Algoritmo de *hashing* utilizando multiplicación de matrices

Facultad de Ingeniería

Universidad Panamericana

Profesor: José Manuel Rosales

Materia: Álgebra Lineal

Fecha de entrega: 15 / 05 / 2020

Alumnos:	**ID**
Álvaro Paulo Creus Parrilla	0223425
Bruno Campos Uribe	0223329
Rafael Patiño Goji	0226064

ÍNDICE

Introducción

La necesidad de guardar secretos siempre ha existido. Por esa razón, la historia ha presenciado el desarrollo de la criptografía. Desde los egipcios con sus jeroglíficos, los romanos con el cifrado de césar y los alemanes con enigma. La criptografía ha sido necesaria para comunicar información de forma segura.

El propósito principal de la criptografía es establecer comunicación segura, de manera que la información esté protegida de cualquier ataque. Incluso si el mensaje llegase a ser interceptado por un tercero, este no podría entenderlo. En este contexto, se debe mantener la privacidad, conservar la integridad de la información y tener capacidad de autenticación. Es por eso que el propósito de la criptografía no sólo es proteger la información, sino también validar la autenticidad del mensaje.

Existen tres principales sistemas de encriptación: Algoritmos *simétricos*, *asimétricos* y *hash.* Mientras que los primeros dos son usados para cifrar y descifrar, el *Hash* es utilizado para autenticar. El algoritmo *Hash* genera a partir de cualquier entrada de datos, una serie de caracteres única, con longitud fija.

Justo esta última característica es lo que hace especial al cifrado hash, pues es de gran utilidad en la informática. Supongamos que se necesita verificar la integridad de un archivo, en vez de verificar la información directamente, se corrobora que el código hash (que es pequeño en comparación con el documento mismo) coincida con el hash previamente conocido. Si coincide, significa que el archivo es auténtico. Pues si la información fuera alterada el código generado cambiaría de igual manera.

Este proyecto explora la posibilidad de generar códigos *hash* con un algoritmo que use multiplicación de matrices.

Marco teórico

La criptografía se basa en transformar los contenidos de un mensaje siguiendo reglas establecidas. Estas reglas modifican la información del mensaje, de modo que, aplicando las reglas inversas se recupere el mensaje original. El método de cifrado consiste en codificar el mensaje de modo que se codifiquen en números y así efectuar determinadas operaciones matemáticas en ellos.

En 1883 el holandés Kerckhoffs von Nieuwenhof escribió: “La seguridad de un criptosistema no debe depender de mantener secreto el algoritmo de cifrado empleado. La seguridad depende sólo de mantener la clave secreta.” Este enunciado se conoce como el “principio de Kerckhoffs” y todos los sistemas de cifrado moderno siguen este paradigma.

Cifrado de Hill

Creado en 1929 por el matemático Lester S. Hill, el cifrado de Hill es un algoritmo que codifica su mensaje en un campo módulo 27 y encripta el mensaje utilizando multiplicación de matrices.

El cifrado de Hill utiliza una matriz cuadrada como clave, la cual se multiplica por vectores columna que contiene el mensaje original, siguiendo la transformación:

$$Y = AX$$

Donde *A* es la matriz clave y *X* es la matriz columna conteniendo el mensaje que se quiere encriptar.

Para decodificar los mensajes producidos por el cifrado de Hill se necesita que la matriz clave sea invertible, para así utilizar la matriz A^{-1} como clave siguiendo la transformación inversa:

$$X = A^{-1}Y$$

Función Hash

La función hash o función resumen genera una salida, comúnmente llamada resumen (o *hash*), de longitud fija a partir de un mensaje de longitud variable. El resumen puede ser utilizado como una “huella digital” del mensaje (o información) original.

Los valores generados por las funciones hash son mucho más pequeños que los mensajes, por lo que pueden ser distribuidos con mayor facilidad. El principal uso de este tipo de funciones es para verificar la integridad de un mensaje, analizándolo con la misma función y comparando el resultado con el resumen almacenado. En este sentido el resumen funciona como una “huella digital” del mensaje.

Dentro de la clasificación de algoritmos de criptografía, las funciones hash son “de una vía”, es decir, que una vez encriptado no se puede recuperar el contenido original del mensaje.

Para ser viable en autenticación, una función hash debe cumplir con las siguientes características:

- Puede aplicarse a cualquier tamaño de bloque de datos.
- Produce una salida de longitud fija.
- Fácil de calcular.
- Para un valor dado de hash h es difícil encontrar una x tal que $H(x) = h$.
- Para cualquier tamaño de bloque x, es difícil encontrar $y \neq x$ tal que $H(y) = H(x)$.
- Es difícil encontrar un par (x, y) tal que $H(y) = H(x)$.

Campos Finitos

Un campo o cuerpo es un sistema numérico en el cual se pueden realizar operaciones aritméticas (adición, multiplicación y sus inversas) y cumplen las propiedades: asociativa, conmutativa y distributiva. Todos los espacios vectoriales también son campos con una cantidad infinita de elementos. ($\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$).

Un campo finito es un cuerpo definido sobre un conjunto finito de elementos. Si *F* es un cuerpo finito, existen un primo *p* y un natural *n*, ambos únicos, donde p^n es el número de elementos. Un campo finito se representa como $GF(p^n)$.

Los campos $GF(p)$ se llaman campos primos y sus elementos pueden ser representados con los enteros $0, 1, \ldots, p-1$. Las operaciones aritméticas de estos campos son adición y multiplicación módulo *p*. El inverso aditivo de cualquier elemento a está dado por la identidad:

$$a + (-a) = 0 \mod p$$

El inverso multiplicativo de cualquier elemento $a \neq 0$ está dado por:

$$a \cdot a^{-1} = 1$$

Un campo primo muy importante es $GF(2) = \{0, 1\}$, que es el campo finito más pequeño que existe. En *GF(2)* la adición es equivalente a la compuerta lógica XOR, y la multiplicación es equivalente a la compuerta lógica AND.

Objetivo

Construir un algoritmo que genere un código *hash* (de un solo sentido) a partir de una entrada de datos de longitud arbitraria y utilizando multiplicación de matrices.

Procedimiento

El cifrado Hill utiliza una matriz invertible para descifrar sus mensajes, por lo que si utilizamos una matriz no invertible como clave obtenemos un algoritmo de una vía que cumple con las condiciones necesarias para una función hash.

En el cifrado original de Hill, se realizan operaciones en módulo 27 para poder codificar todas las letras del alfabeto. Sin embargo, tenemos la ventaja de que una computadora ya codifica cualquier información en binario, por lo que podemos trabajar en el campo finito *GF(2)*.

Matriz no invertible

Para generar una matriz no invertible fácilmente nos basamos en los siguientes lemas:

> **Lema.** Si una matriz M^w tiene precisamente w elementos iguales a 1, entonces M^w es invertible si es una matriz de permutación.
>
> **Prueba.** Cualquier matriz de permutación es invertible. Por el contrario, si M^w tiene precisamente w elementos iguales a 1 pero no es una matriz de permutación, entonces alguna fila o columna contiene todos los ceros, en cuyo caso M^w no es invertible. ∎

> **Lema.** Sean M_1^w y M_2^w matrices de permutación. La suma $M_1^w + M_2^w$ es no invertible.
>
> **Prueba.** Sea $M^w = M_1^w + M_2^w$. Supongamos que existen r y c de tal manera que $M_1^w[r,c] = M_2^w[r,c] = 1$. Entonces la fila r de M^w contiene sólo ceros, así que M^w no es invertible. Por lo tanto, suponemos que no existen tales r y c; en este caso, M^w tiene precisamente dos unos en cada fila y columna. Por inducción se prueba que tales matrices no son invertibles. (Plank & Buchsbaum, 2007, pp. 9)

Algoritmo (pseudocódigo)

INPUT: Matriz no invertible $P \in GF(2)$ de tamaño m y $BitBuffer\{b_0, b_1, \dots, b_k\}$
OUTPUT: valor hash de m bits.

1. $M :=$ padding(m, *BitBuffer*)
2. $N := M.lenght \,/\, m$
3. Initialize $B\{B_0, B_1, \dots, B_N\}$
4. FOR i from 0 to N
    1. Initialize b : array
    2. FOR j from 0 to $m-1$
        1. $b_j = M_{i*m+j}$
    3. $B_i = b$
5. Initialize $h = 0_m$
6. FOR i from 0 to $N-1$
    1. $h = h + B_i$
    2. $h = P \times h$
7. OUTPUT h

FUNCTION padding
INPUT: m, binary values $B\{b_0, b_1, \dots, b_k\}$
OUTPUT: binary values B

1. IF $k \bmod m \mathrel{!}= 0$ then
    1. FOR i from 0 to m - ($k \bmod m$)
        1. $b_{k+i} = 0$

Problema aplicado

Para implementar nuestro algoritmo utilizamos el lenguaje `Python` y la librería `numpy`, la cual implementa estructuras de matriz y operaciones matriciales.

Para nuestra matriz clave elegimos un tamaño de 32, lo cual corresponde a un hash de 32 bits. Para generar la matriz generamos una matriz identidad y con la función `numpy.random.permutation(x)` se convierte en una matriz de permutación. Definimos la función `invertibility(x)` para validar las propiedades necesarias de nuestras matrices.

Debido a la importancia de una llave estándar guardamos la matriz resultante en el archivo `key32.npy`.[1]

generateKey.py

```
1. import numpy as np
2.
3. def invertibility(x):
4.     try:
5.         np.linalg.inv(x)
6.     except np.linalg.LinAlgError:
7.         print("Matriz no invertible")
8.     else:
9.         print("Matriz invertible")
10.
11.    if __name__=="__main__":
12.        I = np.eye(32, dtype=int)
13.
14.        M1 = np.random.permutation(I)
15.        M2 = np.random.permutation(I)
16.
17.        print("\nInvertibilidad M1:")
18.        invertibility(M1)
19.        print("\nInvertibilidad M2:")
20.        invertibility(M2)
21.
22.        P = (M1 + M2)%2
23.        print("\nInvertibilidad P:")
24.        invertibility(P)
25.
26.        print("\nMatriz Clave")
27.        with np.printoptions(threshold=np.inf):
28.            print(P)
29.
```

[1] La matriz que guardamos y utilizamos para los hash fue de otra iteración del programa. Por lo que no es representada por la que aparece en el documento

```
    with open('key32.py', 'wb') as file:
        np.save(file, P)
    print("\nKey saved!")
```

Output:

```
Invertibilidad M1:
Matriz invertible

Invertibilidad M2:
Matriz invertible

Invertibilidad P:
Matriz no invertible

Matriz Clave
[[0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
 [0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
 [0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0]
 [0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
 [0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
 [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0]
 [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
 [0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0]]

Key saved!
```

En `hash.py` implementamos el algoritmo descrito anteriormente con la función `bitHash`. Debido a que python representa la información con el tipo `bytes`[2], implementamos la función `myHash` como un wrapper que convierte los valores `bytes` en una lista de valores binarios. Para facilidad de lectura convertimos los valores obtenidos a hexadecimal.

hash.py

```
1. import numpy as np
2.
3. # Some hashes
4. quotes = [b"Look how they massacred my boy",
5.           b"One does not simply walk into Mordor",
6.           b"It's a trap!",
7.           b"A fine addition to my collection",
8.           b"Another happy landing",
9.           b"Ya like jazz?"]
10.
11.   ## hash function
12.   def padding(B, m):
13.       if len(B)%m != 0:
14.           for i in range(m - len(B)%m):
15.               B.append(0)
16.       return B
17.
18.   def bitHash(P, bits):
19.       m = len(P)
20.       bits = padding(bits, m)
21.       N = len(bits) // m
22.       B = np.empty((N, m), dtype=int)
23.       for i in range(N):
24.           column = []
25.           for j in range(m):
26.               column.append(bits[i*m+j])
27.           B[i] = column
28.       for i in range(N):
29.           if i == 0:
30.               h = B[i]
31.           else:
32.               h = (h + B[i]) % 2
33.           h = np.matmul(P, h)%2
34.       return h
35.
36.   def myHash(P, byteStr):
37.       bits = "{:b}".format(int(byteStr.hex(), 16))
38.       bithash = bitHash(P, list(bits))
39.       hsh = ''
40.       for i in range(len(bithash)):
```

[2] https://docs.python.org/3/library/stdtypes.html#binary-sequence-types-bytes-bytearray-memoryview

```
            hsh += str(bithash[i])
        return "{:0{}x}".format(int(hsh, 2), len(P)//4)


if __name__=='__main__':
    # Load Matrix
    with open('key32.npy', 'rb') as f:
        P = np.load(f)
    print("Key Matrix:")
    print(P)
    print()

    for i in range(len(quotes)):
        print("Message: {}".format(quotes[i]))
        print("Hash: {}".format(myHash(P, quotes[i])))
        print()
    # Hash some file
    with open('key32.npy', 'rb') as f:
        sumthn = f.read()

    print("File: key32.npy")
    print("Hash: ", myHash(P, sumthn))
```

Output:

```
Key Matrix:
[[0 0 0 ... 0 1 0]
 [0 0 0 ... 0 0 0]
 [1 0 0 ... 0 1 0]
 ...
 [0 0 0 ... 1 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]

Message: b'Look how they massacred my boy'
Hash: 5f6db0b7

Message: b'One does not simply walk into Mordor'
Hash: c7ee30a8

Message: b"It's a trap!"
Hash: 5eabf267

Message: b'A fine addition to my collection'
Hash: dde01a9f

Message: b'Another happy landing'
Hash: eff2f77a

Message: b'Ya like jazz?'
Hash: bd73a118

File: key32.npy
Hash: 1f9ca7c6
```

Conclusión

El álgebra lineal es una disciplina con numerosas aplicaciones. Este proyecto se enfocó principalmente en una de sus aplicaciones dentro del mundo de la computación. El desarrollo de un algoritmo que pudiera computar un código *hash* de manera eficiente utilizando la multiplicación de matrices no fue fácil; investigamos áreas de programación que se relacionan profundamente con el tema para comprender mejor el problema presentado. Las fuentes consultadas fueron de gran utilidad para poder entender la lógica que un programa con tales cualidades debería seguir. De igual forma, los conocimientos aprendidos sobre álgebra lineal facilitaron la investigación sobre el tema, pues no bastaba con solo idear un programa sino que se tenía que entender cada proceso del mismo. Todo esto nos llevó a generar un algoritmo que cumple con el objetivo planteado y funciona correctamente. Finalmente nos dimos cuenta de los dos beneficios que recibimos de hacer el proyecto, el primero fue entender el funcionamiento matemático de los algoritmos de encriptamiento, y el segundo fue entender y aplicar el mismo funcionamiento de manera digital.

Bibliografía y fuentes

- Hernández Encinas, L. (2016). La criptografía. Madrid, Spain: Editorial CSIC Consejo Superior de Investigaciones Científicas. Recuperado de https://elibro.net/es/ereader/upanamericana/41843?page=71
- Tomé, C. (2017, enero 11). Criptografía con matrices, el cifrado de Hill. Recuperado de https://culturacientifica.com/2017/01/11/criptografia-matrices-cifrado-hill/
- Forouzan, B. A. (2008). Introduction to Cryptography and Network Security. New York, Estados Unidos: McGraw-Hill Education.
- Farajallah, M., Abu Taha, M., & Tahboub, R. (2011). A Practical One Way Hash Algorithm based on Matrix Multiplication. International Journal of Computer Applications, 23(2), 34-38. https://doi.org/10.5120/2859-3677
- Berisha, A., Baxhaku, B., & Alidema, A. (2012). A Class of Non Invertible Matrices in GF(2) for Practical One Way Hash Algorithm. International Journal of Computer Applications, 54(18), 15-20. https://doi.org/10.5120/8667-2574
- Paar, C., & Pelzl, J. (2011). Understanding Cryptography. New York, Estados Unidos: Springer Publishing.
- Plank, J. S., & Buchsbaum, A. L. (2007). Some Classes of Invertible Matrices in GF(2). Recuperado de http://web.eecs.utk.edu/~jplank/plank/papers/CS-07-599.html