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

Criptografía y Seguridad Computacional - IIC3253
Tarea 1
Plazo de entrega: 4 de mayo

Instrucciones

Cualquier duda sobre la tarea se deberá hacer en los *issues* del repositorio del curso. Si quiere usar alguna librería en sus soluciones debe preguntar primero si dicha librería está permitida. El foro es el canal de comunicación oficial para todas las tareas.

Entrega. Para entregar esta primera tarea usted deberá crear un repositorio en GitHub. Deberá utilizar dicho repositorio para entregar todas las tareas de este curso. Al entregar esta primera tarea, la estructura de archivos de su repositorio se deberá ver exactamente de la siguiente forma:

```
Repositorio
├── Tarea 1
│   ├── requirements.txt
│   ├── Pregunta 1
│   │   └── pregunta1.pdf
│   ├── Pregunta 2
│   │   └── pregunta2.ipynb
│   ├── Pregunta 3
│   │   └── pregunta3.ipynb
│   └── Pregunta 4
│       └── pregunta4.pdf
├── .gitignore
└── README.md
```

Además, deberá considerar lo siguiente:

- El archivo `requirements.txt` deberá especificar todas las librerías que se necesitan instalar para ejecutar todas las respuestas en código de su tarea. Este archivo debe seguir la especificación de Pip, es decir se debe poder ejecutar `pip install -r requirements.txt` suponiendo una versión de `Pip` mayor o igual a 20.0 que apunta a la versión 3.9 de Python.

- La solución de cada problema de programación debe ser entregada como un Jupyter Notebook (esto es, un archivo con extensión `ipynb`). Este archivo debe contener comentarios que expliquen claramente el razonamiento tras la solución del problema, idealmente utilizando *markdown*. Más aun, su archivo deberá ser exportable a un módulo de python utilizando el comando de consola

  ```
  jupyter-nbconvert --to python preguntaX.ipynb
  ```

  Este comando generará un archivo `preguntaX.py`, del cual se deben poder importar las funciones que se piden en cada pregunta. Por ejemplo, para la Pregunta 2, luego de ejecutar este comando, se debe poder importar desde otro archivo python (ubicado en el mismo directorio) la función `custom_md5` simplemente agregando la linea `from pregunta2 import custom_md5`.

- Para cada problema cuya solución se deba entregar como un documento (en este caso las preguntas 1 y 4), usted deberá entregar un archivo `.pdf` que, o bien fue construido utilizando LaTeX, o bien es el resultado de digitalizar un documento escrito a mano. En caso de optar por esta última opción, queda bajo su responsabilidad la legibilidad del documento. Respuestas que no puedan interpretar de forma razonable los ayudantes y profesores, ya sea por la caligrafía o la calidad de la digitalización, serán evaluadas con la nota mínima.

Preguntas

1. En clases se definió la noción de *perfect secrecy* bajo la intuición de que, si un atacante ve sólo un texto cifrado, para él todo par de textos planos debiesen ser equiprobables. Formalmente, si tenemos un sistema criptográfico cuya función de encriptación es *Enc* sobre espacios de llaves, mensajes, y textos cifrados M, K y C, respectivamente, entonces:

   $$\forall c_0 \in C, \forall m_1, m_2 \in M, \quad \Pr_{k \leftarrow K}[Enc(k, m_1) = c_0] = \Pr_{k \leftarrow K}[Enc(k, m_2) = c_0]. \tag{1}$$

   Consideremos ahora otra noción de seguridad basada en la siguiente intuición: Si un atacante piensa que Alice le puede enviar a Bob un mensaje m_0 con una cierta probabilidad, al ver pasar un texto cifrado c_0 dicha probabilidad no cambia. Esta noción la podemos formalizar utilizando probabilidades condicionales de la siguiente forma:

   $$\forall c_0 \in C, \forall m_0 \in M, \quad \Pr_{\substack{k \leftarrow K \\ m \leftarrow M}}[m = m_0 | Enc(k, m) = c_0] = \Pr_{m \leftarrow M}[m = m_0]. \tag{2}$$

   Demuestre que esta segunda noción es equivalente a la noción de *perfect secrecy* vista en clases, es decir que un sistema criptográfico satisface (1) si y sólo si satisface (2).

2. Como vimos en clases, la siguiente es una forma general de implementar una función de hash h. Para la función h se tiene una función $h'(x, y)$ que recibe mensajes x, y de largos fijos, digamos r y s, respectivamente, y produce un mensaje de largo r. Por ejemplo, en el caso de MD5 los mensajes son dados en binario, $r = 128$ y $s = 512$. Entonces dado un mensaje m de largo arbitrario, para calcular $h(m)$ primero m es dividido de la siguiente forma:

$$m = m_1 m_2 \cdots m_k,$$

donde cada m_i es un mensaje de largo s (si el largo de m no es divisible por s, entonces se agregan símbolos adicionales a m). Luego de esto, se calcula $H_1 = h'(H_0, m_1)$, $H_2 = h'(H_1, m_2)$, ..., $H_k = h'(H_{k-1}, m_k)$, donde cada H_i es llamado un estado, H_0 es un estado inicial fijo y $h(m)$ se define como H_k.

En esta pregunta usted debe programar la función de hash MD5 considerando que H_0 es dado como parámetro. Vale decir, la versión de la función MD5 que usted va a implementar debe recibir H_0 y m, y debe realizar los cálculos como es descrito anteriormente pero considerando la definición de la función h' para MD5. Su función deberá retornar un string de largo 32 que represente el resultado de la función de hash en formato hexadecimal.

Su entrega deberá seguir las instrucciones indicadas más arriba, entregando un Jupyter Notebook que defina una función con la siguiente firma:

```python
def custom_md5(m: str, h0: int) -> str:
    # Argumentos:
    #   m:  str - mensaje
    #   h0: int - representacion de las constantes inicial a0, b0, c0, d0
    # Retorna:
    #   str - hash MD5 correcto del mensaje en formato hexadecimal
```

Para calcular los valores a_0, b_0, c_0 y d_0 de MD5 en base a `h0`, consideraremos los últimos 4 chunks de 4 bytes de `h0` (consideramos los últimos en caso de que `h0` sea mayor a 2^{128}). El orden considerará que a_0 es el más significativo de estos cuatro chunks. Consecuentemente, se puede calcular como $a_0 = (\texttt{h0} \mod (2^{32 \cdot 3})) \mod (2^{32})$. De la misma forma, d_0 es el chunk menos significativo, es decir $d_0 = \texttt{h0} \mod (2^{32})$. b_0 y c_0 se computan de forma similar.

3. En clases se mostró una forma de decriptar mensajes encriptados con la misma llave usando OTP bajo la premisa de que se conocía de antemano cierta estructura sobre los mensajes (por ejemplo, que estaban escritos en español). Alice y Bob, que se comunican en inglés y utilizando sólo caracteres cuyo código ASCII es menor o igual a 127, estaban usando OTP siempre con la misma llave y se enteraron de lo anterior. Es por esto que se han puesto de acuerdo para aumentar la complejidad del protocolo con el objetivo de que los participantes de IIC3253 no puedan decriptar sus mensajes. En particular, se han puesto de acuerdo en un conjunto de llaves (no sabemos cuántas). Cuando Alice le envía un mensaje a Bob, selecciona una de las llaves de forma aleatoria y encripta el mensaje con esa llave, para luego enviar el mensaje encriptado a Bob. Por su parte Bob intenta decriptar el mensaje con todas las llaves y, con alta probabilidad, el contenido del mensaje tendrá sentido sólo cuando utilice la llave correcta.

   En esta pregunta usted deberá decriptar un conjunto de mensajes que Alice ha enviado a Bob utilizando el protocolo anterior. A modo de ejemplo podrá sacar los mensajes de un canal que han estado usando Alice y Bob. Las instrucciones respecto de cómo escuchar ese canal se encuentran en este link.

   La entrega de esta pregunta deberá seguir las instrucciones indicadas más arriba, entregando un Jupyter Notebook que defina una función con la siguiente firma:

```python
def break_random_otp(encrypted_messages: [str]) -> {str: list[str]}:
    """
    Argumentos:
      encrypted_messages:  list[str] - lista de mensajes encriptados.
    Retorna:
      {str: list[str]} - diccionario que mapea cada una de las llaves
            utilizadas por Alice y Bob a todos los mensajes que se enviaron
            usando dicha llave (decriptados).
    """
```

Su función deberá decriptar cualquier conjunto de mensajes encriptados, suponiendo que para cada mensaje hay al menos otros quince mensajes encriptados con la misma llave.

4. Sean M, K y C espacios de mensajes, llaves y textos cifrados, respectivamente, tales que $M = K = C = \{0,1\}^n$ con $n \geq 1$. Para un sistema criptográfico (Enc, Dec) sobre M, K y C, se define el siguiente juego con parámetros $r, q \geq 1$:

   (1) El verificador escoge $b \in \{0,1\}$ con distribución uniforme.

   (1.1) Si $b = 0$, entonces el verificador escoge con distribución uniforme $K' \subseteq K$ tal que $|K'| = r$.

   (1.2) Si $b = 1$, entonces el verificador escoge con distribución uniforme una permutación $\pi : M \to M$.

   (2) Para $i = 1, 2, \ldots, q$ se realizan los siguientes pasos.

   (2.1) El adversario elije un mensaje $m_i \in M$.

   (2.2) Si $b = 0$, entonces el verificador responde de la siguiente forma. Si $m_i \neq m_j$ para cada $j \in \{1, \ldots, i-1\}$, entonces el verificar escoge $k \in K'$ con distribución uniforme y entrega la respuesta $Enc(k, m_i)$. Si $m_i = m_j$ para algún $j \in \{1, \ldots, i-1\}$, entonces el verificador entrega la misma respuesta que en el paso j (vale decir, la misma respuesta que para el mensaje m_j).

   (2.3) Si $b = 1$, entonces el verificador entrega la respuesta $\pi(m_i)$.

   (3) El adversario indica si $b = 0$ o $b = 1$, y gana si su elección es la correcta.

   El sistema criptográfico (Enc, Dec) se dice un r-pseudorandom permutation (r-PRP) si no existe un adversario que pueda ganar el juego anterior con una probabilidad significativamente mayor a $\frac{1}{2}$. Notes que el concepto de pseudorandom permutation visto en clases corresponde con esta noción para $r = 1$.

   Considerando $M = K = C = \{0,1\}^{128}$, demuestre que OTP no es un 1000-PRP si consideramos un juego con 40 rondas ($q = 40$) y una probabilidad que gane el adversario mayor a igual a $\frac{3}{4}$ (en este caso $\frac{3}{4}$ se considera significativamente mayor a $\frac{1}{2}$).