COL759 Cryptography Semester 1, 2023-24

Problem Set 2

Instructor: Venkata K *Due Date: 27 September 2023*

Solution Collaborators - Lakshay Kumar {2021061} ; Vickey Kumar {2021299}

Instructions:

- Assignment must be done in groups of size at most 2. Each group must submit one pdf on Gradescope, and mention the partner's name (if any).
- The questions are divided into two parts. The first section (Part A.) consists of theoretical/coding questions (32 marks). In the second part, you can either attempt the coding questions (in Part B.1.) or the theoretical question (in **??**). In case both sections are attempted, we will consider the higher score.
- All solutions must be typeset in LaTeX. For the coding questions, provide a brief explanation of your approach and upload the relevant files on Gradescope.
- For the coding questions, you would need to install the `pycryptodome` python3 package to run the given files. Installation instructions can be found on this link.
- Students who are interested in a BTP/MTP in theoretical cryptography are strongly encouraged to attempt the theoretical question (**??**).
- (Optional) Discuss how much time was spent on each problem. This will not be used for evaluation. We will use this for calibrating future assignments.

Notations:

- $\{0,1\}(+)\leq \ell$ denotes the set of all strings of length at most ℓ. For any string $x \in \{0,1\}(+)\ell$ and $i \in \{1,\ldots,\ell\}$, $x[i]$ denotes the $i(+)th$ bit of x.
- $x \mid\mid y$ denotes the concatenation of x and y.

Part A. (32 marks)

(8 marks) **Cryptosystems secure against side-channel attacks**[1]

When we were discussing the padding oracle attack, we also talked about *timing attacks* — attacks that exploit the amount of time required for decryption. As was pointed out by one of the students in class, it is important to consider such attacks for all cryptographic primitives.

Timing attacks are a special case of a broad class of vulnerabilities called *side-channel attacks*. These include attacks based on power analysis, studying the amount of electromagnetic radiation, etc. How do we capture such attacks in a theoretical security game? This is done by allowing the adversary to leak some bounded information about the secret key. In this problem, we define leakage resilience for pseudorandom functions.

Security Game for Leakage resilient PRFs

Let $F : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ be a keyed function.

1. Challenger chooses a bit $b \leftarrow \{0,1\}$ and key $k \leftarrow \mathcal{K}$. If $b = 0$, challenger sets function $F_0 \equiv F(k, \cdot)$. If $b = 1$, challenger chooses a truly random function $F_1 \leftarrow \mathsf{Func}[\mathcal{X}, \mathcal{Y}]$.
2. **Leakage query:** First, the adversary can query for any ℓ bits of the key k. The challenger sends the corresponding bits. [a] Note that even if $b = 1$, the challenger sends these ℓ bits of the key k.
3. **PRF queries:** After receiving the leakage, the adversary can send polynomially many queries (adaptively). For each query x_i, the challenger sends $F_b(x_i)$.
4. $\mathcal{A}$ finally sends a guess b' and wins if $b' = b$.

[a] This leakage represents the information that the adversary learns about the secret key. The general leakage-resilience security game allows the adversary to receive any arbitrary function of the secret key. Here, we are working with a weaker definition where the adversary gets a subset of the key's bits.

Figure 1: ℓ-Leakage Resilient PRF

A keyed function F is said to be an ℓ-leakage resilient PRF if no ppt adversary can win the above security game with non-negligible advantage.

Having defined leakage resilience, a natural question is whether we get leakage resilience *for free*. That is, given any secure PRF, is it already ℓ-leakage resilient for some $\ell > 0$? Unfortunately, no! There exist secure PRFs where leaking even one bit of the key breaks PRF security.

Let $F : \{0,1\}(+)n \times \{0,1\}(+)n \to \{0,1\}(+)n$ be a secure PRF. Construct a new PRF F' with appropriate key space $\mathcal{K}'$, input space $\mathcal{X}'$ and output space $\mathcal{Y}'$ such that the F' is a secure PRF (assuming F is); however, if the adversary learns even one bit of the key, then the PRF is no longer secure.

[1] This problem is taken from the textbook ([?], 4.14). You can refer to the textbook for hints.

(a) Prove (using formal reduction(s)) that if F is a secure PRF, then F' is also a secure PRF.

F is a secure PRF
Key space : $|\mathcal{K}| = n$, Input space : $|\mathcal{X}| = n$, Output space :$|\mathcal{Y}| = n$

Construction of a new PRF F' ::
Key space $|\mathcal{K}'| = n + 1$, Input space $|\mathcal{X}'| = n$, Output space $|\mathcal{Y}'| = n$

$$F'(k', x) = \left\{ \begin{array}{l} F_{k=k'[0:n]}(0^n) \ ; k'[n+1] = b \quad ; \text{if x} = 0^n \\ F_{k=k'[0:n]}(x) \ ; k'[n+1] = \$ \quad ; \text{if x} \neq 0^n \end{array} \right\}$$

$\forall$ x $\neq 0^n$ ciphertext of $F'(k', x)$ is passing $F(k, x)$'s ciphertext. So, F PRF security holds F' PRF security

For x $= 0^n$, $k'[0 : n]$ that is first n bits of k is uniformly randomly sampled and passed to $F(k, 0^n)$ and its ciphertext is passed as value for $F'(k', x)$

Thus $F'(k', x)$ returned ciphertext is $F(k, x)$'s ciphertext $\forall x$,
So if PRF security of F is holded then $F'(k', x)$ is also holded
And, If F is secure then F' is also secure.

(b) Show that F' does not satisfy 1-leakage resilience.
Proof : Attack that break F' is shown below

1. $\mathcal{C}$ randomly samples a key k' in space $\{0, 1\}^{n+1}$ and sample b in $\{0, 1\}$;
1.1 if $b = 0 : k[n + 1] = b$, i.e $n + 1^{th}$ index bit of k' is replaces with sampled b
2. $\mathcal{A}$ selects x $= 0^n$ and l = n+1 i.e. query for key's l^{th} or n+1^{th} bit
3. $\mathcal{C}$ returns $\{F'(k), b_l\}$
Where, $k = k'[0 : n]$ i.e. first n bits of key k' and $b_l = k_{l=n+1}$ i.e. $n + 1^{th}$ bit of k
4. $\mathcal{A}$ return b_l

If $b = b_l$ then $\mathcal{A}$ wins. And when $\mathcal{C}$ query for 0^n the advantage is:

$$AdvF' = |\Pr[\mathcal{A} \text{ sends } 0 \,|\, \mathcal{C} \text{ picks } 0] - \Pr[\mathcal{A} \text{ sends } 0 \,|\, \mathcal{C} \text{ picks } 1]|$$

$Pr[\mathcal{A}$ sends $0 \,|$ Challenger picks 0] equals to 1,
as when pseudorandom function is used then the b_l leaks the correct b = 0 every time.

$Pr[\mathcal{A}$ sends $0 \,|$ Challenger picks 1] equals to $\frac{1}{2}$,
as $n + 1^{th}$ bit is sampled uniform randomly of k', so 50% times b=1, rest incorrectly b=0.

$$AdvF' = |\Pr[\mathcal{A} \text{ sends } 0 \,|\, \mathcal{C} \text{ picks } 0] - \Pr[\mathcal{A} \text{ sends } 0 \,|\, \mathcal{C} \text{ picks } 1]| = \frac{1}{2}$$

Therefore, the $AdvF'$ is non-negligible.

(8 marks) **MACs: unique queries vs non-unique queries**

In the minor, we saw how to convert a MAC with randomized signing into one which has deterministic signing. This also ensures that if an adversary sends repeated queries for the same message, it gets the same response (and therefore repeated queries are useless when the signing algorithm is deterministic). However, is this true in general?

Consider the UFCMA security game for MACs (Figure 18 in the lecture notes), but the adversary is allowed only **unique** signature queries. A MAC scheme is said to be *unforgeable against unique chosen message attacks* (UFCMA-Unique) if, for any ppt adversary $\mathcal{A}$, the advantage in this restricted security game is negligible. This security game is strictly weaker than the regular MAC security game.

Let $F : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ be a secure pseudorandom function. Use F to construct a MAC scheme $\mathcal{I}_{\text{uq}} = (\mathsf{Sign}_{\text{uq}}, \mathsf{Verify}_{\text{uq}})$ with message space $\{0,1\}^n$, and appropriate key space that is UFCMA-Unique secure, but not UFCMA secure.

1. Describe the signing and verification algorithms.

$$\mathcal{I}_{\text{uq}} = \left\{ \begin{array}{l} \mathsf{Sign}_{\text{uq}} = c \longleftarrow \{0,1\}; \left\{ \begin{array}{ll} k \oplus F(k,x); & ; \text{if } c = 0 \\ F(k,x) & ; \text{if } c = 1 \end{array} \right\} \\ \\ \mathsf{Verify}_{\text{uq}} = \left\{ \begin{array}{ll} 1; if \sigma = k \oplus F(k,x) \text{ OR } \sigma = F(k,x) & \\ & \\ 0; & \text{else} \end{array} \right\} \end{array} \right\}$$

2. Show that the MAC scheme $\mathcal{I}_{\text{uq}}$ is UFCMA-Unique secure, assuming F is a secure pseudorandom function. If your security proof requires multiple hybrid-games, describe the games formally, and complete the proof by giving appropriate reductions.

   Show : MAC scheme $\mathcal{I}_{\text{uq}}$ is UFCMA-Unique secure

   It is ensured that all messages are unique,
   i.e. $m_i \neq m_j$, $\forall$ i,j $\leq$ #no_of_query and $i \neq j$.

   $$\{F(k,x)\}, \{k \oplus F(k,x)\} \text{ are secure PRF,}$$

   thus by PRF security guarantee for any two distinct messages, ciphertext will be 'indistinguishable' from each other (winning advantage is non-negligible).

   $SS_{game}\{m_0 = \mathsf{Sign}_{\text{uq}} m_i, m_1 = \mathsf{Sign}_{\text{uq}} m_j\}$
   P_r [$\mathcal{A}$ wins Semantic Security Game w.r.t. $\mathcal{E}$] -$\frac{1}{2} = \mu$ (negligible)

   And, Shannon one time pad ensure PRF's ciphertext for any message, is 'indistinguishable' from uniformly random sampled message

   Therefore, $\mathcal{I}_{\text{uq}}$ is UFCMA-Unique secure.

3. Show that the MAC scheme $\mathcal{I}_{\text{uq}}$ is not UFCMA secure. Construct a ppt adversary $\mathcal{A}$ that wins the UFCMA security game against $\mathcal{I}_{\text{uq}}$ with non-negligible advantage. Analyse the success probability of $\mathcal{A}$.

   Show : MAC scheme $\mathcal{I}_{\text{uq}}$ is not UFCMA secure, as shown by below attack :-

   Sign any message m_1, $p \geq 2$ times till a signature is found different from the first signature.

$$\mathsf{Sign}_{\text{uq}}(m_1) = \sigma_1$$
$$\mathsf{Sign}_{\text{uq}}(m_1) = \sigma_2$$
$$\mathsf{Sign}_{\text{uq}}(m_1) = \sigma_3$$
$$..$$
$$p \text{ times total sign same } m_1$$
$$..$$

   So,

$$k = \sigma_1 \oplus \sigma_p$$

   Now, send a forgery of any message m_j as $\sigma_j = F(k, m)$, which is correct forgery every time, thus advantage is 1.

(2 marks) **A mistake in the lecture notes**

In class, we discussed two definitions for MACs: one with verification queries, and one without. We argued that if there exists a p.p.t. adversary that makes q_v verification queries and wins with probability ϵ, then there exists a p.p.t. adversary that makes no queries and wins with probability at least ϵ/q_v.

Based on one of the in-class suggestions, I had included the following in the notes:

> If every message has a unique signature, then verification queries do not give any additional power to the adversary. For any p.p.t. adversary $\mathcal{A}$ that wins the security game in Figure 19 with probability ϵ using q_s signing queries and q_v verification queries, there exists a p.p.t. reduction $\mathcal{B}$ that wins the security game in Figure 18 with probability ϵ using $q_s + q_v$ signing queries. On receiving a verification query (m, σ), the reduction algorithm $\mathcal{B}$ sends m to the challenger, and receives a signature σ'. If $\sigma = \sigma'$, then it sends 1, else it sends 0.

In the above (informal and flawed) argument, we did not use the fact that $\mathcal{A}$ is p.p.t. As a result, using the same argument, we can state that for any adversary $\mathcal{A}$ that runs in time t, makes q_s signing queries and q_v verification queries and wins with probability ϵ, there exists a reduction that makes at most $q_s + q_v$ signing queries, runs in time $t + \mathsf{poly}(\lambda) \cdot (q_s + q_v)$ and wins with probability ϵ.

Show that this argument is incorrect. Let $\mathcal{I}$ be a MAC scheme where each message has a unique valid signature. Propose an adversary that runs in time t, makes q_s signing queries, q_v verification queries and wins with probability 1 (note that here we did not assume t, q_s or q_v are polynomial). However, the above reduction does not win with probability 1.

Show : $\mathcal{I}$ is a MAC scheme where each message has a unique valid signature.
Without the assumption of t, q_s or q_v being polynomial we can brute force our way out to the solution, and the property that all messages have unique signatures, we know all the possible input and output space.

We just have to query, except one (let say m_i) all the messages in input space $\mathcal{M}$
All the signatures of messages are stored by $\mathcal{A}$; $|\mathcal{M}| - 1$ signatures stored.
At this point, only one viable datum/signature is left in output space (let say σ_i as only a subset of output space can ever be imaged to by the inputs pace (unique signature of all message property)

Thus, Send a forgery of (m_i, σ_i)
Forgery will be correct or in other words, Winning probability is 1

{If one, query less messages(more than polynomial many) in input space, we can decrease the number of viable options in output space, still the Winning probability would be $\frac{1}{2}$ + non-negligible.}

We couldn't submit our coding questions (in late day subm) and raisied same query, and Prof allowed us to submit our pseudocode for the same;
Please grade us accordingly, Thankyou :)

4. Even-Mansour instantiated with a bad permutation (5 marks) Even-Mansour cipher has been proved to be secure in the *ideal permutation model*. However, using a "bad" permutation may lead to attacks in the real world. In this question, your goal is to understand the given implementation, find vulnerabilities based on the public permutation being used and code an attack to break PRP security of the scheme using this permutation. Hopefully, this question successfully conveys that security in the ideal permutation model does not translate directly to security in the real world and that one needs to be rather careful while implementing these in p

Correctness of the below algorithm using slidex attack proof

- The slidex attack works by exploiting the following property of the Even-Mansour cipher:

  E(P) $\oplus$F(P) = E(P*) $\oplus$ F(P*) if and only if P $\oplus$ P^*= K_1

- Where E and F are the two permutations that are used to construct the Even-Mansour cipher, and K1 is the first key of the Even-Mansour cipher.
- The slidex attack works by first finding a pair of plaintexts P and P* such that $P \oplus P^* = K_1$. This can be done by calculating the XOR of each pair of known plaintexts and sorting the XOR values in ascending order. If a collision is found, then the two plaintexts that correspond to the collision satisfy the condition P$\oplus$ $P* = K_1$.
- Once the attacker has found a pair of plaintexts P and P* such that P$\oplus P* = K_1$, the attacker can query the oracle on the two plaintexts. If the oracle outputs the same value for both plaintexts, then the attacker knows that the bit b=0. Otherwise, the attacker knows that the bit b=1.

Algorithm 1 The pseudocode of the slidex attack function

- Here is the pseudocode of the slidex attack function:

  def attack(oracle, pi_func, p):
  """Implements the slidex attack on the attack.py file.

  Args:
  oracle: A function that takes an input number x between 0 to p-1 and returns the output on x.
  pi_func: A function that takes an input number x between 0 to p-1 and returns $\pi(x)$
  p: A large prime number

  Returns: A tuple (b, k1, k2), where b is the flag that was sampled by the challenger, k1 is the first key of the Even-Mansour cipher, and k2 is the second key of the Even-Mansour cipher.
  """

- Generate a list of 2(n+1)/2 known plaintexts. plaintexts = random.sample(range(0, p), 2 * (p // 2 + 1))

  For each pair of plaintexts, calculate the XOR of the plaintexts. xor_values = [plaintext1 $\oplus$ plaintext2 for plaintext1, plaintext2 in zip(plaintexts[:-1], plaintexts[1:])]

- Sort the XOR values in ascending order.
  xor_values.sort()

- For each collision in the sorted XOR values, check if the corresponding plaintexts satisfy the condition $P \oplus P^* = K1$. If they do, then we have found a pair of plaintexts that can be used to distinguish between the PRP and a random permutation.

  for i in range(len(xor_values) - 1):
  if xor_values[i] == xor_values[$i+1$] :
  $k1 = \text{plaintexts}[i] \oplus \text{plaintexts}[i+1]$
  Query the oracle on the two plaintexts and check if $E(P) \oplus F(P) = E(P^*) \oplus F(P^*)$.
  if oracle$(0, k1)$ == oracle$(1, k1)$:
  return$(0, k1, \text{None})$
  The attack has failed.
  return (None, None, None)[b is the first element of result]

5. 3-round Luby-Rackoff with inversion queries (4 marks)
Pseudocode: (`COL759_A2_Coding2.zip`)

Algorithm 2 Predict Function

- attack(permute, inverse_permute):
  Generate x1 and x2 as 16 bytes of 0
  a1 = 16 bytes of 0
  a2 = 16 bytes of 0

- Concatenate a1 and a2 to form y
  y = concatenate(a1, a2)

- Query the decryption oracle for decryptions of 0 and 0
  D_1= inverse_permute(y) x1 = D_1[:16]
  x2 = D_1[16:]

- Query the encryption oracle for 0 and first 16 bytes of D_1
  E_1 = permute(concatenate(a1,x1))

- Extract y1 and y2 from the E_1
  y1 = E_1[:16]
  y2 = E_1[16:]

- Query the decryption oracle for (x2⊕y2 and y1)
  D_2 =inverse_permute(concatenate(x2 ⊕ y2,y1))
  x3 and x4 are the first and last 16 bytes of D_2 respectively
  X3 = D_2[:16]
  x4 = D_2[16:]

- Check if x3 = y1 ⊕ x1 and return 1 otherwise 0

proof of correction of algorithm:

- Additionally, a short intuitive proof is provided below, as to how the 3-round construction fails when the adversary is allowed access to a decryption oracle. Consider the following adversary A:

1. Query the decryption oracle for the decryptions of $0, 0$:
$\mathrm{D}(0||0) \rightarrow (x_1, x_2)$
2. Query the encryption oracle for $0, x_1$:
$\mathrm{E}(0||x_1) \rightarrow (y_1, y_2)$
3. Query the decryption oracle for $(x_2 \oplus y_2, y_1)$:
$\mathrm{D}(x_2 \oplus y_2||y_1) \rightarrow (x_3, x_4)$
4. If $x_3 = y_1 \oplus x_1$ output 1,
else output 0

Claim. Adversary A can distinguish between a 3-round Feistel construction and a pseudorandom permutation with non-negligible probability.

- From the definition of the Feistel construction, we have the following values:

$$\begin{aligned} x_1 &= F_{k_2}(F_{k_3}(0^n)) \\ x_2 &= F_{k_3}(0) \oplus F_{k_1}(x_1) \\ y_1 &= x_1 \oplus F_{k2}(F_{k_1}(x_1)) \\ y_2 &= F_{k1}(x_1) \oplus F_{k_3}(y_1) \end{aligned}$$

- From the above, it's easy to see that:

$$x_2 \oplus y_2 = F_{k_3}(0) \oplus F_{k_3}(y_1)$$

So at the last carefully selected query, we get:

$$x_3 = y_1 \oplus F_{k2}(x_2 \oplus y_2) \oplus F_{k_3}(y_1) = y_1 \oplus F_{k_2}(F_{k_3}(0^n)) = y_1 \oplus x_1$$

This is exactly as expected and is not a behavior that a pseudorandom permutation would exhibit. Thus, an adversary can easily distinguish between this construction and a pseudorandom permutation if allowed access to a decryption oracle.

6. CBC mode with bad initialization (5 marks)
Pseudocode: (`COL759_A2_Coding3.zip`)

- `attack.py`:

  we did in gradescope ,so evaluate according to that file

  - // Create a new ciphertext by concatenating the first 16 bytes, 16 zeros, and the first 16 bytes
    new_ct = concatenate(ciphertext[0:16], [0] * 16, ciphertext[0:16])
  - // Decrypt the new ciphertext using the decryption function
    message = decrypt(new_ct)
  - // Extract the first 16 bytes as the start block and the last 16 bytes as the last block start = message[0:16]
    last = message[-16:]
  - // XOR each byte of the start block with the corresponding byte of the last block
    result = []
    for i = 0 to 15:
    result[i] = start[i] XOR last[i]
  - // Convert the result to bytes
    result_bytes = convert_to_bytes(result)
  - // Return the result
    return result_bytes

Part B. Coding/Theoretical Problems (8 marks)

Part B.1. Coding Problem: Padding Oracle Attack

Consider the following encryption scheme:

- $\mathsf{Enc}(k, m)$: The message m is an arbitrary sequence of bytes. Here the key k is a 16 byte string. We use AES in CBC mode to encrypt this message. Since AES can only handle messages whose length (in bits) is a multiple of 128, we have to pad m appropriately.

  Padding Scheme: Instead of padding at the right-most end, we would instead pad at the left-most end (as suggested by one of the students in class). Let p be the number of bytes to be padded – then include the number p (in binary) in each of the p bytes.

  Examples:

  - m = (11 42 33 01 89 12). This message is 6 bytes long. We need to pad it with 10 bytes. The resulting message m' would be

    $$m' = (10\ 10\ 10\ 10\ 10\ 10\ 10\ 10\ 10\ 10\ 11\ 42\ 33\ 01\ 89\ 12).$$

  - m = (02 02 00 00 00 00 00 00 00 00 00 00 00 00 00 16). This message is 16 bytes long. Add a new padding block to avoid ambiguity. The padded message

    $$m' = (16\ 16\ \ldots 16\ 16\ 02\ 02\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 16).$$

- $\mathsf{Dec}(\mathsf{ct}, k)$: Decryption algorithm simply decrypts the ciphertext to obtain the padded message $m' = (y_1, y_2, ..., y_\ell)$ where each y_i is 16 bytes long. It checks if y_1 is a valid padded string. That is, check that the first byte of y_1 is a number between 1 and 16. If the number is z, then check that the next $z - 1$ bytes after it have the value z. If any of these is violated, output "Error: Bad Padding". Otherwise, output the decrypted string (without the padding).

Files Given: Since the messages and the cipher-texts are arbitrary sequences of bytes, we represent each of them by a list of integers within the range [0, 255]. You are given the following `python` files on MS Teams (`A2_Coding_Students.zip`):

- `encrypt.py`:
  - This file has a 16-bit key hardcoded into it for the AES scheme
  - It has a function called `encrypt(message)`, which takes a list of integers as input, pads it appropriately and returns the encrypted bytes
  - This script can be used to generate cipher-texts with the given key. You can use it to check the correctness of your code.
- `decrypt.py`:
  - This file has the same key hardcoded for AES as in `encrypt.py`

- It has a function called `check_padding(encd)`, which takes a ciphertext as input (in the form of an integer list), decrypts it and checks for a valid padding.
- It returns 0 if it was a valid padding, else returns 2. It does NOT return the decrypted message

- `attack.py`:
  - Implement function `attack(cipher_text)` taking a ciphertext as input, and the return the original message (as a list of integers)
  - You are allowed to make calls to `check_padding()` from `decrypt.py`

Listing 1: Pseudo-Code for the 'attack' Function

```
from decrypt import check_padding

def attack(cipher_text):
    """
    Takes ciphertext(list of int from 0 to 255 [byte array]) as input
    Return a list of integers representing the original message
    """
    encd = bytes(cipher_text)
    message = [0]*16
    i_vec, data = encd[:16], encd[16:]

    # Finding Padding length
    pad_length = 1
    for i in range(15,0,-1):
        _i_vec = bytes([i(+)j for j in i_vec])
        if check_padding(list(_i_vec+data)) == 2:
            pad_length=i
            break
    # Finding plaintext using Padding length
    for pi in range(0,16):
        for m in range(255,0,-1):
            a = bytes([message[j](+)i_vec[j](+)(pi+1) for j in range(pi)
            b = bytes([i_vec[pi](+)m(+)(pi+1)])
            c = bytes(i_vec[pi+1:])
            _i_vec =  a+b+c

            if check_padding(list(_i_vec+data)) == 0:
                message[pi] = m
                break
    plaintext = bytes(message[pad_length:])
    return plaintext
```

The `attack` function step by step:

1. `encd = bytes(cipher_text)`:
    - This line converts the input `cipher_text`, which is a list of integers representing bytes, into a `bytes` object called `encd`. This is a common step to work with binary data in Python.
2. `message = [0]*16`:
    - An empty list called `message` is initialized with 16 zeros. This list will be used to store the bytes of the decrypted plaintext as they are recovered.
3. `i_vec, data = encd[:16], encd[16:]`:
    - The `encd` variable is divided into two parts: `i_vec` and `data`. `i_vec` represents the first 16 bytes of the ciphertext, which is typically referred to as the Initialization Vector (IV). `data` contains the remaining ciphertext bytes.
4. `pad_length = 1`:
    - The `pad_length` variable is initialized to 1. It will be used to keep track of the padding length, which is necessary for decrypting the ciphertext.
5. `for i in range(15, 0, -1)`:
    - This loop iterates in reverse from 15 down to 1, as it's trying to find the length of the padding (the number of bytes added to pad the plaintext to a multiple of the block size).
6. `_i_vec = bytes([i(+)j for j in i_vec])`:
    - Inside the loop, a modified `_i_vec` is calculated by XORing each byte in `i_vec` (the IV) with the current iteration index `i`. This simulates removing potential padding bytes.
7. `if check_padding(list(_i_vec+data)) == 2`:
    - The `_i_vec` is concatenated with the `data` portion of the ciphertext, and the `check_padding` function is called to check if the padding is valid. A return value of 2 indicates valid padding.
8. `pad_length=i`:
    - If valid padding is detected, the `pad_length` is updated to the current iteration index `i`, which represents the length of the padding.
9. `break`:
    - The loop is exited since the correct padding length has been found, and there's no need to continue checking.

10. `for pi in range(0, 16)`:
    - This loop iterates over each position `pi` in the `message`, representing the byte positions of the decrypted plaintext.
11. `for m in range(255, 0, -1)`:
    - Nested within the outer loop, this inner loop iterates over byte values `m` in reverse order from 255 down to 1.
12. `a = bytes([message[j](+)i_vec[j](+)(pi+1) for j in range(pi)])`:
    - A component `a` is calculated by XORing specific bytes of the `message`, `i_vec`, and `(pi+1)`.
13. `b = bytes([i_vec[pi](+)m(+)(pi+1)])`:
    - Another component `b` is calculated by XORing a byte from `i_vec`, byte `m`, and `(pi+1)`.
14. `c = bytes(i_vec[pi+1:])`:
    - A component `c` is extracted from `i_vec` starting from the `(pi+1)`-th byte.
15. `_i_vec = a + b + c`:
    - `_i_vec` is formed by concatenating `a`, `b`, and `c`. This represents a modified IV with guessed values.
16. `if check_padding(list(_i_vec+data)) == 0`:
    - The `_i_vec` is concatenated with the `data` portion of the ciphertext, and the `check_padding` function is called to check if the padding is now valid (return value of 0). This indicates a successful guess for the byte at position `pi`.
17. `message[pi] = m`:
    - If the padding is valid, it means that the guessed byte `m` at position `pi` is correct, so it is assigned to the `message` list at that position.
18. `plaintext = bytes(message[pad_length:])`:
    - After recovering all the bytes of the plaintext, this line extracts the valid decrypted portion of the `message` list starting from `pad_length`.
19. `return plaintext`:
    - The function returns the recovered plaintext as a `bytes` object.

Overall, the `attack` function attempts to recover the original plaintext from the ciphertext by iteratively guessing bytes of the plaintext, starting with the padding length and working backward through the message, using a padding oracle to validate each guess. This process continues until all the bytes of the plaintext have been recovered.