



PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE
ESCUELA DE INGENIERIA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACION

Criptografía y Seguridad Computacional - IIC3253

Rúbrica Tarea 2

Preguntas

1. En esta pregunta deberá implementar la construcción de Merkle-Damgård de funciones de hash para mensajes de largo arbitrario, utilizando la construcción de Davies-Meyer para funciones de compresión. Concretamente, deberá escribir un Jupyter notebook de acuerdo a las instrucciones explicadas arriba que contenga las siguientes funciones:

```
def davies_meyer(encrypt: (bytearray, bytearray) -> bytearray,
                 l_key: int, l_message: int) -> (bytearray) -> bytearray:
    """
    Arguments:
        encrypt: an encryption function
        l_key: length in bytes of the keys for encrypt
        l_message: length in bytes of the messages for encrypt
    Returns:
        A compression function from messages of length l_key + l_message to
        messages of length l_message, defined by using the Davies-Meyer
        construction
    """
```

```
def pad(message: bytearray, l_block: int) -> bytearray:
    """
    Arguments:
        message: message to be padded
        l_block: length in bytes of the block
    Returns:
        extension of message that includes the length of message
        (in bytes) in its last block
    """
```

```
def merkle_damgard(IV: bytearray, comp: (bytearray) -> bytearray,
                  l_block: int) -> (bytearray) -> bytearray:
    """
    Arguments:
        IV: initialization vector for a hash function
        comp: compression function to be used in the Merkle-Damgard
        construction
        l_block: length in bytes of the blocks to be used in the Merkle-Damgard
```

```

    construction
Returns:
    A hash function for messages of arbitrary length, defined by using
    the Merkle-Damgard construction
"""

```

Como un ejemplo de cómo pueden ser utilizadas estas funciones considere el siguiente código, donde AES_128 es una función que implementa el algoritmo de cifrado AES con llaves y mensajes de 16 bytes (128 bits).

```

if __name__ == "__main__":
    compresion = davies_meyer(AES_128, 16, 16)
    hash = merkle_damgard(bytearray(b'0123456789012345'), compresion, 16)
    s1 = bytearray(b'Este es un mensaje de prueba para la tarea 2')
    s2 = bytearray(b'Este es un segundo mensaje de prueba para la tarea 2')
    h1 = hash(s1)
    h2 = hash(s2)
    print(h1)
    print(h2)

```

Corrección. Para corregir esta pregunta se utilizará la siguiente implementación de AES_128

```

from Crypto.Cipher import AES

def AES_128(key: bytearray, message: bytearray) -> bytearray:
    a = AES.new(bytes(key), AES.MODE_ECB)
    return bytearray(a.encrypt(bytes(message)))

```

y se realizarán los siguientes tests:

- 4 tests para verificar su implementación de la función `davies_meyer`, cada uno con un puntaje de 0.25,
- 4 tests para verificar su implementación de la función `pad`, cada uno con un puntaje de 0.5, y
- 4 tests para verificar su implementación de la función `merkle_damgard`, cada uno con un puntaje de 0.75.

2. **Disclaimer.** Los algoritmos que deberá implementar en esta pregunta tienen fines exclusivamente académicos y **no deben ser utilizados en la práctica**. No es recomendado encriptar repetidas veces utilizando la misma llave cuando se utiliza criptografía asimétrica. La criptografía asimétrica se usa en la práctica para intercambiar llaves simétricas y producir firmas digitales. Al encriptar un mensaje usando criptografía asimétrica (por ejemplo, para intercambiar una llave simétrica) se recomienda siempre utilizar versiones aleatorizadas y estandarizadas como lo propuesto en PKCS #1.

Enunciado. En esta pregunta deberá programar dos clases que interactúan entre ellas para comunicarse utilizando el protocolo RSA visto en clases. Concretamente, deberá escribir un Jupyter notebook de acuerdo a las instrucciones explicadas arriba que contenga:

- Una clase que represente a quien recibe los mensajes. Esta clase debe permitir generar un par de llaves, entregar la llave pública, y decryptar mensajes.
- Una clase que represente a quien envía los mensajes. Para inicializar un objeto de esta clase se debe entregar como parámetro una llave pública con la que luego se debe poder encriptar mensajes.

Las firmas de estas clases se deben ver de la siguiente forma:

```
class RSARceiver:

    def __init__(self, bit_len: int) -> None:
        """
        Arguments:
            bit_len: A lower bound for the number of bits of N,
                    the second argument of the public and secret key.
        """

    def get_public_key(self) -> bytearray:
        """
        Returns:
            public_key: Public key expressed as a Python 'bytearray' using the
                        PEM format. This means the public key is divided in:
                        (1) The number of bytes of e (4 bytes)
                        (2) the number e (as many bytes as indicated in (1))
                        (3) The number of bytes of N (4 bytes)
                        (4) the number N (as many bytes as indicated in (3))
        """

    def decrypt(self, ciphertext: bytearray) -> str:
        """
        Arguments:
            ciphertext: The ciphertext to decrypt
        Returns:
            message: The original message
        """
```

```
class RSASender:

    def __init__(self, public_key: bytearray) -> None:
        """
        Arguments:
            public_key: The public key that will be used to encrypt messages
        """

    def encrypt(self, message: str) -> bytearray:
        """
        Arguments:
            message: The plaintext message to encrypt
        Returns:
            ciphertext: The encrypted message
        """
```

Puede definir funciones adicionales antes de definir las clases. También puede definir otros

métodos dentro de estas clases. El nombre de todos los métodos y todas las funciones adicionales debe comenzar con un guión bajo (_).

Detalles de implementación

- Para pasar un mensaje a bytes necesitamos usar alguna codificación particular. Utilizaremos la codificación UTF-8, lo que significa que para pasar un string `s` a un objeto de tipo `bytearray` podemos usar `bytearray(s, 'utf-8')`.
- Para encriptar el `bytearray` proveniente de un mensaje, lo separaremos en bloques de n bytes, donde n es el mayor múltiplo de 8 tal que $8 \cdot n$ es menor que el número de bits de N . Por ejemplo, si para representar N se necesitan 2056 bits, entonces n será 256, dado que $256 \cdot 8 = 2048$ y 2048 es el mayor múltiplo de 8 estrictamente menor que 2056.
- Para encriptar un bloque b (que es un `bytearray` de tamaño n), primero pasaremos dicho `bytearray` a un número a usando `int.from_bytes(b, 'big')`. Luego encriptaremos dicho número como $c = a^e \bmod N$. Finalmente, guardaremos c en un `bytearray` de largo $n + 1$, dado que c podría no caber en n bytes.
- La cantidad de bytes del mensaje no tiene por qué dividir al número n . En caso de que no lo divida simplemente utilizaremos un último bloque más corto. Si el mensaje tiene ℓ bytes entonces el último bloque tendrá $\ell \bmod n$ bytes.
- Supondremos que el byte 0 nunca vendrá codificado en el string de entrada, lo que resulta útil para decriptar el último bloque.

Corrección. Para corregir esta pregunta se crearon 15 ejemplos para encriptar y 15 ejemplos para decriptar. Cada ejemplo encriptado/decriptado correctamente suma un punto. Si al decriptar sólo el último bloque es incorrecto debido a no considerar que el byte `0x00` no podía ocurrir en el texto, entonces dicho ejemplo suma 0.9 puntos en vez de 1 punto. Finalmente la nota de esta pregunta se calcula como

$$\text{Nota} = 1 + \frac{\text{puntaje}}{5}$$

La solución a esta pregunta junto al código utilizado para generar los ejemplos de corrección se encuentran en el archivo `/Tareas/Tarea 2/soluciones/p2/T2P2.ipynb`. Los ejemplos particulares utilizados en la corrección están en `/Tareas/Tarea 2/soluciones/p2/grading.json`.