Universidad Diego Portales
Criptografia y Seguridad en redes
Primer Semestre 2022

Actividad 3 - Hash

Javier Ramos Acevedo javier.ramos1@mail.udp.cl

1. Introducción

En el siguiente informe se especificara la construcción y diseño de un algoritmo de Hash propio, llamado "za Hash", el cual fue implementado en un programa en el lenguaje python, donde además se implementaron distintas funcionalidades, como el calculo de la entropía de la clave/texto ingresado, el calculo del tiempo de ejecución y hash de archivos de texto (linea por linea).
Además se realizo un análisis comparando el rendimiento de "za Hash" tanto a nivel de entropía como a nivel de tiempo de ejecución.

2. Desarrollo y Diseño

El algoritmo za Hash es un algoritmo sencillo, el cual se apoya de la creación de una semilla basada en el tiempo y operaciones XOR junto con expansiones de bits, todo esto para otorgar un output de 55 caracteres, el cual es el valor fijo de salida.

Cabe destacar que todo el código que sera explicado a continuación puede ser encontrado en el siguiente enlace: Github: GrahaExarch - Actividad 3

2.1. Proceso

El proceso que "za Hash" realiza se aplica de la siguiente forma:

1. Se recibe el input (ya sea por stdin o por archivo de texto) y se calcula su equivalente ASCII en bits el cual se almacena caracter por caracter en una lista.

2. Una vez se tienen los bits del input se procede a calcular la semilla, esto se hace con la librería time de python, valor que se divide en 100 para obtener una variación en su resultado cada cierto periodo de tiempo (más o menos 1 o 2 min.), luego se calcula el equivalente en bits de este tiempo. Finalmente la semilla se crea con los bits del tiempo mas los bits del primer caracter del texto a hashear, mas los bits del tiempo nuevamente, seguidos por un 0. (En la sección del código se vera en mas detalle).

3. Una vez obtenido el texto a hashear y la semilla, se procede a comenzar el proceso de hash. Para esto se divide la semilla en una bloques binarios de 7 bits. Por otra parte también se tienen los caracteres del input en binario, se comienza a iterar sobre la misma. Por cada iteración i, se realiza un XOR entre el i-esimo caracter del binario del input y el n-esimo bloque de la lista de la semilla. En caso de que se hayan recorrido todos los bloques de la semilla, el contador n de la semilla se reinicia.

4. Los bits "hasheados" se almacenan concatenando el XOR calculado junto al caracter del stream de bits que recibió la función. Si el resultado de este proceso es un binario de largo mayor a 330 bits, se procede a la ultima etapa del algoritmo, en caso contrario, se vuelve a hashear, pero esta vez el texto ingresado son los bits que se obtuvieron como resultado, y la semilla es cambiada por un XOR entre la semilla antigua y los bits obtenidos del hash. Este proceso funciona de forma recursiva hasta que se obtienen mas de 330 bits.

5. Una vez se obtiene un binario de al menos 330 bits, se realiza un ultimo XOR entre los primeros 330 bits y los últimos 330 bits de este binario, obteniendo de esta forma un ultimo binario de 330 bits.

6. Para terminar todo el proceso se transforma este ultimo binario a base64, obteniendo así nuestro texto zaHasheado.

El algoritmo se puede ver en el siguiente diagrama:

Figura 1: Diagrama de za Hash

2.2. Código

2.3. Librerías

Las librerías utilizadas se muestran a continuación:

```
1  import hashlib
2  import re
3  import textwrap
4  import time
5  from math import log
6  from timeit import default_timer as timer
```

Figura 2: Librerías Utilizadas

- hashlib: Incluye las funciones de hash md5, sha1 y sha256.
- re: Funciones con Expresiones regulares.
- textwrap: Permite crear listas a partir de strings.
- item: Otorga distintas funcionalidades de tiempo.
- math: Permite el uso de distintas operaciones matemáticas, en este caso el Logaritmo.
- timeit: Permite el calculo del tiempo de ejecución con mayor exactitud que métodos que usan las funciones de time.

2.4. Funciones

A continuación se muestra el código de cada función desarrollada y una explicación de ser necesaria.

1. **measure_time**: Decorador, se usa antes de cada función de hash.

```python
def measure_time(funcion: Callable) -> Callable:
    """Mide el tiempo de ejecucion

    Parameters
    ----------
    funcion : function
        funcion a la cual se le mide el tiempo
    Returns
    -------
    function: resultado de la funcion
    :Authors:
        - Javier Ramos
    """

    def measure_func(*args, **kwargs):
        inicio = timer()
        c = funcion(*args, **kwargs)
        print(
            '================================================================================'
        )
        print(f"\u2193 Tiempo de Ejecucion: {float(timer() - inicio)}")
        return c

    return measure_func
```

Figura 3: Código y comentario Decorador

2. **toBits**.

```python
def toBits(bits: list) -> list:
    """Dada una lista de caracteres ASCII, Retorna su version en bits

    Parameters
    ----------
    list : list
        Lista de caracteres ASCII
    Returns
    -------
    list: lista de bits correspondiente a los caracteres ASCII
    :Authors:
        - Javier Ramos
    """
    bit_list = []
    arreglo_de_bytes = bytearray(bits, "utf8")
    for b in arreglo_de_bytes:
        bit_list.append(bin(b).split('b')[1])
    return bit_list
```

Figura 4: Código y comentario toBits

3. **createSeed**: La semilla se crea concatenando los bits de la semilla, el primer elemento de la lista, los bits de la semilla (nuevamente) y un "0" (variable seed).

```python
def createSeed(bitList: list) -> str:
    """Crea la semilla a usar en za Hash, usando el tiempo
            y el primer elemento de la lista que recibe.

    Parameters
    ----------
    bitList : list
        Binarios de 7 bits que representan characteres en ASCII.
    Returns
    -------
    str: Retorna la semilla inicial de largo 56 bits.
    :Authors:
        - Javier Ramos
    """
    timeseed = int(time.time() / 100)
    seedbits = bin(timeseed).split('b')[1]
    seed = seedbits + bitList[0] + seedbits + "0"
    return seed
```

Figura 5: Código y comentario createSeed

4. **xor**.

```python
def xor(base: str, key: str) -> str:
    """calcula el XOR entre 2 streams de bits

    Parameters
    ----------
    base : str
        steam de bit al cual se le aplica el XOR
    key : str
        llave para aplicar el XOR
    Returns
    -------
    str: Retorna un string del XOR resultante
    :Authors:
        - Javier Ramos
    """
    xorList = [(ord(a) ^ ord(b)) for a, b in zip(base, key)]
    return "".join(map(str, xorList))
```

Figura 6: Código y comentario xor

5. **bestHash**.

```python
def bestHash(bitList: list, seed: str) -> str:
    """ Calcula el hash de una lista de bits dada,
            asegurandose que el resultado sea de 330 bits.

    Parameters
    ----------
    bitList : list
        bloques binarios de los caracteres del string a hashear
    seed : str
        la semilla en binario
    Returns
    -------
    str: retorna un string de 330 caracteres en binario
    :Authors:
        - Javier Ramos
    """
    hashedBits = ""
    count = 0
    seedSplit = textwrap.wrap(seed, 7)
    for char in bitList:
        a = xor(char, seedSplit[count])
        hashedBits += a + char
        count = count + 1 if count < 7 else 0
    lenght = len(hashedBits)
    if lenght < 330:
        hashedList = textwrap.wrap(hashedBits, 7)
        hashedList = list(
            map(
                lambda x: (x + x[0] * (7 - len(x))) if len(x) < 7 else x,
                hashedList,
            )
        )
        return bestHash(
            hashedList,
            xor(hashedBits, seed),
        )

    else:
        # print(f" {hashedBits} seed: {seed} len: {len(hashedBits)}")
        return splitBits(hashedBits)
```

Figura 7: Código y comentario bestHash

La función se divide en 2 partes:

- En el *for* se recorren los caracteres del input que se recibe y se calcula un *XOR* contra un bloque de la semilla, estos bloques se recorre con su propio contador (**count**), el cual a llegar a 7 se reinicia.
- La segunda parte corresponde a la verificación del tamaño del hash, específicamente en el *if* y su *else*, donde el *if* presenta la recursividad de la función para asegurar **al menos 330 bits**.
- la función lambda recorre bloques de 7 bits del teto hasheado y cuando encuentra algún bloque con menos de 7 bits, lo rellena con el primer bit de dicho bloque.

6. **splitBits**.

```python
def splitBits(bits: str) -> str:
    """ recibe un binario de mas de 330 bits de largo y
        retorna el resultado de un xor entre 2 mitades del
        binario recibido. finalBits es de 330 bits exactos.

    Parameters
    ----------
    bits : str
       string binario de largo mayor a 330.
    Returns
    -------
    str: string binario de largo 330 bits.
    :Authors:
        - Javier Ramos
    """
    firstBits = bits[:330]
    lastBits = bits[-330:]
    finalBits = xor(firstBits, lastBits)
    return finalBits
```

Figura 8: Código y comentario splitBits

7. **toBase64**.

```python
def toBase64(hashedBits: str) -> str:
    """recibe una cadena de bits y los transforma a base64,
        utiliza como apoyo la lista base64Alph.
        Por implementacion no se agregan = si faltan bloques

    Parameters
    ----------
    hashedBits : str
        cadena binaria resultante de bestHash
    Returns
    -------
    str: retorna un string en base64
    :Authors:
        - Javier Ramos
    """
    hashSplit = textwrap.wrap(hashedBits, 6)
    base64 = ""
    for i in hashSplit:
        bin_int = int(i, 2)
        base64 += base64Alph[bin_int]
    return base64
```

Figura 9: Código y comentario toBase64

8. **zaHashu**.

```python
@measure_time
def zaHashu(x: str) -> str:
    """Se encarga de llamar las funciones necesarias para
       obtener un hash "za hash"

    Parameters
    ----------
    x : str
        string a hashear
    Returns
    -------
    str: string hasheado en za hash
    :Authors:
        - Javier Ramos
    """
    bits = toBits(x)
    seed = createSeed(bits)
    hashBits = bestHash(bits, seed)
    hash = toBase64(hashBits)
    return hash
```

Figura 10: Código y comentario zaHashu

9. **md5**.

```python
@measure_time
def md5(word: str) -> str:
    """calcula el hash md5 de un string dado

    Parameters
    ----------
    word : str
        string a hashear
    Returns
    -------
    str: string hasheado en md5
    :Authors:
        - Javier Ramos
    """
    hash = hashlib.md5(word.encode())
    return hash.hexdigest()
```

Figura 11: Código y comentario md5

10. **sha256**.

```python
@measure_time
def sha256(word: str) -> str:
    """calcula el hash sha256 de un string dado

    Parameters
    ----------
    word : str
        string a hashear
    Returns
    -------
    str: string hasheado en sha256
    :Authors:
        - Javier Ramos
    """
    hash = hashlib.sha256(word.encode())
    return hash.hexdigest()
```

Figura 12: Código y comentario sha256

11. **sha1**.

```python
@measure_time
def sha1(word: str) -> str:
    """Calcula el hash sha1 de un string dado

    Parameters
    ----------
    word : str
        string a hashear
    Returns
    -------
    str: string hasheado en sha1
    :Authors:
        - Javier Ramos
    """
    hash = hashlib.sha1(word.encode())
    return hash.hexdigest()
```

Figura 13: Código y comentario sha1

12. **getBase**: Se identifica su base por medio de expresiones regulares.

```python
def getBase(word: str) -> int:
    """dado un string, calcula la base en la que esta.
        (se considero solo los caracteres ASCII imprimibles hasta el 125)

    Parameters
    ----------
    word : str
        string en alguna base
    Returns
    -------
    int: entero que representa la base
    :Authors:
        - Javier Ramos
    """
    base = 0
    baseAscii = re.compile(r"[\x21-\x2f]|[\x3a-\x40]|[\x5b-\x60]|[\x7b-\x7e]")
    if re.search("[a-z]", word):
        base += 26
    if re.search("[A-Z]", word):
        base += 26
    if re.search("[0-9]", word):
        base += 10
    if baseAscii.search(word):
        base += 32
    return base
```

Figura 14: Código y comentario getBase

13. **entropy**

```python
def entropy(word: str) -> float:
    """Dado un string, se encarga de calcular su base y el largo,
        para luego determinar la entropia del texto ingresado

    Parameters
    ----------
    word : str
        string al cual se le calculara la entropia (clave)
    Returns
    -------
    float: numero de bits de entropia.
    :Authors:
        - Javier Ramos
    """
    base = getBase(word)
    Llargo = len(word)
    Wbase = base
    Hentropia = Llargo * log(Wbase, 2)
    return Hentropia
```

Figura 15: Código y comentario entropy

3. Tabla de Comparaciones

nº Lineas	za Hash	MD5	SHA256	SHA1
1	0.0004807	0.0000220	0.0000134	0.0000138
10	0.0046409	0.0000253	0.0000187	0.0000179
20	0.0094844	0.0000367	0.0000305	0.0000258
50	0.0218470	0.0000756	0.0000617	0.0000634
100000	40.637600	0.1051705	0.1125493	0.1010864

Figura 16: Tabla tiempo de Ejecución en [s]

Entropia	za Hash	MD5	SHA256	SHA1
con Base	330	128	256	160
sin Base	360.502	165.437	330.8752	206.797

Figura 17: Tabla de entropia Output

4. Análisis

A continuación se realizara un análisis y comparación entre el hash desarrollado contra otros algoritmos de hash,para ser mas especifico md5, sha1 y sha256. Esto se realizara enfocado en 2 áreas principales, el tiempo de ejecución y la entropia que presenta la clave.

4.1. Tiempo de Ejecución

Para comenzar con el análisis del tiempo de ejecución, analizaremos el tiempo en hashear una sola linea de texto. Podemos observar en la tabla de tiempo (16), que si bien el tiempo de ejecución para “za hash” es bastante pequeño (del orden de 10^{-4}), el tiempo de ejecución de los otros algoritmos esta en el orden de 10^{-5}, lo cual es una diferencia considerable. Por otro lado si se observa las ejecuciones de lineas de forma secuencial, podemos ver que el crecimiento de “za Hash” es prácticamente lineal, debido a que cuando se hashearon 100000 lineas, el orden del tiempo aumento en 10^5, llegando a los 40 [s]. Por otro lado al observar los otros hash, no se ve que el aumento de tiempo sea de forma lineal, de hecho parece tener un comportamiento mas exponencial o logarítmico (haciendo mas pruebas de hashes grandes se podría demostrar a cual corresponde, pero probablemente son logarítmicas por temas de optimización, asi que se asume que son de este tipo), entonces aquí podemos observar la primera desventaja para ”za hash”, al tener un crecimiento lineal, versus el crecimiento logarítmico, tiene una gran desventaja dado que al procesar archivos de texto muy grandes, se demorara mucho, esto viene por la implementación, dado que la recursividad que se implemento para asegurar el tamaño del hash podría tomar mucho tiempo en resolverse.

4.2. Entropia

Cuando se analiza la entropía, debemos recordar que esta es la cantidad de variación que presentan los caracteres de un string, por esto en la tabla 17, se muestran 2 cálculos de entropía. La entropía **"con Base"** podríamos decir que es la entropía real del hash generado, dado que se calculo conociendo el valor de la base, los cuales son *"za Hash"*: base 64, *"md5/sha1/sha256"*: base 16 (Hexadecimal).
Por otra parte tenemos la **"sin Base"** la cual corresponde a la entropía calculada sin saber el valor exacto de la base, de esta forma las bases cambian, aumentando su valor en todos los casos, siendo para *"za Hash"*: base 94, *"md5/sha1/sha256"*: base 36.
Ahora si observamos los resultados, podemos ver que en términos de entropía, en ambos casos **"za Hash"** presenta mayor cantidad de variación, siendo en ambos casos seguido por **SHA256**. Por otra parte podemos observar que el hash que presenta menor variación es md5 en ambos casos.

5. Conclusiones

Podemos concluir que se logro construir un buen algoritmo de hash, esto debido a que se tomaron muchos factores importantes en consideración al momento de construirlo, como lo son el efecto avalancha y las cadenas largas de caracteres, además de considerar el obtener entropía de factores aleatorios, como lo son el input y el tiempo de la función time(). Esto ultimo se puede ver también en la alta entropía que presenta el hash resultante , donde supero a todos los otros hash con los que fue comparado. Por otra parte, si bien el proceso no es lento, no es el algoritmo mas rápido para hashear, esto debido a que los otros algoritmos de hash fueron aproximadamente un 99.8 % mas rápido en procesar las 100000 lineas del archivo rockyou. Por esto el algoritmo podría ser una buena opcion si el tiempo de ejecución no es una limitante, pero si se debe procesar mas rápido, seria una mejor opción sha256 ya que presenta mayor entropía y menores tiempos.