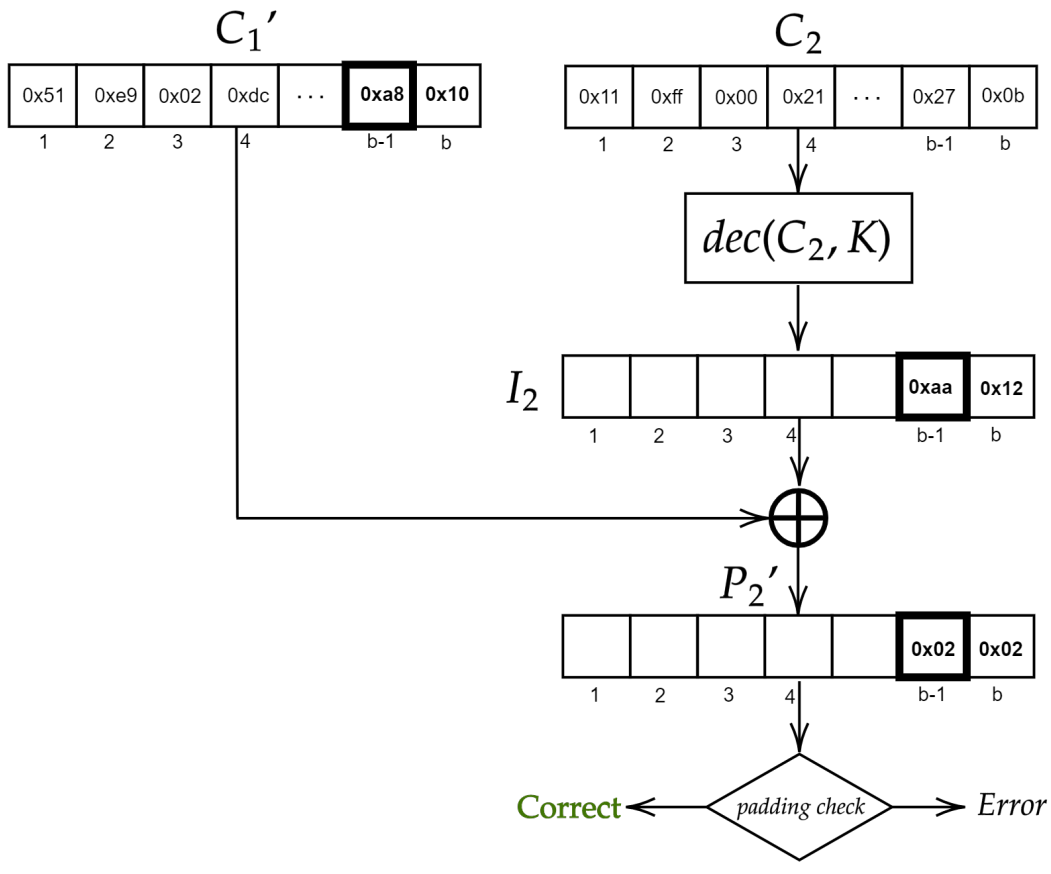
- While $P_2'(b) = \texttt{0x02}$, we will vary $C_1'(b-1)$ until $P_2'(b-1) = \texttt{0x02}$ as well
- The goal is to create the padding [0x02, 0x02] in bytes $[P_2'(b-1),\ P_2'(b)]$

  e.g. the valid padding [0x02, 0x02] is detected for $C_1'(b-1) = \texttt{0xa8}$



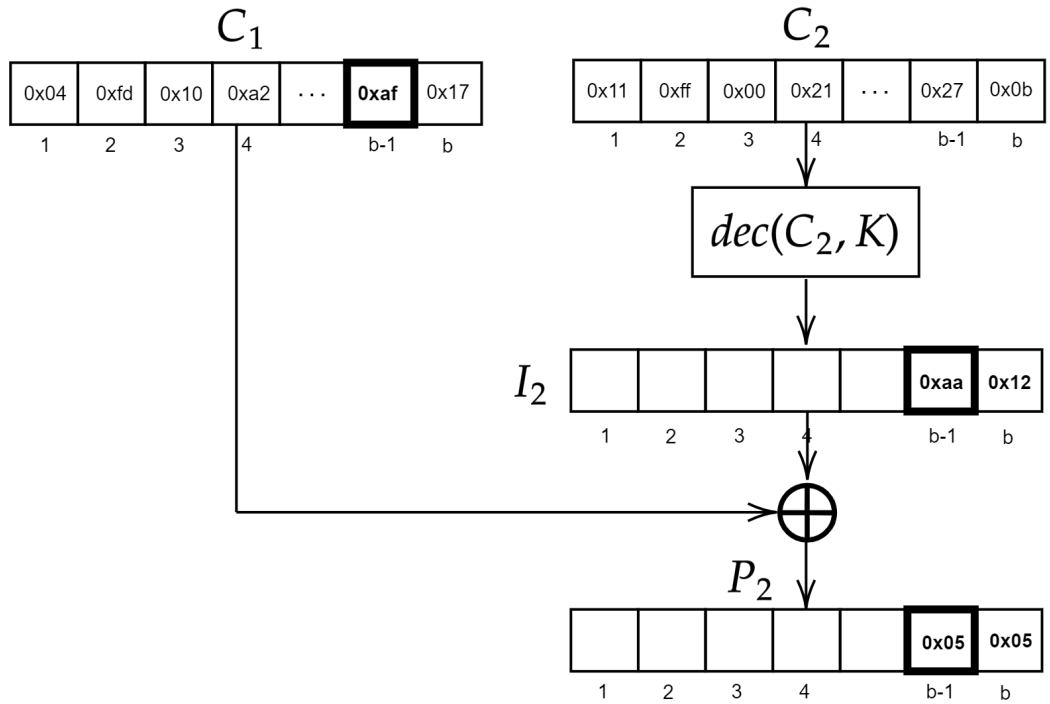
Padding Oracle Attack

- We can now compute $I_2(b-1) = C_1'(b-1) \oplus \texttt{0x02} = \texttt{0xa8} \oplus \texttt{0x02} = \texttt{0xaa}$



Padding Oracle Attack

- Having recovered $I_2(b-1)$, we go back to the original ciphertext block C_1
- We can compute $P_2(b-1) = I_2(b-1) \oplus C_1(b-1) = \texttt{0xaa} \oplus \texttt{0xaf} = \texttt{0x05}$



- We can continue this process by choosing $[C_1'(b-2), C_1'(b-1), C_1'(b)]$ such that $[P_2'(b-2), P_2'(b-1), P_2'(b))] = [\texttt{0x03 0x03 0x03}]$. Then we can compute $I_2(b-2)$ and use $C_1(b-2)$ to recover $P_2(b-2)$.
- Repeating the process yields gradually all bytes of P_2 and we refer to it as the **Last-Block Oracle**.

Padding Oracle Attack

Final notes on the padding oracle attack:

- It is a chosen ciphertext attack that uses the padding check mechanism (and not the padding of the original ciphertext)
- It is possible in symmetric and public key cryptography
- It shows why computer security has a strange reputation
- It can be (partially) prevented by being careful with the error messages
- It can be prevented by stopping unauthorized decryption requests to the server