Universidad Nacional Autónoma de México

Facultad de Ciencias

Criptografía y Seguridad - 7133

P R A C T I C A 1

Equipo:

****** - 320083527**

Sosa Romo Juan Mario - 320051926

****** - 320017438**

Sánchez Estrada Alejandro - 320335950

Ortega Medina David - 319111866

Fecha de entrega:
1 de Septiembre de 2025

Profesora:
M. en C. Anayanzi Delia Martínez Hernández

Ayudantes:
Cecilia del Carmen Villatoro Ramos
José Angel Arévalo Avalos
David Armando Silva de Paz
Jesús Alberto Reyes Gutiérrez

Practica 1

Introducción

La siguiente práctica tiene como objetivo aprender cómo funcionan los cifrados clásicos y programar de manera practica un descifrador de:

- **Cifrado César**
- **Cifrado decimado**
- **Cifrado afín**
- **Base64**

Ademas busca explorar las firmas de los archivos para dar entendimiento de que son los `Magic Bytes` y su importancia a la hora de poder *leer* un archivo. Durante el desarollo, se utilizan 2 LDP para leer/manipular y escribir bytes.

El desarrollo de la practica es importante pues nos dará pie a ver como funcionan estos fundamentos teórico matemáticos a la hora de hacer encripciones que sean practicas y duras. Ademas nos brindara familiaridad con los sistemas de archivos y la manipulacion a nivel de bytes.

Desarrollo - Descifrado de archivos

1. **Descifrar el archivo file1.lol**

   Para el archivo `file1.lol` se comenzó inspeccionando su contenido binario en bruto, mostrando tanto los primeros caracteres en ASCII como en hexadecimal. El resultado no presentaba texto legible, sino bytes aparentemente aleatorios, lo que sugirió que se trataba de un cifrado clásico sobre bytes y no de una simple codificación.

   A continuación, se implementó un procedimiento en Python para probar de manera automática los cifrados César, Decimado y Afín, utilizando fuerza bruta sobre todas sus posibles claves. Una vez obtenido cada descifrado, se verificaba la cabecera con los *magic bytes* característicos de distintos formatos (PDF, PNG, MP3, MP4).

   Para el caso del cifrado Afín, se utilizó la siguiente función que aplica la fórmula de descifrado con aritmética modular en Z_{256}:

```
def affine_decrypt(data, a, b):
    try:
        inv = pow(a, -1, 256)  # inverso modular de a
    except ValueError:
        return None

    result = bytearray()
    for byte in data:
        val = ((byte - b) * inv) % 256
        result.append(val)
    return result
```

   Tras las pruebas, se encontró que el método correcto era el **cifrado Afín** con parámetros $a = 143$ y $b = 157$. El archivo resultante comenzaba con la secuencia `0xFF 0xFB`, correspondiente al *frame sync* de los archivos de audio MPEG Layer III (`.mp3`), lo que confirmó el formato.

   Una vez guardado el archivo descifrado, se verificó que podía reproducirse correctamente como audio. Con esto se validó el éxito del proceso de descifrado.

2. **Descifrar el archivo file2.lol**

   Al abordar el archivo `file2.lol`, se veían sus primeros bytes en ASCII y en hexadecimal. El contenido no presentaba texto comprensible, por lo que se asumió que se trataba de un cifrado clásico aplicado *byte a byte* en Z_{256}.

   Para automatizar el proceso, se hizo un programa que, a partir del encabezado, prueba sistemáticamente los cifrados **César** y **Afín** sobre los *magic bytes* de varios formatos (PDF, PNG, MP3, MP4). La estrategia consiste en buscar un desplazamiento (en el caso de César) o un par (a, b) (en el caso afín) que haga coincidir el encabezado descifrado con las firmas típicas de archivo.

   En particular, para **César** se verifica si existe un *shift* constante s tal que para todos los bytes del encabezado se cumpla:

$$c_i \equiv p_i + s \pmod{256},$$

donde p_i son los bytes de la firma conocida (por ejemplo, PNG) y c_i los bytes observados en el archivo cifrado. Si ese mismo s explica todos los bytes probados, se considera una detección positiva y se procede a descifrar con:

$$p_i \equiv c_i - s \pmod{256}.$$

Finalmente, se guardó el binario descifrado con extensión `.png` y se verificó su apertura correcta con un visor de imágenes, validando el éxito del proceso.

3. Descifrar el archivo file3.lol

Comencé por inspeccionar un poco el archivo utilizando un poco de código de Python:

```python
def ascii_preview(b, length=512):
  s = b[:length]
  txt = ''.join(chr(x) if 32 <= x < 127 or x in (9,10,13) else '.' for x in s)
  return txt

print("=== ASCII preview")
print(ascii_preview(head, 512))
print("\n=== Hex ===")
print(' '.join(f"{x:02x}" for x in head[:128]))
```

Como nota, todo el código que voy a mostrar para este procedimiento está incluido en el `p1CriptoEj3.ipynb` dentro del `src`.

```
=== ASCII preview
AAAAIGZ0eXBpc29tAAACAGlzb21pc28yYXZjMW1wNDEAABLebW9vdgAAAGxtdmhkAAAAAAAAAAAAAAAAAAAD

=== Hex ===
41 41 41 41 49 47 5a 30 65 58 42 70 63 32 39 74 41 41 41 43 41 47 6c 7a 62 32 31 70
```

De aquí, consulte con diversos LLMs para que me resaltaran el hecho de que todos los caracteres que encontré son imprimibles pertenecientes al alfabeto Base64 por lo que hice algunos tests para ver si se trataba de esta codificación.

```python
# 1) bytes del head en base 64
only_b64 = all((c in B64_CHARS_WITH_NL) for c in head)
print("\n1) Bytes de head en b64?", only_b64)

# 2) padding con =
has_eq = b'=' in raw[-16:] or b'=\n' in raw or raw.rstrip().endswith(b'=')
print("2) Padding con =:", has_eq)

# 3) contar caracteres en ASCII
visible_chars = [c for c in head if 32 <= c < 127 or c in (9,10,13)]
if visible_chars:
    b64_count = sum(1 for c in visible_chars if c in B64_STD or c in (10,13))
    frac = b64_count / len(visible_chars)
else:
    frac = 0.0
```

```python
print(f"3) Fraccion de contenido que es ASCII: {frac:.3f}")

# 4) Medir porcentaje de lineas longitud 4
lines = ascii_preview(raw, 4096).splitlines()
if lines:
    lengths = [len(l) for l in lines if len(l.strip())>0]
    if lengths:
        multiples_of_4 = sum(1 for L in lengths if L % 4 == 0)
        pct_mult4 = multiples_of_4 / len(lengths)
    else:
        pct_mult4 = 0.0
else:
    pct_mult4 = 0.0
print(f"4) Medir porcentaje de lineas longitud 4: {pct_mult4:.3f}")

# 5) Buscar firmas de formatos binarios
signs = []
if raw.startswith(b"%PDF"):
    signs.append("PDF")
if raw.startswith(b"\x89PNG\r\n\x1a\n"):
    signs.append("PNG")
if raw.startswith(b"\xff\xd8\xff"):
    signs.append("JPEG")
if raw.startswith(b"PK\x03\x04"):
    signs.append("ZIP")
print("5) Firmas de formatos binarios en el inicio:", signs if signs else "False")
```

Dandome el siguiente resultado:

```
1) Bytes de head en b64? True
2) Padding con =: True
3) Fraccion de contenido que es ASCII: 1.000
4) Medir porcentaje de lineas longitud 4: 1.000
5) Firmas de formatos binarios en el inicio: False
```

Por estas métricas, todo indica que es base64. Como breviario cultural, la base 64 es una forma de codificar binario en texto de manera que los datos binarios están representados en secuencias de caracteres imprimibles usando el formato ASCII, específicamente cada 6 bits de los datos binarios de representan con un carácter del conjunto de 64 posibles. Es útil para transmitir datos binarios si solo podemos poner texto ademas, si la cadena no es del tamaño adecuado se utiliza el carácter de relleno =. Vemos que de las métricas recolectadas, todo parece apuntar a esta codificación.

Hice un código que verifica con heurísticas si es base 64, e implemente una decodificación de base64, después busca en la cabeza del nuevo archivo los magic bytes para saber la extensión del archivo y después de saber que es `mp4` puse también código para poder verlo en el mismo notebook. Todo esto se puede ver en la ruta antes mencionada y aquí los resultados:

```
[+] Leído file3.lol: 205704 bytes
[*] Heurística: parece Base64
[+] Decodificación Base64 OK, tamaño 154276 bytes
[+] Guardado: file3_decoded.mp4  (detected: MP4 (ftyp))
Salida: file3_decoded.mp4

Video:
```

Incluyo aquí también el decodificador:

```python
def base64_decode_from_scratch(s):
    """
    Decodificador base64 desde cero.
    Acepta s como str (puede contener espacios/newlines).
    Devuelve bytes decodificados.
    """
    # Mantener solo chars válidos + '='
    s = ''.join(ch for ch in s if ch in _b64chars or ch == '=')
    if not s:
        return b''
    # Si la longitud no es múltiplo de 4, rellenar con '='
    pad_len = (4 - (len(s) % 4)) % 4
    if pad_len:
        s += '=' * pad_len

    out = bytearray()
    for i in range(0, len(s), 4):
        block = s[i:i+4]
        vals = []
        pad = 0
        for ch in block:
            if ch == '=':
                vals.append(0)
                pad += 1
            else:
```

```
            # Asumir 'A' (valor 0) si char inválido
            vals.append(_b64rev.get(ch, 0))
    # recomponer 24 bits
    triple = (vals[0] << 18) | (vals[1] << 12) | (vals[2] << 6) | (vals[3])
    b1 = (triple >> 16) & 0xFF
    b2 = (triple >> 8) & 0xFF
    b3 = triple & 0xFF
    out.append(b1)
    if pad < 2:
        out.append(b2)
    if pad == 0:
        out.append(b3)
return bytes(out)
```

4. **Descifrar el archivo file4.lol**

Para el archivo `file4.lol` se comenzó con la ver sus primeros bytes, tanto en ASCII como en hexadecimal. A diferencia de un binario "puro", el contenido estaba conformado exclusivamente por caracteres válidos de Base64 (`A-Z`, `a-z`, `0-9`, `+`, `/` y posibles `=` de relleno), distribuidos como texto. Esto sugirió una **codificación Base64** más que un cifrado clásico.

Primero se descartaron los cifrados **César** y **Afín** aplicados *byte a byte*: al intentar alinear el encabezado descifrado con firmas conocidas (PNG, PDF, MP3, MP4), no se encontró un desplazamiento constante (César) ni un par (a, b) (Afín) que explicara coherentemente varios bytes de una misma firma. Con ello, se procedió a decodificar en Base64.

Tras la **decodificación Base64**, se verificó el encabezado del binario resultante contra los *magic bytes* conocidos. Los primeros cinco bytes fueron:

`25 50 44 46 2D` (`%PDF-`),

que corresponden inequívocamente al formato **PDF**. Esto confirmó que el archivo original era un documento `.pdf`.

Desarrollo - preguntas

1. **¿Cuantos primos relativos hay en $\mathbb{Z}_{256}$?**

   Para responder esta pregunta voy a hacer uso de la función ϕ de Euler [4]. La función $\phi(n)$ cuando $n = 256 = 2^8$ cuenta la cantidad de enteros positivos menores o iguales a n que son primos relativos con este mismo:

   $$\begin{aligned}\phi(p^k) &= p^k - p^{k-1} \\ &= 2^8 - 2^7 \\ &= 256 - 128 \\ &= 128\end{aligned}$$

   Por tanto hay 128 enteros en el anillo que son primos relativos a 256.

2. **Sea $f : \mathbb{Z}_n \to \mathbb{Z}_n$ tal que $f(i) = k * i$ para $i \in A$ con A un alfabeto cualquiera ¿qué se debe de cumplir para que f sea una función biyectiva?**

   Tenemos la función

   $$f : \mathbb{Z}_n \to \mathbb{Z}_n, \quad f(i) = k \cdot i \pmod{n}$$

   donde $\mathbb{Z}_n$ son los enteros módulo n (los números $0, 1, 2, \ldots, n-1$), y k es un número fijo.

   Para que f sea biyectiva debemos de cumplir dos cosas:
   1. Inyectiva: que no haya dos elementos distintos que se manden al mismo resultado.
   2. Sobreyectiva: que todos los elementos de $\mathbb{Z}_n$ se puedan obtener como imagen de alguno.

   Es un conjunto finito, sol debemos de revisar la inyectividad: si la función no repite valores, entonces será sobreyectiva.

   Si tomamos dos elementos i y j y resulta que $f(i) = f(j)$, entonces

   $$k \cdot i \equiv k \cdot j \pmod{n}.$$

   Esto se simplifica a

   $$k \cdot (i - j) \equiv 0 \pmod{n}.$$

   Lo que significa que n divide a $k \cdot (i - j)$.

   Aquí aparece el máximo común divisor $\gcd(k, n)$:
   - Si $\gcd(k, n) = d > 1$, entonces es posible que $i \neq j$ pero aún así $k(i - j)$ sea múltiplo de n. En este caso la función no es inyectiva.
   - Si $\gcd(k, n) = 1$, no hay divisores comunes, y la única forma de que $k(i - j)$ sea múltiplo de n es que $i - j$ lo sea. Es decir, si $i \neq j$, sus imágenes no se confunden.

   En conclusión, la función $f(i) = k \cdot i \pmod{n}$ es biyectiva si y sólo si

   $$\gcd(k, n) = 1.$$

   [3]

3. **¿Cuántas posibles combinaciones no triviales existen para cifrar bytes con César, Decimado y Afin?**

   Primero nos surge la duda de que nos referimos con "no triviales", con eso en mente, vamos a intentar responder lo mejor posible.
   Cifrado César:

   a) Combinaciones no triviales: 255

   b) Nuestro razonamiento es el siguiente, con 256 valores posibles (0-255) habrá 256 desplazamientos totales, excluimos el dezplazmaiento 0, trivial, nos quedan 255 útiles, en el alfabeto de 26 letras siguiendo la misma regla hay 25 desplazamientos utiles, , entonces para 256 simbolos, los que consideramos no triviales son 256-1=255 [1]

   CifradoDecimado

   a) COmbinaciones no triviales: 127

   b) NUestro razonamiento, este cifrado usa la fórmula $C = aM$ mód 256, donde a debe ser coprimo con 256. El número de enteros coprimos con 256 es $\varphi(256) = 128$. Excluyendo la clave trivial $a = 1$ (identidad), quedan $128 - 1 = 127$ combinaciones no triviales. [4]

   Cifrado Afín

   a) COmbinaciones no triviales: 32767

   b) El cifrado afín usa la fórmula $C = aM + b$ mód 256. Con $n = 256$ hay $\varphi(256) = 128$ valores posibles de a (coprimos con 256) y 256 valores posibles para b, resultando en $128 \times 256 = 32768$ pares clave totales. Al excluir la clave identidad ($a = 1, b = 0$), quedan $32768 - 1 = 32767$ combinaciones no triviales.[2]

4. **¿Por qué el sistema de archivos de UNIX, aunque un archivo tenga una extensión diferente (o incluso no tenga), sigue reconociendo al archivo original?**

   Porque el sistema de archivos de UNIX identifica el tipo de archivo por su contenido interno, especialmente por los magic bytes, no por la extensión, por ejemplo, los archivos **.png** empieza con los bytes **89 50 4E 47 0D 0A 1A 0A**, y un archivo **.pdf** empieza con **%PDF**. Por esto cuando se utiliza **file** en UNIX, el comando lee los primeros bytes para determinar el formato del archivo.

5. **¿Por qué los archivos descifrados tienen exactamente el mismo tamaño antes de cifrar pero no pudimos leerlos? ¿Por qué no tuvimos que agregar/quitar nada?**

   Lo primero que hay que recordar es que estamos usando cifrados en donde reemplazamos o reacomodamos la información original, por ejemplo con sustituciones 1 a 1 entre letras o en el caso de base64 solo codificamos (aunque esto si aumenta el tamaño) porque usamos 4 caracteres por cada 3 bytes pero regresa al mismo tamaño al descifrar.

   El que no se pueda leer es porque los programas que se utilizan para leer estos archivos esperan cierto formato en los archivos, especialmente cuando ven la firma y buscan los magic bytes para saber como leerlo, entonces al modificar las estructuras y los bytes que utilizan para decodificarlo la información queda de cierta manera inutilizada.

   En cuanto a agregar a eliminar cosas vamos por casos:

a) César, Decimado, Afín: esto es sustitución monoalfabetica 1 a 1 por lo que el tamaño no debe cambiar y no es necesario quitar o añadir nada.

b) B64: Esto como ya dijimos es una reacomodación de la información en bloques, si se agrega el carácter de padding usualmente `=` pero al decodificar se omite este mismo por lo que no se quita ni agrega nada.

De manera concisa podríamos decir que las operaciones que usamos son biyectivas sobre el conjunto de símbolos.

6. **Ya que *base64* no es un cifrado, sino codificación, ¿en qué casos podemos usarlo?**

Base64 nos sirve para representar datos binarios en texto ASCII, y esto es útil cuando un canal de comunicación unicamente permite caracteres imprimibles, los casos en los que sería útil usar Base64 son:

- Para transmitir datos binarios en canales de texto, por ejemplo enviar imágenes en correos electrónicos.
- Para incluir datos binarios en un JSON o XML, por ejemplo incrustar una imagen un una respuesta JSON para un API.
- Para Codificar datos para URLs, por ejemplo, evitar caracteres inválidos o reservados un una URL

Conclusiones

Para cerrar, queremos expresar como equipo lo que observamos durante la P1. Lo primero que consideramos relevante destacar es lo interesante y casi sorprendente que resultó transformar archivos que parecían ruido o errores en videos, PDFs, entre otros formatos. Esto evidencia lo potente que puede ser el entorno de la criptografía. Ahora bien, algo que nos llamó la atención fue que cuatro estudiantes sin una preparación profunda fuimos capaces de romper todos estos métodos de cifrado. Esto nos lleva a concluir que, en el contexto actual, tales métodos resultan demasiado vulnerables: son simples y pueden ser quebrados con relativa facilidad mediante fuerza bruta o análisis básico. Nos interesa continuar con el curso para aprender cómo expandir estos algoritmos y diseñar propuestas más robustas a partir de herramientas similares.

Otro aspecto que identificamos fue la dificultad de trabajar en equipo a distancia, sin conocernos previamente y sin tener experiencia sólida en la materia. Aun así, logramos llegar a un producto que, aunque no fue exactamente lo que esperábamos, representó un esfuerzo conjunto. Además, consideramos que hubiera sido útil plantear más dudas sobre las especificaciones de la práctica con varios días de anticipación, lo que nos habría permitido organizarnos de una manera más eficiente.

Como aprendizaje, nos llevamos que aún queda mucho por explorar en el campo de la criptografía y que estamos motivados para realizar una entrega más sólida en la siguiente ocasión.

Bibliografía

Referencias

[1] dCode. Cifrado césar. https://www.dcode.fr/cifrado-cesar, Sin año. Último acceso: 1 de septiembre de 2025.

[2] GusA2. Understanding affine cipher: Key space, encryption. https://www.coursesidekick.com/computer-science/3282538#:~:text=case%2C%20we%20let%20n%20%3D,256%20%3D%2032%2C768.%20Question%202, Junio, 2023. Último acceso: 1 de septiembre de 2025.

[3] su18 eecs70. Building the foundations of modular arithmetic. https://www.su18.eecs70.org/static/slides/lec-7-handout.pdf, Sin año. Último acceso: 1 de septiembre de 2025.

[4] Wikipedia contributors. Función de euler. https://es.wikipedia.org/wiki/Funci%C3%B3n_%CF%86_de_Euler, 2025. Último acceso: 1 de septiembre de 2025.