Ingeniería de Sistemas y Computación
Facultad de Ingeniería
Universidad de los Andes

Caso III
Seguridad

Ortiz Almanza David Santiago (201913600)
Tamaio Ramírez Thais (202022213)
Latorre Munar Santiago (202111851)

Tabla de contenido

Introducción

En la actual era digital, garantizar la seguridad y protección de la información es un aspecto crítico para cualquier sistema informático. La confidencialidad y la integridad de los datos constituyen dos pilares fundamentales en este ámbito. Teniendo esto en cuenta, el presente informe se enfoca en el estudio y análisis de los códigos criptográficos de hash, específicamente, en su aplicación para el manejo de la confidencialidad e integridad de las contraseñas en los sistemas operativos Linux y Windows.

El objetivo principal de este informe es obtener una comprensión profunda del alcance y las limitaciones de los códigos criptográficos de hash en relación con la protección de contraseñas. Mediante la construcción de un prototipo a escala, se evaluará el esquema de manejo de confidencialidad de contraseñas en los mencionados sistemas operativos. Esta herramienta permitirá examinar la efectividad de los mecanismos de protección frente a ataques de fuerza bruta, como los realizados por los denominados "Password Crackers".

Para alcanzar estos objetivos, se diseñará un programa en Java que implemente diversos algoritmos de generación de códigos criptográficos de hash y permita explorar un espacio de búsqueda de posibles contraseñas. El programa aceptará como entrada un algoritmo de hash, el código criptográfico de una contraseña, una sal y el número de threads a utilizar. A través de este prototipo, se llevarán a cabo pruebas exhaustivas con el fin de determinar la solidez y seguridad de los mecanismos de protección de contraseñas en los sistemas operativos en estudio.

En el informe subsiguiente, se detallarán la metodología empleada, los resultados obtenidos y las conclusiones derivadas de esta investigación.

Descripción del programa desarrollado

Diagrama de clases

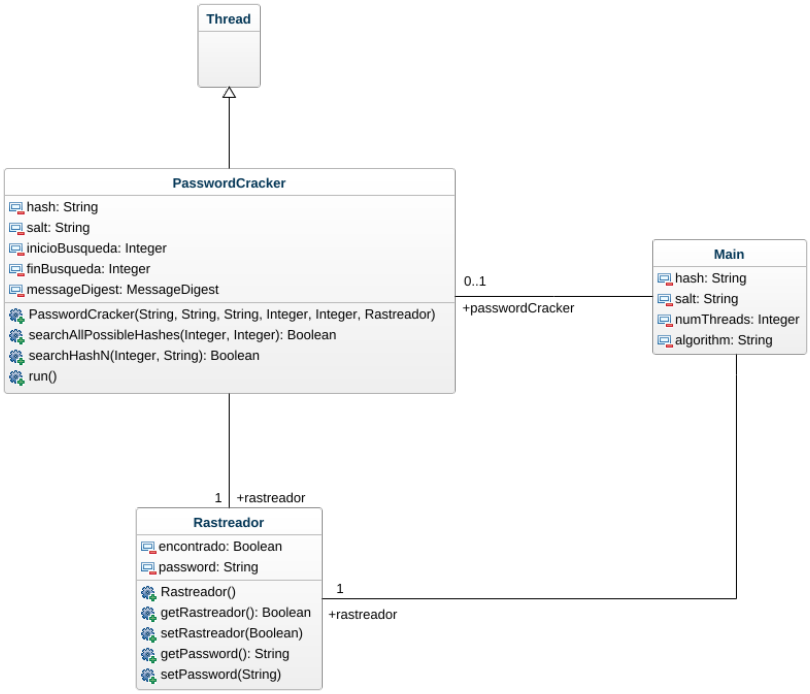

Figura 1. Diagrama de clases

Descripción del algoritmo usado para explorar el espacio de búsqueda

```java
public boolean searchAllPossibleHashes(int inicioBusqueda, int finBusqueda) {

        boolean found = false;

        for (int i = inicioBusqueda; !rastreador.getRastreador() && i <= finBusqueda; i++) {
            if(searchHashN(i, "")){
                rastreador.setRastreador(true);
                found = true;
            }
        }
        System.out.println(i);
        return found;
    }
```

Figura 2. Algoritmo para explorar todo el espacio de búsqueda

```java
public boolean searchHashN(int pLength, String pPrefix) {
        boolean found = false;
        if (pLength == 0) {
            byte[] bytes = md.digest((pPrefix + salt).getBytes());
            StringBuilder sb = new StringBuilder();
            for (byte b : bytes) {
                sb.append(String.format("%02x", b));
            }
            if (sb.toString().equals(hash)) {
                rastreador.setPassword(pPrefix);
                found = true;
            }
            i+=1;
        } else {
            int i = 0;
            while (!found && i < ALPHABET.length() && !rastreador.getRastreador()) {
                found = searchHashN(pLength - 1, pPrefix + ALPHABET.charAt(i));
                i++;
            }
        }
        return found;
    }
```

Figura 3. Algoritmo para explorar todas las cadenas de longitud pLength

El programa en Java desarrollado consta principalmente de dos métodos que realizan una búsqueda de fuerza bruta para encontrar una contraseña que coincida con un valor hash dado. Estos dos métodos son searchAllPossibleHashes () y searchHashN().

- **Método searchAllPossibleHashes (inicioBusqueda, finBusqueda):**

  Este método actúa como un controlador para el proceso de búsqueda de la contraseña. Acepta dos argumentos: inicioBusqueda y finBusqueda, que representan las longitudes iniciales y finales de las contraseñas a probar, respectivamente.

  Dentro de este método, se itera desde la longitud de la contraseña inicioBusqueda hasta la longitud de la contraseña finBusqueda (ambas inclusive). En cada iteración, se llama al método searchHashN () con la longitud de la contraseña actual (i) y una cadena vacía como prefijo.

  Si searchHashN () devuelve true, se establece rastreador (objeto compartido para establecer sincronización entre Threads) en true y se actualiza la variable 'found' a true. La búsqueda continúa hasta que se encuentra la contraseña o se alcanza la longitud final.

- **Método searchHashN (pLength, pPrefix):**

Este método realiza una búsqueda recursiva de todas las posibles combinaciones de contraseñas con una longitud dada (pLength) y un prefijo dado (pPrefix).

Si pLength es igual a 0, significa que se ha construido una contraseña completa utilizando el prefijo. Se genera el hash de la contraseña (con la sal agregada) y se compara con el hash objetivo. Si coincide, se actualiza el rastreador con la contraseña encontrada y se establece 'found' en true.

Si pLength no es 0, se itera a través de todos los caracteres del alfabeto y se realiza una llamada recursiva a searchHashN () con pLength-1 y el prefijo actualizado. La búsqueda se detiene si se encuentra la contraseña o si se ha buscado en todo el alfabeto sin éxito.

Casos Considerados

Se analizaron diversos escenarios generados a partir de variaciones en los datos de entrada. Estas variaciones incluyen la longitud de las cadenas de entrada, el número de hilos (threads) utilizados, las sales empleadas, los algoritmos de encriptación y los mejores / peores casos en cada situación. Los detalles de los casos considerados son los siguientes:

Longitud de la cadena de entrada: Se evaluaron cadenas de longitudes 1, 2, 3, 4, 5, 6 y 7.

Número de hilos (threads): Se examinaron casos con 1 y 2 hilos.

Sales: En cada situación, se utilizaron dos sales distintas, "AB" y "YZ".

Algoritmos de encriptación: Para cada escenario, se emplearon los algoritmos SHA-256 y SHA-512.

Códigos de hash: En cada caso, se probaron dos códigos de hash diferentes, el mejor y el peor dentro de su espacio de búsqueda.

Resultados Obtenidos

A continuación, se detallan los resultados obtenidos en los casos de prueba especificados en la sección anterior a través de tablas y gráficas.

Tablas de datos

Len	Input	Tiempo (ms)							
		SHA-256				SHA-512			
		1 Thread		2 Threads		1 Thread		2 Threads	
		ab	yz	ab	yz	ab	yz	ab	yz
1	a	3	1	15	0	13	1	2	1
	z	198	4	368	20	20	4	24	3
2	aa	2	1	5	1	5	2	9	3
	zz	43	20	62	29	60	28	43	29
3	aaa	13	10	18	11	27	20	21	24
	zzz	294	262	308	277	498	494	562	528
4	aaaa	259	257	1	1	493	493	1	0
	zzzz	6.797	6.725	6.493	6.465	13.126	13.054	12.617	12.581
5	aaaaa	6.716	6.709	6.458	6.473	13.043	13.141	12.581	12.625
	zzzzz	170.022	170.472	169.618	169.947	330.893	331.239	330.893	330.153
6	aaaaaa	169.909	170.164	169.705	169.768	329.585	331.581	329.362	328.565
	zzzzzz	4.524.533	4.516.376	4.510.692	4.537.640	8.811.935	8.851.755	8.814.444	8.801.532
7	aaaaaaa	4.524.733	4.516.689	4.510.734	4.537.720	8.812.734	8.853.677	8.815.231	8.802.324
	zzzzzzz	117.649.050	117.425.941	117.278.055	117.978.726	146.708.365	230.154.844	228.692.207	228.878.728

Tabla 1. Tiempo en ms en cada uno de los casos considerados

Comparación de tiempo promedio (mejor/peor caso) para cada escenario

- **1 Thread**

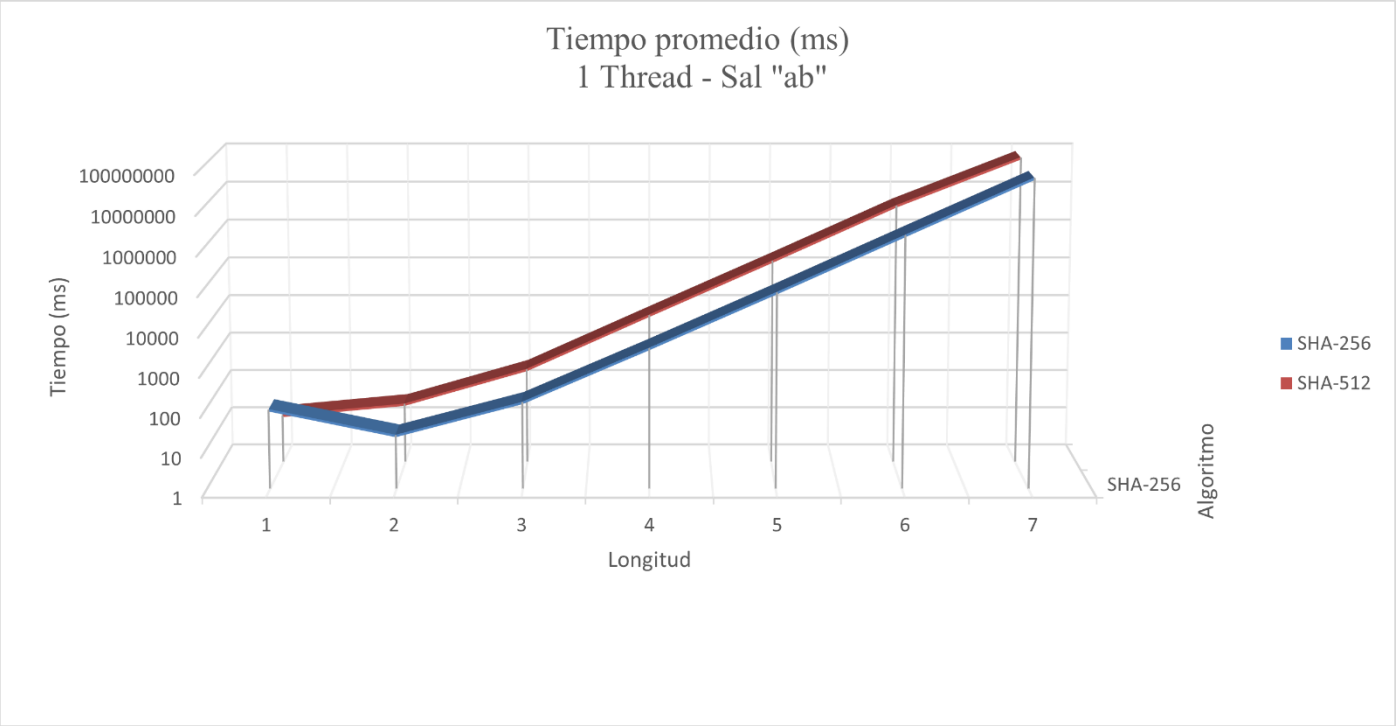

Figura 4. Tiempo promedio (mejor y peor caso) en milisegundos (escala logarítmica) variando la longitud y el algoritmo
(1 Thread – Sal "ab")

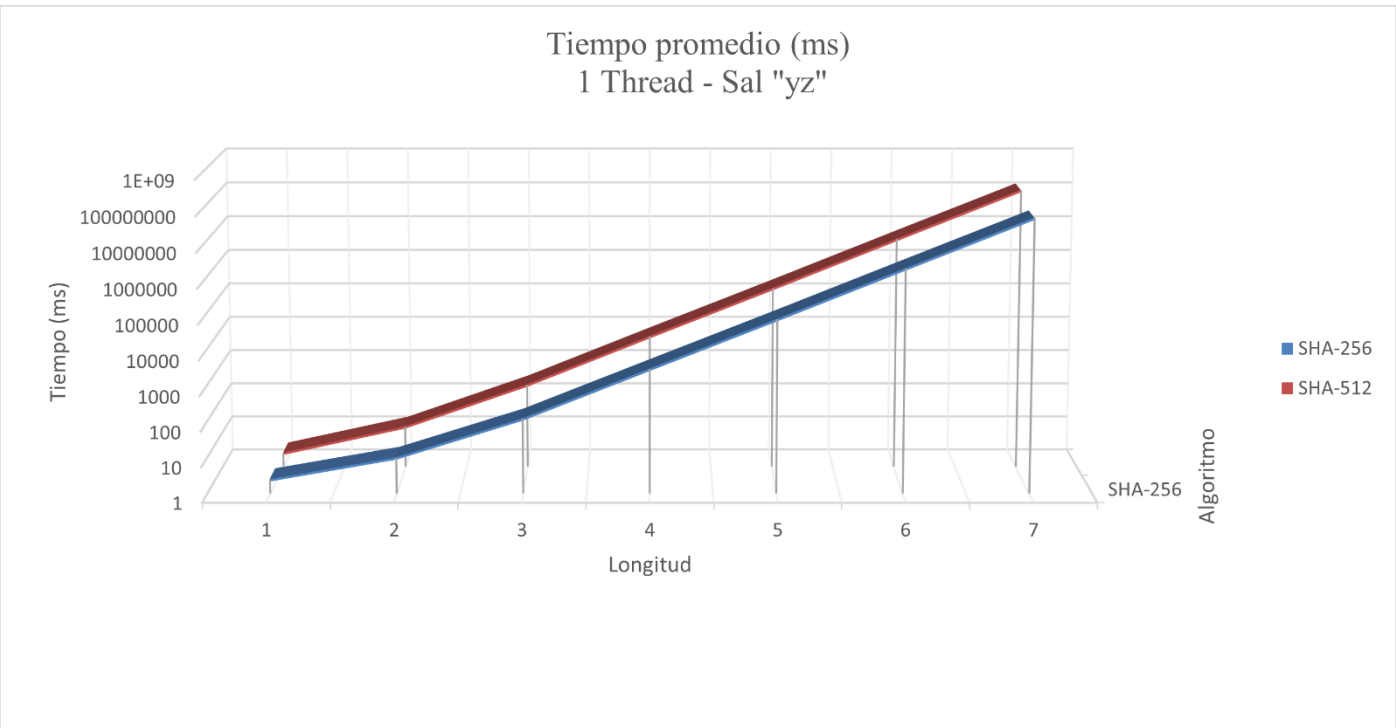

Figura 5. Tiempo promedio en milisegundos (escala logarítmica) variando la longitud y el algoritmo
(1 Thread – Sal "yz")

- **2 Threads**

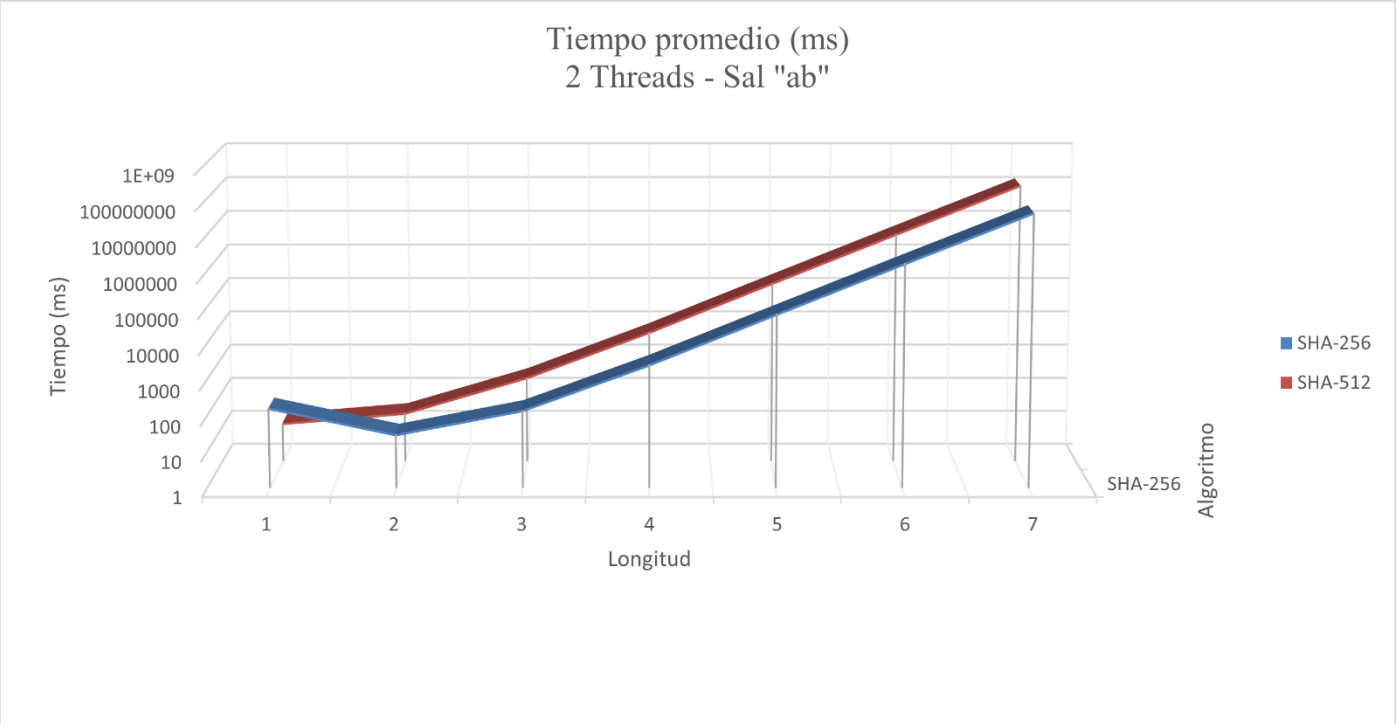

Figura 6. Tiempo promedio en milisegundos (escala logarítmica) variando la longitud y el algoritmo (2 Threads – Sal "ab")

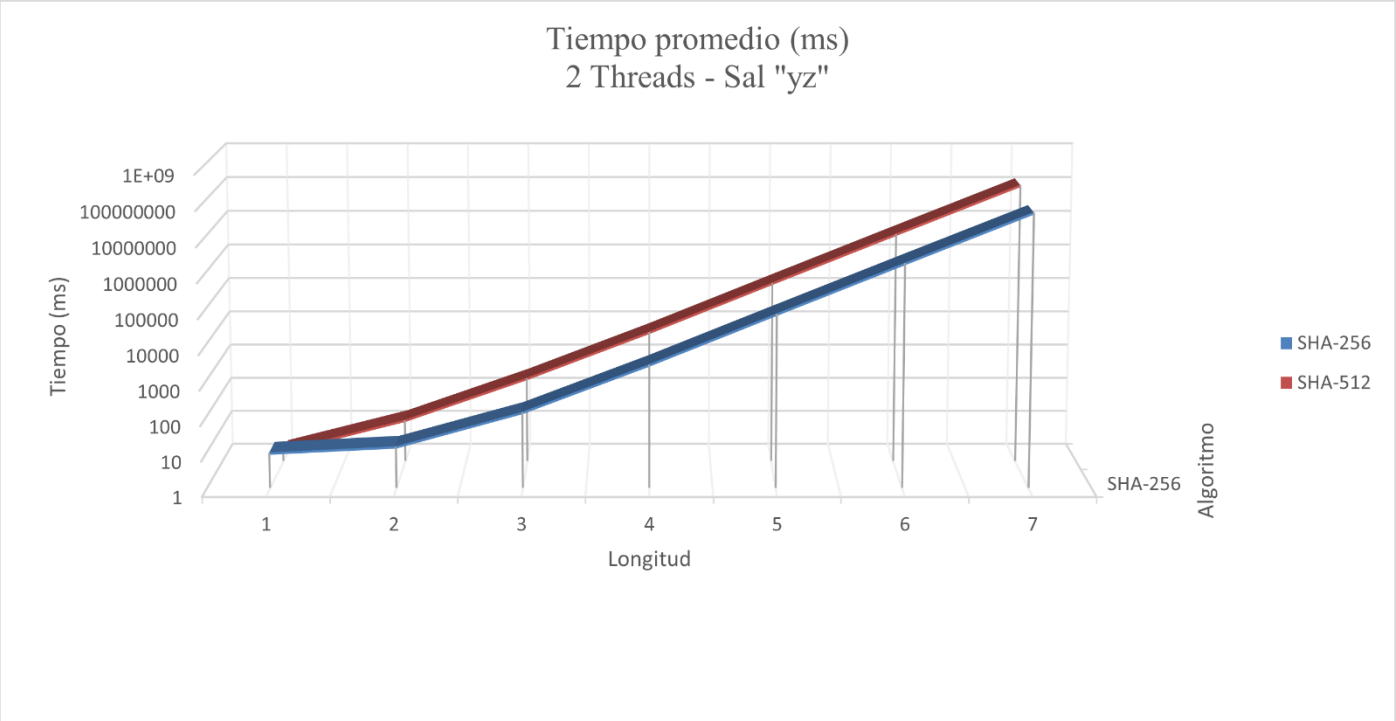

Figura 7. Tiempo promedio en milisegundos (escala logarítmica) variando la longitud y el algoritmo (2 Threads – Sal "yz")

Comparación entre mejor y peor caso por escenario

- **1 Thread**

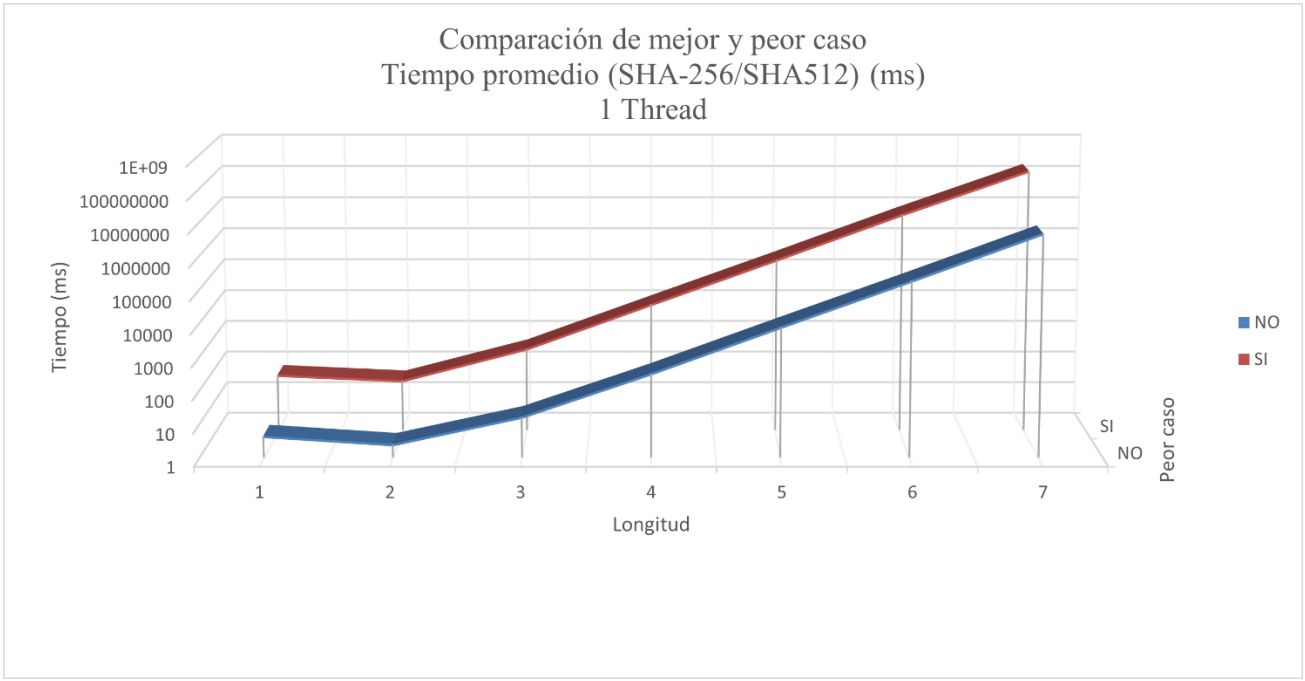

Figura 8. Tiempo promedio en milisegundos (escala logarítmica) variando la longitud para el mejor y peor caso

- **2 Threads**

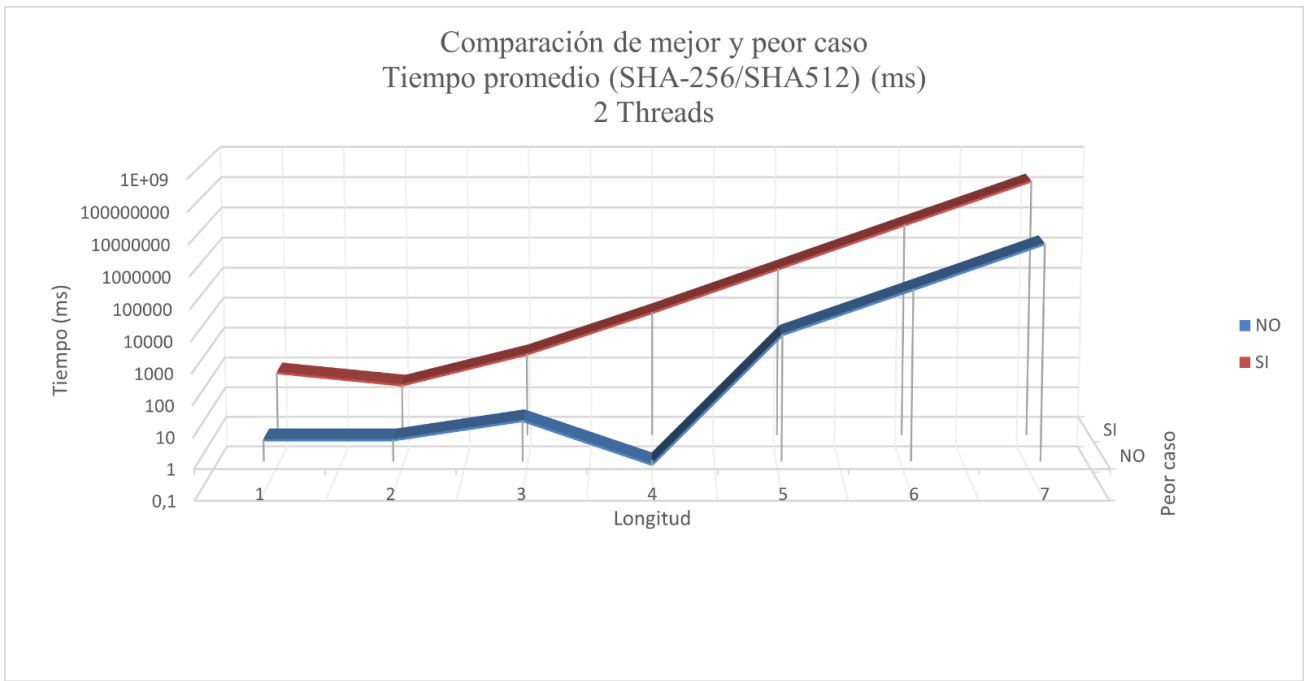

Figura 9. Tiempo promedio en milisegundos (escala logarítmica) variando la longitud para el mejor y peor caso

Interpretación de Resultados

Los tiempos de descifrado obtenidos para las diferentes contraseñas revelan información valiosa acerca de cómo diversos factores pueden afectar el tiempo necesario para descifrar una contraseña.

En cuanto al algoritmo utilizado, se observa que, en general, el algoritmo SHA-512 es más lento que el algoritmo SHA-256 para cifrar las contraseñas. Esto se debe a que SHA-512 utiliza una longitud de bloque más grande y un espacio de hash más grande (512 bits) en comparación con SHA-256 (256 bits), lo que significa que requiere más operaciones y tiempo de procesamiento.

En cuanto a la cantidad de threads utilizados, se puede observar que en algunas ocasiones se obtienen mejores tiempos con 2 threads para una misma prueba, mientras que en otras no hay una diferencia significativa. Esto se debe a que la utilización de múltiples threads para el procesamiento en paralelo puede mejorar el rendimiento, pero esto depende de la complejidad de la tarea y de cómo se divide el trabajo entre los threads.

Es importante mencionar que en algunas ocasiones, los tiempos de descifrado con dos threads son extremadamente bajos. Esto ocurre cuando la contraseña tiene una longitud de 4 caracteres, ya que el segundo thread comienza a descifrar contraseñas a partir de los 4 caracteres. En este caso, si la contraseña tiene exactamente 4 caracteres, el procedimiento de descifrado es casi instantáneo. Es por eso que en la tabla de tiempos de descifrado, se puede observar que algunas contraseñas con longitud 4 tienen tiempos de descifrado de solo unos pocos milisegundos cuando se usan dos threads. Esto se debe a que el primer thread descifra contraseñas de longitud 1 a 3, mientras que el segundo thread descifra contraseñas de longitud 4 a 7. Por lo tanto, si la contraseña es de longitud 1 o 4, se encontrará más rápidamente que otras longitudes cuando se usan dos threads

En cuanto a la sal utilizada, se puede ver que en general no hay una gran diferencia en los tiempos de descifrado entre las contraseñas que tienen sal y las que no la tienen. Esto se debe a que la sal se agrega al inicio de la contraseña antes del proceso de hashing y no afecta la complejidad del algoritmo utilizado para descifrar la contraseña.

En cuanto a la longitud de la contraseña, se puede observar que el tiempo de descifrado aumenta significativamente a medida que se incrementa la longitud de la contraseña, específicamente crece de manera exponencial. Esto se debe a que el número de combinaciones posibles aumenta exponencialmente con la longitud de la contraseña, lo que hace que el proceso de descifrado sea más complejo y, por lo tanto, más lento.

Además, es importante destacar que el input de la contraseña también puede afectar los tiempos de descifrado. Se puede observar que las contraseñas que contienen la letra 'z' requieren más tiempo de descifrado en comparación con las que no la contienen, ya que todas las combinaciones anteriores al 'z' tienen que ser probadas antes de llegar a esta letra.

Finalmente, al correr todos los casos se encontró que la contraseña más rápida de descifrar con el algoritmo SHA-256 y dos threads fue la contraseña 'a'. Esto se debe a que es la contraseña más corta y es el primer caso que se evalúa en el rango de contraseñas, sumado a que el algoritmo SHA-256 es más rápido que el SHA-512 y los dos threads lo hacen más eficiente. Por otro lado, el caso más demorado fue la contraseña 'zzzzzzz', utilizando el algoritmo SHA-512 y un solo thread. En este caso, se deben recorrer todas las posibles combinaciones dentro del rango de contraseñas para encontrarla, sumado al hecho de que el algoritmo SHA-512 es más lento y un solo thread hace la ejecución más lenta en este caso.

Ciclos de Procesador

Para estimar el número promedio de ciclos de procesador necesarios para generar y evaluar un valor con el fin de determinar si produce el código buscado, utilizaremos los tiempos obtenidos en la Tabla 1.

- **SHA-256:**

  En el peor caso, con una longitud de 6 caracteres y utilizando SHA-256 con un solo hilo, el programa tardó 4,524,533 milisegundos. Para encontrar esta contraseña, se evaluaron todos los casos desde longitud 1 hasta longitud 6. Por lo tanto, el número total de casos evaluados y generados se calcularía de la siguiente manera:

  $$Número\ de\ casos = (26^1 + 26^2 + \cdots + 26^6)$$

  Esto es:

  $$Número\ de\ casos = 321.272.406\ casos$$

  Ahora, podemos calcular el número de casos por milisegundo:

  $$\frac{321.272.406\ casos}{4.524.533\ ms} = 71\ \frac{casos}{ms}$$

  Así, por segundo:

  $$71000\ \frac{casos}{s}$$

  Entonces, podemos hallar el tiempo promedio que tarda cada caso en segundos:

  $$1\ caso \rightarrow 1{,}408 * 10^{-5}\ segundos$$

  Dado que el procesador utilizado tiene una frecuencia de 1.6 GHz, esto implica:

  $$1.6 * 2^{30}\ \frac{ciclos\ de\ procesador}{segundo}$$

  Por lo tanto, podemos calcular el número promedio de ciclos de procesador para generar y evaluar 1 caso:

  $$1{,}408 * 10^{-5}\ segundos * \ 1.6 * 2^{30}\ \frac{ciclos\ de\ procesador}{segundo} = 24.195\ ciclos\ de\ procesador$$

  Así, en promedio se necesitan 24,195 ciclos de procesador para generar y evaluar 1 caso y determinar si produce el código buscado.

- **SHA-512:**

  Podemos ver que para el peor caso de longitud 6 utilizando SHA-512 y solo un Thread le tomó al programa 8.811.935 milisegundos. Para encontrar esta contraseña se tuvo que evaluar todos los casos desde longitud 1 hasta longitud 6, así, el número de casos totales generados y evaluados se calcularía de la siguiente manera:

  $$Número\ de\ casos = (26^1 + 26^2 + \cdots + 26^6)$$

  Esto es:

  $$Número\ de\ casos = 321.272.406\ casos$$

  Ahora, podemos encontrar el número de casos por milisegundo:

  $$\frac{321.272.406\ casos}{8.811.935\ ms} = 36.5\ \frac{casos}{ms}$$

  Así, por segundo:

  $$36500\ \frac{casos}{s}$$

  Con esto podemos encontrar en promedio cuánto tiempo se demora cada caso en segundos:

  $$1\ caso \rightarrow 2{,}742 * 10^{-5}\ segundos$$

  Ahora bien, el procesador utilizado es de 1.6 GHz, esto quiere decir:

  $$1.6 * 2^{30}\ \frac{ciclos\ de\ procesador}{segundo}$$

  Con esto, podemos calcular el número de ciclos de procesador, en promedio, de generar y evaluar 1 caso:

  $$2{,}742 * 10^{-5}\ segundos * \ 1.6 * 2^{30}\ \frac{ciclos\ de\ procesador}{segundo} = 47.121\ ciclos\ de\ procesador$$

  Así, el número de ciclos de procesador necesarios para generar y evaluar 1 caso para determinar si genera o no el código buscado serían 47.121.

Proyección de tiempo para 3 escenarios

Contraseñas donde cada carácter puede ser mayúscula, minúscula, número o uno de los siguientes caracteres especiales: .:,;!?(%)\+-/*{}, la sal es una secuencia de 16 bits.

Lo primero que se debe hacer es encontrar la cantidad total de elementos del conjunto de caracteres, así:

- Letras mayúsculas: Hay 26 letras mayúsculas en el alfabeto (A-Z).
- Letras minúsculas: Hay 26 letras minúsculas en el alfabeto (a-z).
- Números: Hay 10 números (0-9).
- Caracteres especiales: Hay 16 caracteres especiales en la lista (. , : ; ! ? (%) \ + - / * { }).

Ahora, sumamos todos estos conteos: 26 (mayúsculas) + 26 (minúsculas) + 10 (números) + 16 (caracteres especiales) = 78 elementos en total en este alfabeto.

Aclaración: En los cálculos presentados a continuación se sigue asumiendo que la sal es conocida, en caso, de que la sal fuera desconocida, entonces habría que multiplicar la cantidad de casos posibles por 2^{16} que sería la cantidad de sales posibles. Esto significaría que el tiempo sería 65.536 veces lo calculado.

- **Contraseñas de 8 caracteres:**

  Asumiendo que se sabe que la contraseña tiene 8 caracteres y considerando el peor caso, entonces, se tienen que generar y probar todos los casos del espacio, esto es:

$$Número\ de\ casos = 78^8$$

$$Número\ de\ casos = 1.370.114.370.683.136$$

  Ahora, para calcular el tiempo que tomaría generar y probar estos casos se hace uso del número de ciclos de procesador necesarios calculados en la sección anterior por cada uno de los algoritmos.

  - **SHA-256:**

$$\text{Tiempo (s)} = \text{Número de casos} * \frac{\text{Ciclos de procesador}}{Caso} * \frac{1\ segundo}{Ciclos\ de\ procesador}$$

$$= 1.370.114.370.683.136\ casos * \frac{24.195\ \text{Ciclos de procesador}}{Caso} * \frac{1\ segundo}{1.6 * 2^{30}\ \ Ciclos\ de\ procesador}$$

$$= 19.295.791.396\ segundos$$

$$= 611.86\ años^*$$

**** Leer la aclaración al inicio de la sección relacionada con la sal***

  - **SHA-512**

$$\text{Tiempo (s)} = \text{Número de casos} * \frac{\text{Ciclos de procesador}}{Caso} * \frac{1\ segundo}{Ciclos\ de\ procesador}$$

$$= 1.370.114.370.683.136\ casos * \frac{47.121\ \text{Ciclos de procesador}}{Caso} * \frac{1\ segundo}{1.6 * 2^{30}\ \ Ciclos\ de\ procesador}$$

$$= 37.579.540.664\ segundos$$

$$= 1191.64\ años^*$$

**** Leer la aclaración al inicio de la sección relacionada con la sal***

- **Contraseñas de 10 caracteres, cada carácter puede ser mayúscula, minúscula, número o uno de los siguientes caracteres especiales: .:,;!?(%)\+-/*{}, la sal es una secuencia de 16 bits**

  Asumiendo que se sabe que la contraseña tiene 10 caracteres y considerando el peor caso, entonces, se tienen que generar y probar todos los casos del espacio, esto es:

$$Número\ de\ casos = 78^{10}$$

$$Número\ de\ casos = 8.335.775.831.236.199.424$$

  Ahora, para calcular el tiempo que tomaría generar y probar estos casos se hace uso del número de ciclos de procesador necesarios calculados en la sección anterior por cada uno de los algoritmos.

  - **SHA-256:**

$$\text{Tiempo (s)} = \text{Número de casos} * \frac{\text{Ciclos de procesador}}{Caso} * \frac{1\ segundo}{Ciclos\ de\ procesador}$$

$$= 8.335.775.831.236.199.424\ casos * \frac{24.195\ \text{Ciclos de procesador}}{Caso} * \frac{1\ segundo}{1.6 * 2^{30}\ \ Ciclos\ de\ procesador}$$

$$= 117.395.594.853.884{,}45\ segundos$$

$$= 3.722.589{,}89\ años^{*}$$

  **** Leer la aclaración al inicio de la sección relacionada con la sal***

  - **SHA-512**

$$\text{Tiempo (s)} = \text{Número de casos} * \frac{\text{Ciclos de procesador}}{Caso} * \frac{1\ segundo}{Ciclos\ de\ procesador}$$

$$= 8.335.775.831.236.199.424\ \ casos * \frac{47.121\ \text{Ciclos de procesador}}{Caso} * \frac{1\ segundo}{1.6 * 2^{30}\ \ Ciclos\ de\ procesador}$$

$$= 228.633.925.402.351{,}27\ segundos$$

$$= 7.249.934{,}21\ \ años^{*}$$

  **** Leer la aclaración al inicio de la sección relacionada con la sal***

- **Contraseñas de 12 caracteres, cada carácter puede ser mayúscula, minúscula, número o uno de los siguientes caracteres especiales: .:,;!?(%)\+-/*{}, la sal es una secuencia de 16 bits**

Asumiendo que se sabe que la contraseña tiene 12 caracteres y considerando el peor caso, entonces, se tienen que generar y probar todos los casos del espacio, esto es:

$$Número\ de\ casos = 78^{12}$$

$$Número\ de\ casos = 50.714.860.157.241.037.295.616$$

Ahora, para calcular el tiempo que tomaría generar y probar estos casos se hace uso del número de ciclos de procesador necesarios calculados en la sección anterior por cada uno de los algoritmos.

- **SHA-256:**

$$\text{Tiempo (s)} = \text{Número de casos} * \frac{\text{Ciclos de procesador}}{Caso} * \frac{1\ segundo}{Ciclos\ de\ procesador}$$

$$= 50.714.860.157.241.037.295.616\ casos * \frac{24.195\ \text{Ciclos de procesador}}{Caso} * \frac{1\ segundo}{1.6 * 2^{30}\ \ Ciclos\ de\ procesador}$$

$$= 714.234.799.091.032.995{,}70\ segundos$$

$$= 22.648.236.906{,}74\ \ años^{*}$$

**** Leer la aclaración al inicio de la sección relacionada con la sal***

- **SHA-512**

$$\text{Tiempo (s)} = \text{Número de casos} * \frac{\text{Ciclos de procesador}}{Caso} * \frac{1\ segundo}{Ciclos\ de\ procesador}$$

$$= 50.714.860.157.241.037.295.616\ casos * \frac{47.121\ \text{Ciclos de procesador}}{Caso} * \frac{1\ segundo}{1.6 * 2^{30}\ \ Ciclos\ de\ procesador}$$

$$= 1.391.008.802.147.905.178{,}37\ segundos$$

$$= 44.108.599.763{,}69\ \ años^{*}$$

**** Leer la aclaración al inicio de la sección relacionada con la sal***

Análisis y Entendimiento del Problema

1. Algoritmos de generación de código criptográfico de Hash

Los algoritmos de SHA (Secure Hash Algorithm) son funciones de Hash criptográficas que permiten la creación de cadenas aleatorias, teóricamente distintas entre sí, que facilitan seguir un registro de cambios en la huella digital. El uso de SHA es particularmente útil en campos de la seguridad tales como la creación de cuentas asociadas a una contraseña de un único dueño o claves de desencriptación de ficheros, pues dada su capacidad de no reflexividad, es casi imposible hallar una cadena de salida que retorne el mismo mensaje que la de origen.

A lo largo del tiempo, la implementación práctica del algoritmo ha ido evolucionando según las vulnerabilidades que se encontraron en versiones previas. Esto se evidenció claramente con el SHA-1, una función muy utilizada en su tiempo que vio su seguridad comprometida en el 2004, cuando se demostró que era posible romperla en 263 operaciones, que, aunque es un número muy elevado de operaciones, es computacionalmente posible en la actualidad. Debido a esto, se implementó una nueva versión, SHA-2, que cerró esta brecha de seguridad al cambiar la construcción de la cadena y aumentar el tamaño de salida de 160 a 256. Esto hizo que tanto los métodos utilizados para romper SHA-1 como la fuerza bruta se volvieran completamente inútiles frente a SHA-2.

2. Proceso de Mining

La tecnología Bitcoin utiliza un ‘libro de contabilidad’ público donde se encuentran guardados todos los registros de cada transacción realizada mediante Bitcoin, denominado la Blockchain. Esto tiene como ventaja la transparencia, ya que cualquiera puede leer este libro y revisar la información que se encuentra en este; pero por otro lado genera dificultades en la seguridad, pues teóricamente cualquiera podría agregar, eliminar o modificar las transacciones que se realizan en la Blockchain, y es por esto por lo que se hace necesario el uso de funciones de Hash criptográficas.

se quiere agregar un bloque de transacciones a la Blockchain, todos los mineros compiten para realizar una verificación con el fin de registrar el bloque universalmente, esta verificación consiste en una prueba criptográfica que tiene como objetivo encontrar un hash específico utilizando como inputs el bloque de transacciones y un número aleatorio. Llevar a cabo esta tarea supone un consumo computacionalmente muy elevado, y dado que para modificar y verificar el bloque en la Blockchain es necesario hallar el número aleatorio antes que los otros mineros, resulta muy improbable que el ataque sea efectivo. Aun si esto llegase a ocurrir, para seguir garantizando la validez del bloque modificado, el minero debe seguir llegando al hash correcto antes que los demás, pues cuando se modifica un bloque, se forma dos versiones de la Blockchain, una con la modificación y la otra sin la modificación; para determinar la real se elige aquella que tenga una mayor cantidad de bloques validados. Dado que el minero está compitiendo contra todos los demás en el mundo, resulta imposible que consiga que su versión de la Blockchain tenga más bloques validados que aquella que no tuvo modificaciones maliciosas.

En conclusión, el proceso de mining, y la Blockchain en general, es bastante certero gracias a los mecanismos de seguridad fundamentados en algoritmos criptográficos con los que cuenta.

3. Rainbow Tables

Las Rainbow Tables son bases de datos que contienen Strings (usualmente las contraseñas comunes o más utilizadas) y una cadena de valores de hash que resultan de un proceso de aplicación de la función de Hash, reducción de la cadena resultado y repetición. Una vez se cuenta con la contraseña encriptada se compara con cada una de las cadenas hasta encontrar una similitud, y posteriormente se va al inicio de la cadena para obtener la contraseña en texto plano. Lo que resulta particularmente efectivo de las Rainbow Tables (en sistemas que no utilizan mecanismos de prevención) es que no importa el String en texto plano: si su hash es igual al que estamos buscando la autenticación aceptará esta String aunque no sea la contraseña correcta. Esto quiere decir que aun sin tener que calcular el hash de todas las combinaciones de contraseñas, es posible hallar un hash que permita autenticarse.

Debido a esto, se llegó a una solución simple, pero muy efectiva: adicionar caracteres aleatorios a las contraseñas y luego realizar el hash, lo que garantiza que no existan dos contraseñas que compartan el mismo hash. Esto inutiliza completamente a las Rainbow Tables, pues sin conocimiento de la sal es imposible precalcular los hashes correspondientes, y aun si se tuviese la contraseña calculada en la base de datos, el hash es completamente diferente gracias a la adición de la sal.

Referencias

- D. Eastlake,P. Jones. *US Secure Hash Algorithm 1 (SHA1)*
- https://web.archive.org/web/20070118021843/http://en.epochtimes.com/news/7-1-11/50336.html
- https://nordvpn.com/es/blog/what-is-rainbow-table-attack/
- https://www.geeksforgeeks.org/understanding-rainbow-table-attack/
- Udacity. *Rainbow Tables - Web Development* . https://www.youtube.com/watch?v=SOV0AeHuHaQ
- Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System. chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://bitcoin.org/bitcoin.pdf*