Caso 3 – Infraestructura computacional

Samuel J. Freire Tarazona - 202111460
German L. Moreno Cainaba - 202116701
Sofia Velasquez Marin - 202113334

Tabla de contenido

Implementación del Prototipo

1.1 Programa en java

En primer lugar, el *main* se encarga de pedirle al usuario los datos necesarios para el correcto funcionamiento del programa, estos son: el algoritmo (SHA512 o SHA256), el hash del código concatenado con la sal a descifrar, la sal y por último el número de threads (1 o 2). Una vez tenga esos datos, dependiendo el número de threads establecido, creamos el *PasswordCracker*, el cual tiene toda la lógica para poder descifrar el hash recibido por parámetro. El *PasswordCraker* verifica todas las posibilidades de strings de longitud 1 a 7, hasta que encuentre el que pertenece al hash recibido. Si se establece que el programa debe correr con 2 threads, el espacio de búsqueda se dividió de la siguiente manera: un thread revisa los strings que comienzan de la ‘a’ a la ‘m’, y el otro thread, revisa los strings que comienzan de la ‘n’ a la ‘z’.

El *PasswordCraker*, recibe el algoritmo, la sal, el hash como un array de bits, los caracteres que puede tener el código a descifrar, los caracteres de inicio a revisar (‘a’ – ‘m’ o ‘n’ – ‘z’ o ‘a’ – ‘z’), el total de posibles combinaciones a revisar (en el caso de 1 thread 8353082582 y en caso de 2 threads 4176541291), cuantos threads se van a ejecutar y por

último recibe el tiempo de inicio de la ejecución. Esta clase extiende a *Thread*, y se ejecuta mientras no se haya encontrado el código correspondiente al hash recibido o mientras no haya revisado todas las posibles combinaciones.

```
public void run() {

    int times = 0;

    while (!Main.found && times < total) {
        times++;
        password = Combinacion();

        String temp = new String(password).trim();

        byte[] hashedComb = md.digest((temp + sal).getBytes(StandardCharsets.UTF_8));

        if (Arrays.equals(passwordHash, hashedComb)) {
            Main.found = true;
            System.out.println("\nThe password is : " + temp);
            System.out.println("Password + salt: " + temp + sal);
            long endTime = System.currentTimeMillis();
            System.out.println("Time elapsed: " + (endTime - startTime) + " ms");
        }
    }
}
```

Figura 1. Método run de PasswordCracker

En cada iteración establecemos password como la siguiente combinación, es decir si acabamos de revisar ‘a’, ahora password es ‘b’ y así sucesivamente. Dicha variable es un array de chars que su máximo tamaño es 7. Para generar la siguiente combinación utilizamos el método *nextCombination.*

```
public void nextCombination() {
    if (password[0] == charsSide.get(charsSide.size() - 1)) {
        password[0] = charsSide.get(index:0);
        boolean incremented = false;
        int i = 0;
        for (int j = i + 1; j < password.length && !incremented; j++) {
            if (password[j] == chars.get(chars.size() - 1)) {
                password[j] = chars.get(index:0);
            } else {
                password[j] = chars.get(chars.indexOf(password[j]) + 1);
                incremented = true;
            }
        }

    } else {
        password[0] = (char) charsSide.get(charsSide.indexOf(password[0]) + 1 % charsSide.size());
    }
}
```

Figura 2. Función para dar la siguiente combinación

El método nextCombination revisa si la posición 0 de *password*, es decir, la primera letra de la combinación esta en su valor máximo (‘z’ si es 1 thread, ‘m’ o ‘z’ si son 2 theads). En caso de que esta condición sea verdadera, se incrementa el siguiente carácter, pero si este se encuentra en su máximo valor, en este caso siempre es ‘z’, intenta aumentar el siguiente y así sucesivamente hasta poder aumentar la cadena. En caso de que la primera letra no esté en su máximo valor, entonces se reemplaza por el siguiente carácter.

Anteriormente se diseño el prototipo para que el digest (el array de bytes) de la combinación se convirtiera en un hexadecimal, pero después de un análisis, se llegó a la conclusión que era mas eficiente solo convertir el hash que nos dan a un array de bytes y compararlo con el digest, que convertir cada combinacion a un hexadecimal y compararlo con el hash. Después de esta optimización se vio una diferencia muy grande, por ejemplo, para el peor caso de 6 de longitud con 1 thread para el algoritmo SHA256 pasamos de un tiempo de 118997 ms a un tiempo de 33231 ms; otro ejemplo es para el peor caso de longitud 7 con 1 thread para el algoritmo de SHA256 pasamos de un tiempo de 5978930 ms a 2651992 ms.

1.2 Casos de prueba

Para los casos de prueba, se plantearon 2 escenarios:

a) En el primer escenario, utilizamos para códigos de longitud 1 a ‘m’, para longitud 2 a ‘mm’, para longitud 3 ‘mmm’, para longitud 4 ‘mmmm’, para longitud 5 ‘mmmmm’, para longitud 6 ‘mmmmmm’ y para longitud 7 ‘mmmmmmm’, y concatenando la sal correspondiente: para el algoritmo “SHA512” la sal para el primer caso es ‘22’ y para “SHA256” la sal es ‘11’.

b) En el segundo escenario, se consideró el “peor caso”. Utilizamos para códigos de longitud 1 a ‘z’, para longitud 2 a ‘zz’, para longitud 3 ‘zzz’, para longitud 4 ‘zzzz’, para longitud 5 ‘zzzzz’, para longitud 6 ‘zzzzzz’ y para longitud 7 ‘zzzzzz’, y concatenando la sal correspondiente: para el algoritmo “SHA512” la sal para el primer caso es ‘44’ y para “SHA256” la sal es ‘33’.

1.2.1 SHA512

Esta es la tabla con los resultados para el algoritmo SHA512. Como se mencionó anteriormente, para el caso 1 se utiliza una sal = ’22’, y para el peor caso, una sal = ‘44’

Longitud	Codigo	Sal	Tiempo 1 Thread	Tiempo 2 Threads
1	m	22	34	27
2	mm	22	43	41
3	mmm	22	195	137
4	mmmm	22	473	409
5	mmmmm	22	4099	2384
6	mmmmmm	22	76784	47835
7	mmmmmmm	22	3058944	1399310
1	z	44	37	34
2	zz	44	58	50

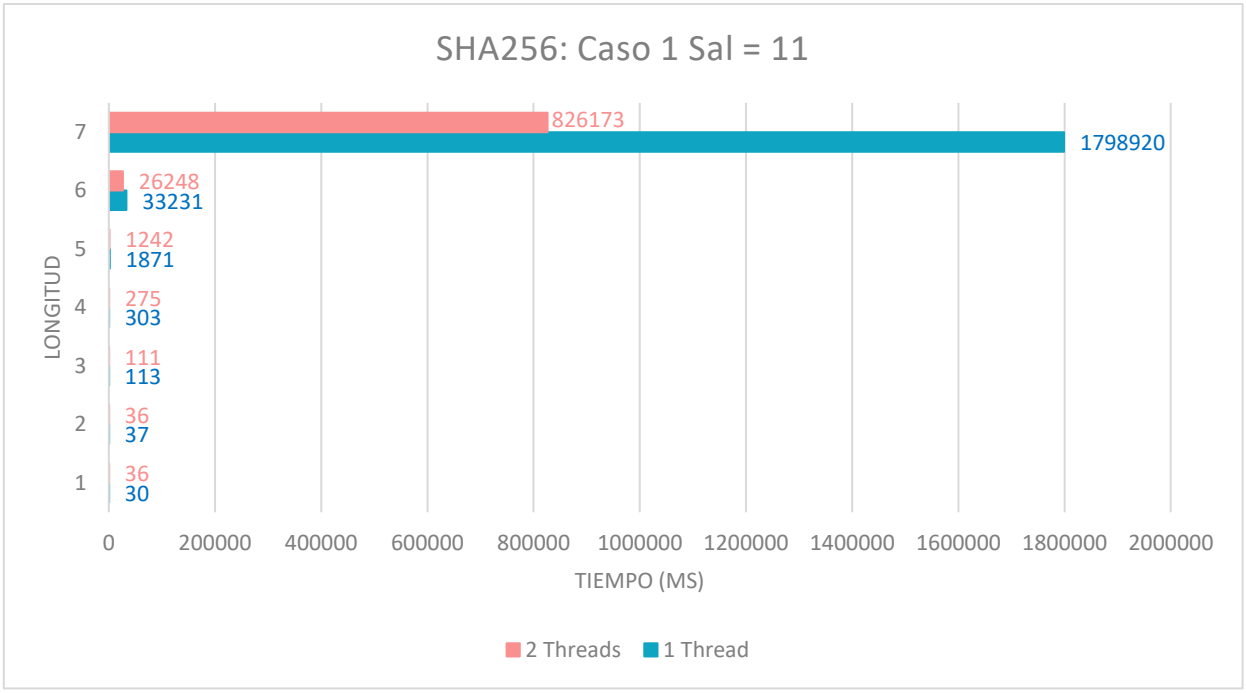
3	zzz	44	247	241
4	zzzz	44	507	466
5	zzzzz	44	7185	4864
6	zzzzzz	44	178888	69773
7	zzzzzzz	44	4026927	2141287

1.2.2 SHA256

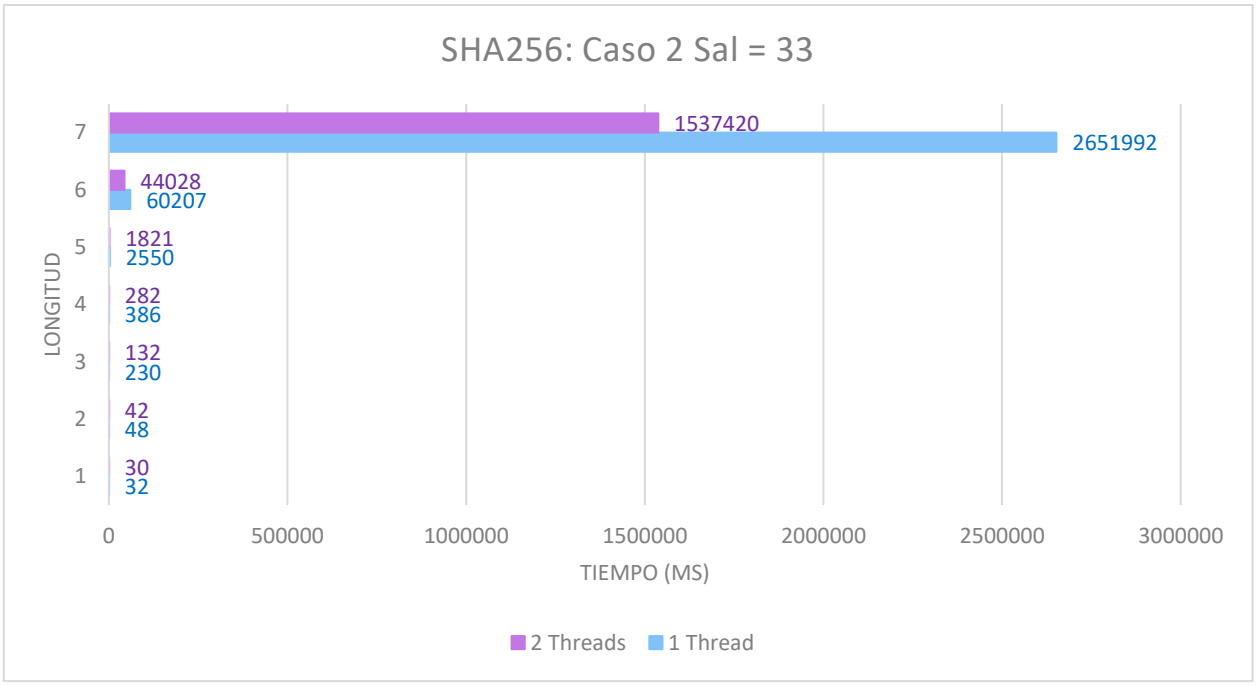
Esta es la tabla con los resultados para el algoritmo SHA512. Como se mencionó anteriormente, para el caso 1 se utiliza una sal = ’11’, y para el peor caso, una sal = ‘33’

Longitud	Codigo	Sal	Tiempo 1 Thread	Tiempo 2 Threads
1	m	11	30	36
2	mm	11	37	36
3	mmm	11	113	111
4	mmmm	11	303	275
5	mmmmm	11	1871	1242
6	mmmmmm	11	33231	26248
7	mmmmmmm	11	1798920	826173
1	z	33	32	30
2	zz	33	48	42
3	zzz	33	230	132
4	zzzz	33	386	282
5	zzzzz	33	2550	1821
6	zzzzzz	33	60207	44028
7	zzzzzzz	33	2651992	1537420

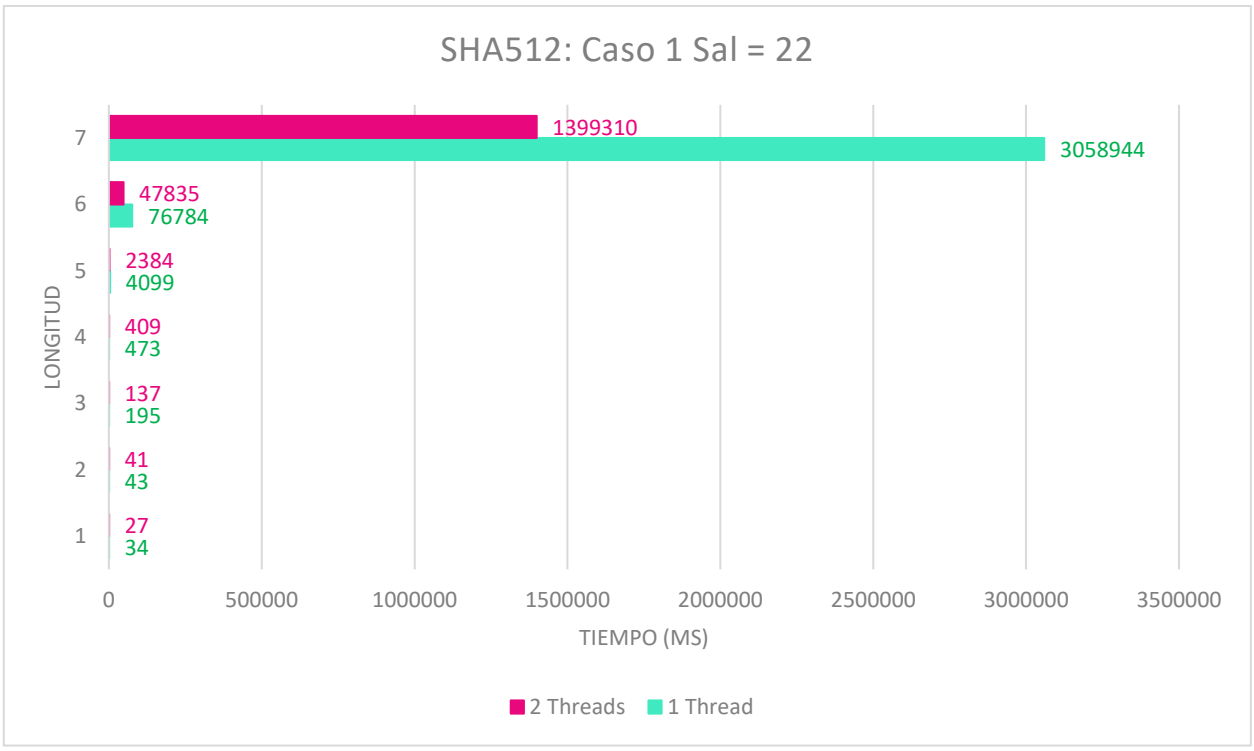
1.2 Gráficas



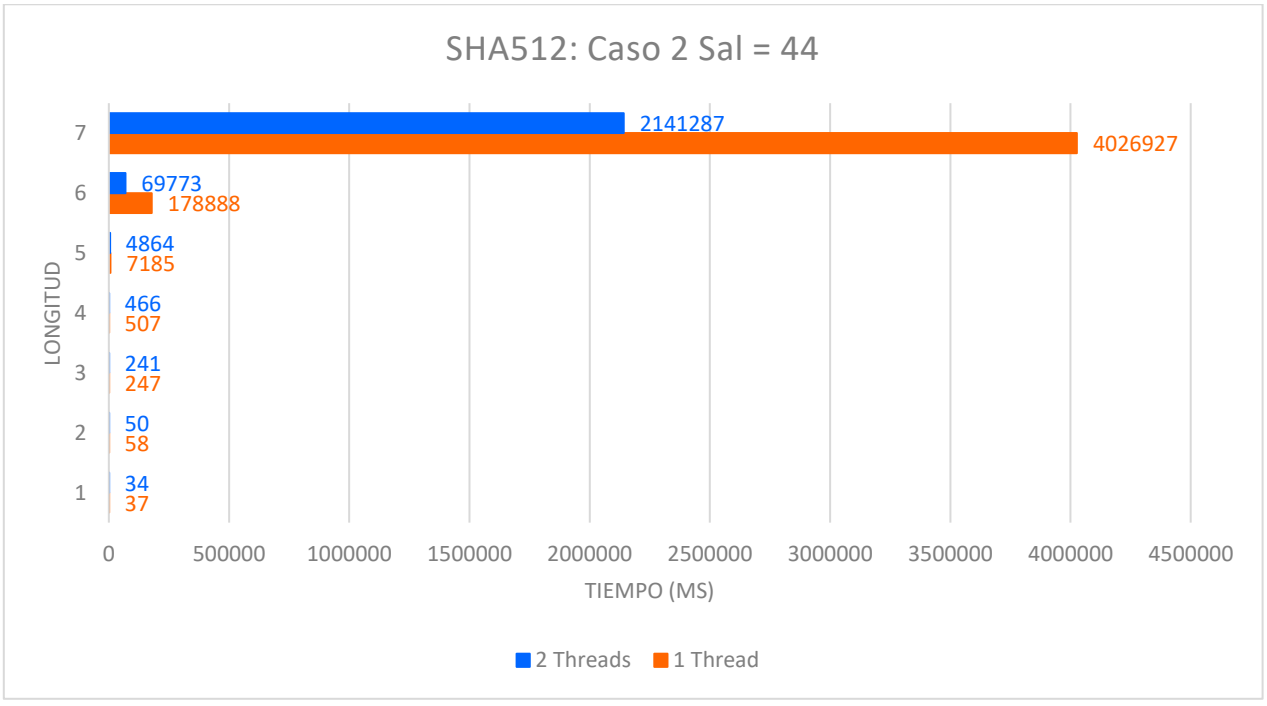
Grafica 1. Comparación tiempos caso 1 SHA256



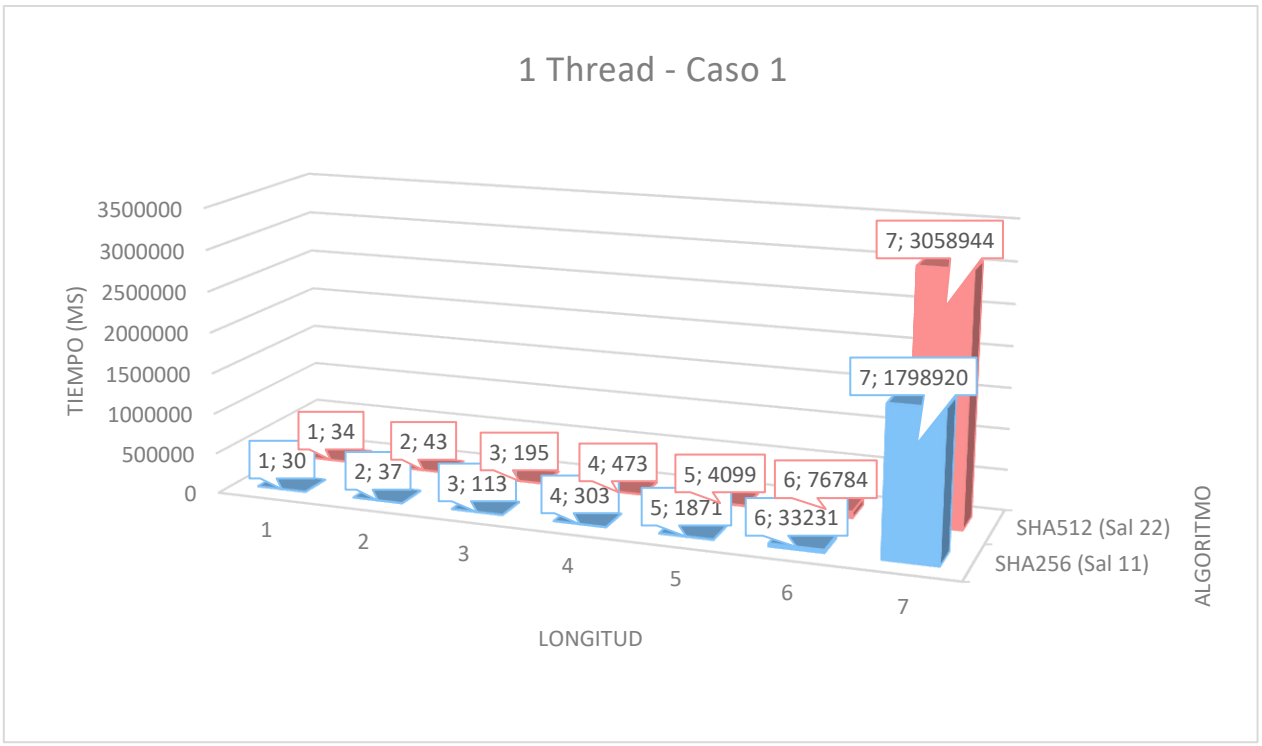
Grafica 2. Comparación tiempos caso 2 SHA256



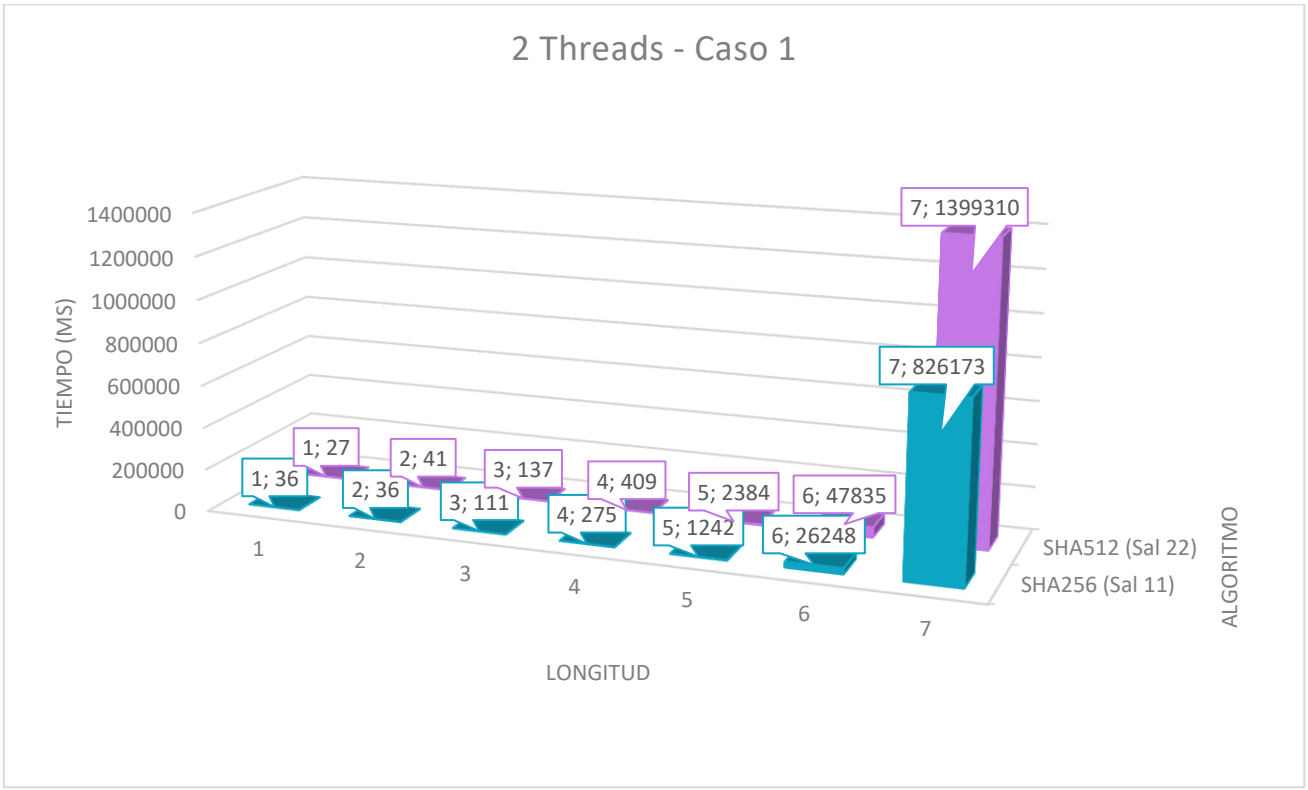
Grafica 3. Comparación tiempos caso 1 SHA512



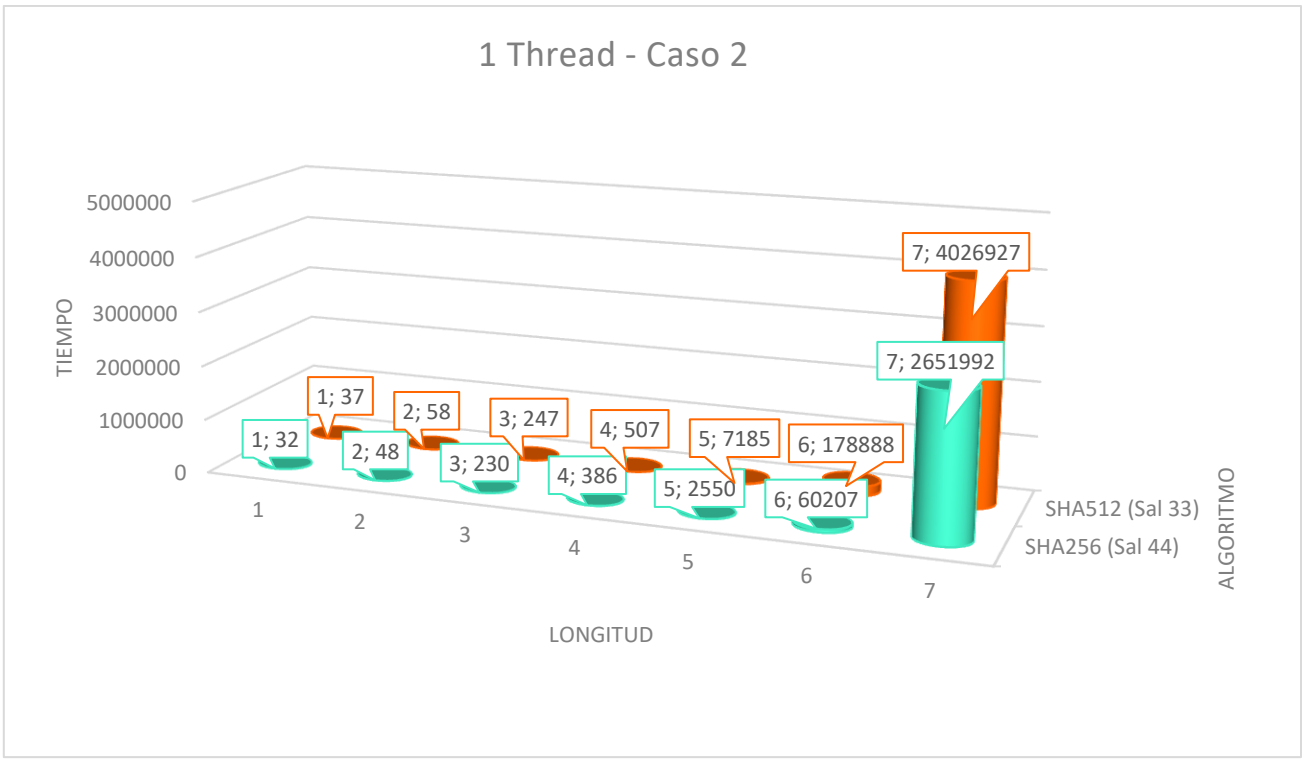
Grafica 4. Comparación tiempos caso 2 SHA512



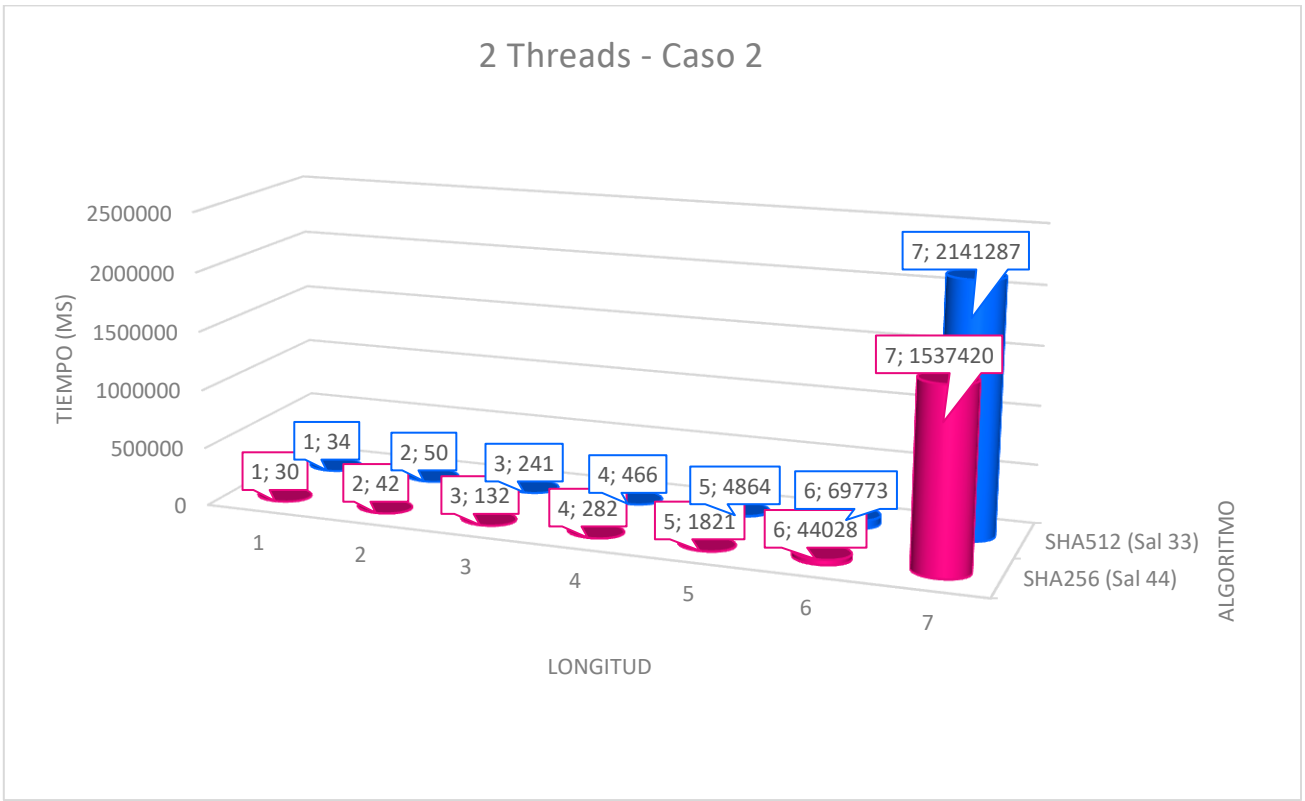
Grafica 5. Comparación de tiempos de los algoritmos en caso 1 con 1 thread



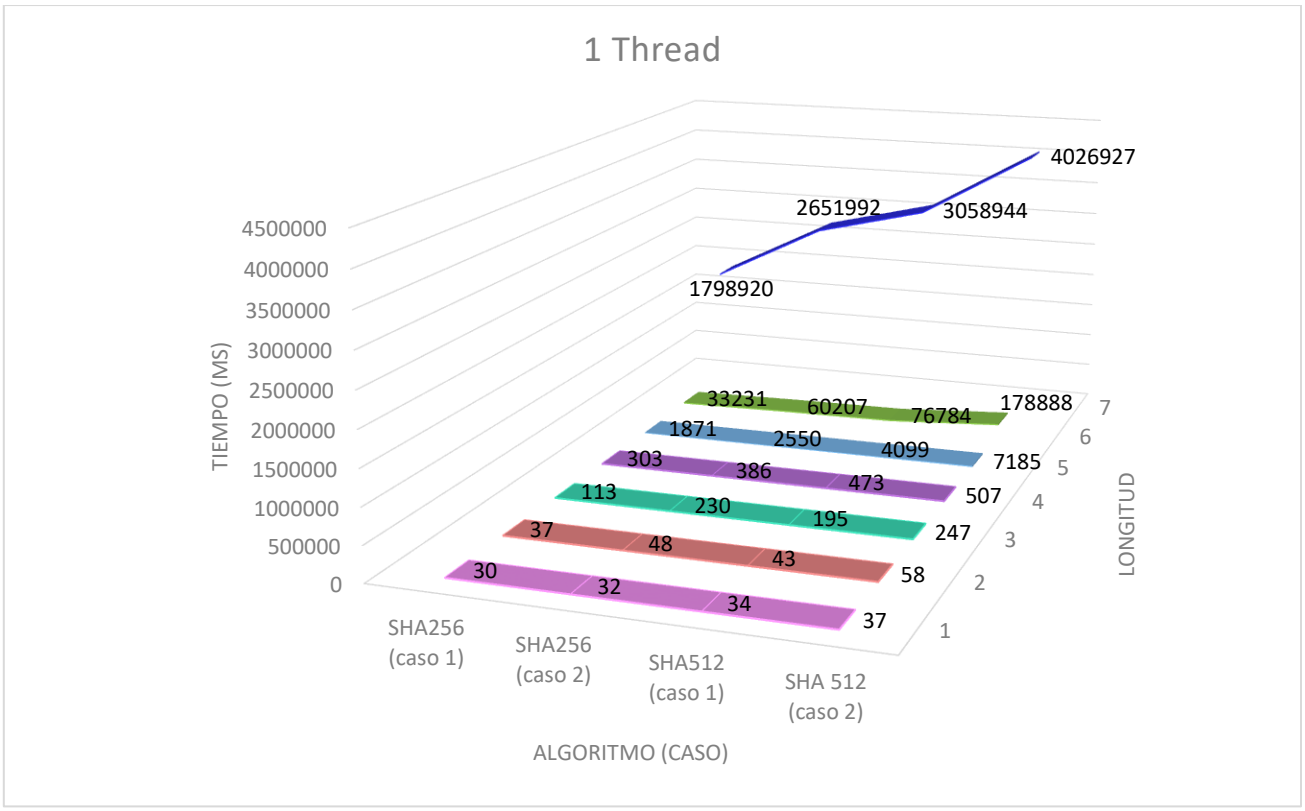
Grafica 6. Comparación de tiempos de los algoritmos en caso 1 con 2 thread



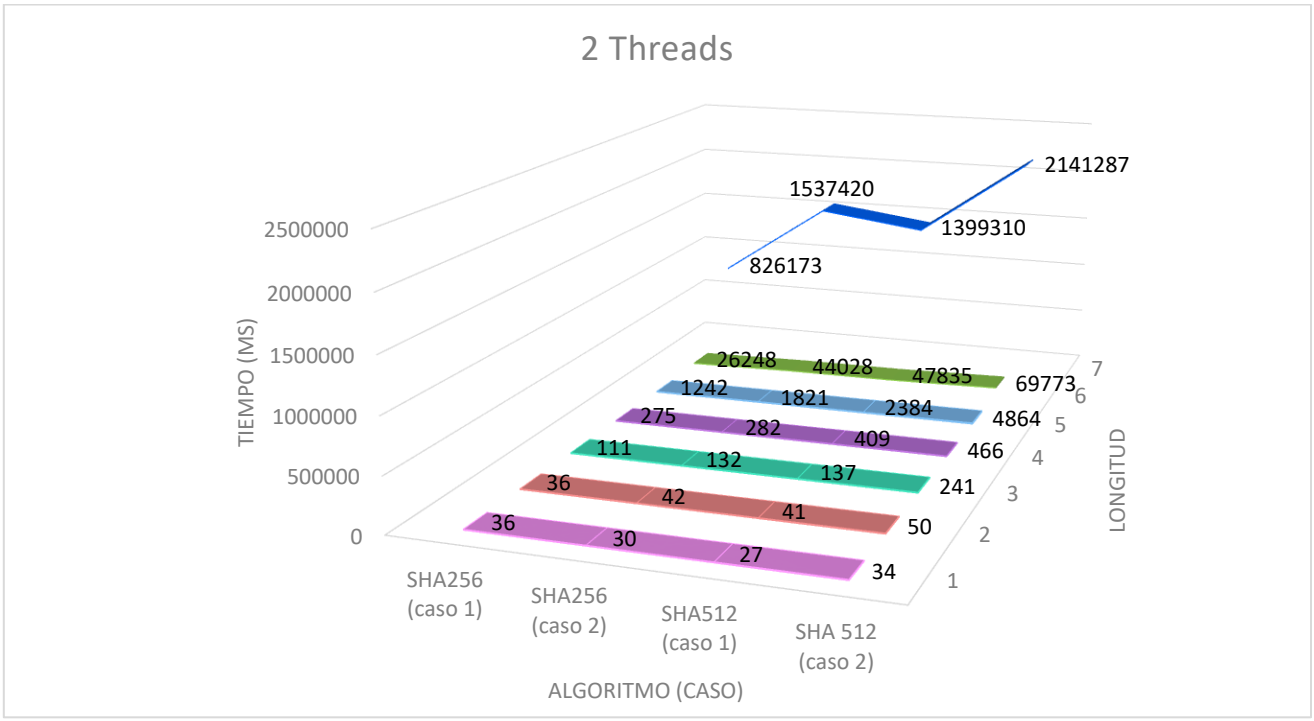
Grafica 7. Comparación de tiempos de los algoritmos en caso 2 con 1 thread



Grafica 8. Comparación de tiempos de los algoritmos en caso 2 con 2 threads



Grafica 9. Comparación de tiempos de los algoritmos en caso 1 y 2 con 1 thread



Grafica 10. Comparación de tiempos de los algoritmos en caso 1 y 2 con 2 threads

Con estas graficas podemos concluir, en términos de tiempo, descifrar un código cifrado con el algoritmo SHA256 es más rápido que descifrarlo con SHA512.

Adicionalmente, con longitudes de 1 hasta 5, la diferencia entre usar 1 thread o 2 threads no es tan significante como lo es para la de longitud 7. Para un código de longitud 7 usar 2 threads para descifrar es casi 2 veces más eficiente que solo hacerlo con 1.

1.3 Ciclos de procesador

El procesador utilizado para las pruebas fue: i7 118H 2.30GHz

$$fecuencia\ de\ reloj = 2.3 * 10^6 \frac{ciclos}{ms}$$

Para conocer los ciclos de procesador vamos a utilizar la siguiente formula:

$$Tiempo\ de\ ejecución\ =\ Ciclos\ de\ reloj *\ Duración\ del\ ciclo\ de\ reloj$$

Cabe aclarar que todo esto va relacionado a lo realizado en una sola ejecución. Es decir, estos valores, no son generales para el programa sino, que solo representa un caso. Para lograr saber un promedio de los ciclos que toma el programa en ejecutar, vamos a realizar como su nombre lo indica un promedio. Esto sobre los ciclos de reloj que toma la mínima combinación o contraseña de generar y comprobar. Luego, le sumamos los mismo, pero para la ejecución de 7 caracteres, es decir, el peor caso de zzzzzzz. Sobre estas dos realizamos el promedio para dar una respuesta que logre estar en una media del estudio.

1) Mejor caso (a, cadena de longitud 1)

$$Tiempo\ de\ ejecución\ =\ Ciclos\ de\ reloj *\ Duración\ del\ ciclo\ de\ reloj$$

$$Tiempo\ de\ ejecución\ =\ Ciclos\ de\ reloj * \frac{1}{frecuencia\ de\ reloj}$$

$$Ciclos\ de\ procesador =\ Tiempo\ de\ ejecución\ *\ fecuencia\ de\ reloj$$

$$Ciclos\ de\ procesador =\ 25\ ms\ *\ 2.3 * 10^6 \frac{ciclos}{ms}$$

$$Ciclos\ de\ procesador =\ 57500000\ ciclos$$

2) Peor caso (zzzzzzz, cadena de longitud 7)

$$Ciclos\ de\ procesador\ =\ Tiempo\ de\ ejecución\ *\ fecuencia\ de\ reloj$$

$$Ciclos\ de\ procesador =\ 4026927\ ms\ *\ 2.3 * 10^6 \frac{ciclos}{ms}$$

$$Ciclos\ de\ procesador =\ 9261932100000\ ciclos$$

3) Promedio de ciclos de reloj

$$Promedio\ ciclos\ procesador = \frac{(Ciclos\ mejor\ caso\ +\ Ciclos\ peor\ caso)}{2}$$

$$\frac{(9261932100000\ +\ 57500000)}{2}$$

$$4630994800000\ ciclos\ de\ reloj\ promedio\ en\ el\ caso$$

1.4 Cálculos

$$Tiempo\ de\ ejecución\ promedio = Ciclos\ de\ procesador\ promedio * Duración\ del\ ciclo\ de\ reloj$$

$$Tiempo\ de\ ejecución\ promedio = 4630994800000\ ciclos * \frac{1}{(2300000\ ciclos/ms)}$$

$$Tiempo\ de\ ejecución\ promedio = 2013476\ ms$$

$$Combinaciones\ caso\ promedio = \frac{8353082582}{2} = 4176541291\ combinaciones$$

$$Combinaciones\ por\ ms = \frac{Combinaciones\ caso\ promedio}{Tiempo\ de\ ejecución\ promedio} = \frac{4176541291\ combinaciones}{2013476\ ms}$$

$$Combinaciones\ por\ ms = 2074{,}2940\ \frac{combinacion}{ms}$$

$$ms\ por\ combinacion = \frac{1}{2074{,}2940\ \frac{combinacion}{ms}} = 0{,}000482\ \frac{ms}{combinacion}$$

$$Ciclos\ por\ combinacion = \frac{2.3 * 10^6\ \frac{ciclos}{ms}}{2074{,}2940\ \frac{combinacion}{ms}} = 1108{,}81\ \frac{ciclos}{combinacion}$$

Espacio de cada carácter = minúsculas + mayúsculas + naturales y 0 + .:,;!?(%)\+-/*{}, = 26+26+10+16 = 78

1) 8 caracteres

Cada carácter puede tomar cualquiera de los 78 valores antes mencionados. Así que el cálculo para el tiempo que toma la ejecución del peor caso es:

$$(78^8) * ms\ por\ combinacion = Tiempo\ de\ ejecución\ de\ combinación$$

$$78^8 * 0{,}000482 \frac{ms}{combinacion} = 660395126669{,}271552\ ms$$

Es decir, para hallar la contraseña de 8 caracteres en su peor caso sería un tiempo 20.92 años

2) 10 caracteres

Cada carácter puede tomar cualquiera de los 78 valores antes mencionados. Así que el cálculo para el tiempo que toma la ejecución del peor caso es:

$$78^{10} * 0{,}000482 \frac{ms}{combinacion} = 4017843950655848{,}12\ ms$$

Es decir, para hallar la contraseña de 10 caracteres en su, pero caso sería un tiempo 127317.79 años

3) 12 caracteres

Cada carácter puede tomar cualquiera de los 78 valores antes mencionados. Así que el cálculo para el tiempo que toma la ejecución del peor caso es:

$$78^{12} * 0{,}000482 \frac{ms}{combinacion} = 24444562595790179976{,}48\ ms$$

Es decir, para hallar la contraseña de 12 caracteres en su, pero caso sería un tiempo 774601446.11 años

Análisis y entendimiento del problema

2.1 Información de los códigos criptográficos

2.1.1 Códigos usados hoy día, y en que contexto

Hoy día, los algoritmos de hash son probados por organización, gubernamentales, para lograr verificar su seguridad, si se quisiera decir así. A lo largo, de los años se han logrado identificar falencias en alungo y otro han logrado seguir pasando estas pruebas. A continuación, vamos a mencionar algunos algoritmos que son usados o que siguen siendo usado actualmente de diversas maneras. No se pretender presentar de manera detalla cada uno, sino, se espera presentar los detalles más importantes para así entendimiento y en que

ejemplos se puede ver utilizado o en qué áreas son más utilizados. Vamos a ir mencionado cada uno y presentado características de estos. En primer lugar, encontramos el algoritmo de has BLAKE2. Este algoritmo, a grandes rasgos, recoge un texto plano y lo convierte en un código de hash. Antes de seguir, es necesario aclarar que un código de hash es una cadena de bit de tamaño fija, que define el tipo de algoritmo, de números (xilinx) . Ahora bien, además de su capacidad y ser flexible ante los requerimientos que se le quieran dar, también es en términos de tiempos de ejecución uno de los más eficientes. En segundo lugar, tenemos el algoritmo SHA256. Este algoritmo a grandes rasgos realiza la misma tarea que el mencionado antes. Sin embargo, presenta una característica que se menciona en su nombre. Y es que como se dijo antes, el hash es una cadena de bits fija. En este caso, el 256 hace referencia a que presenta una cadena de bits de este tamaño. Todas estas cadenas presentadas normalmente en hexadecimal. En tercer lugar, tenemos el algoritmo SHA512. Este algoritmo a grandes rasgos realiza la misma tarea que el mencionado antes. En este caso, el 512 hace referencia a que presenta una cadena de bits de este tamaño. Por último, encontramos el algoritmo SHA3. Ahora bien, este no presenta la característica de que el 3 represente la longitud de la cadena. Sin embargo, su característica principal es que realiza una mezcla hash en cada momento de encriptación que realiza. Es decir, presenta un nivel de complejidad mayor en términos teóricos (nina.az). Ahora bien, luego de haber mencionado los principales algoritmos usados actualmente, es necesario definir en qué contexto se están utilizando. Uno de los ejemplos más simples es el que se presenta en este mismo caso. El uso de hash para guardar las claves locales es la funcionalidad más normal del hash. Esto se debe a que es mucho más seguro guardar el hash de una contraseña que guardar la propia clave. En toro contextos más actuales encontramos las criptomonedas. Específicamente, la actualidad la identificación de los bitcoins esta desarrollada sobre el algoritmo de SHA256 (CodeSigningScore). Además, otro ejemplo es que el algoritmo de SHA3, es actualmente usando para la modificación de datos de red, en tecnologías como las VPN (nina.az).

2.1.2 Por qué se dejó de usar ciertos algoritmos

Todo este tipo de algoritmos tiene un objetivo común, puede ser para redes privadas, para identificar archivos, para guardar claves. Sin embargo, todo recaen en la misma finalidad y es que se cuente con cierta conversión de datos con el fin que vuelva difícil encontrar el mensaje inicial. Como ha sucedido con varios contextos relacionado con el avance de la tecnología, este apartado de seguridad no se queda atrás. Con el mejoramiento de los equipos de cómputo muchos algoritmos se han quedado obsoletos porque, con la mejora capacidad de los computadores, encontrar el mensaje inicial que generen esos códigos se ha vuelto mucho más sencillo con el pasar del tiempo. Esto quiere decir muchos algoritmos se han quedado obsoletos por su incapacidad de mantener segura la información o su facilidad de crackeo si se quiere decir de alguna manera. Entre estos algoritmos que se han quedado obsoletos encontramos MD5, SHA1 entre otros (NSIT). Todos estos han sufrido la misma vulnerabilidad. Y es que se ha logrado encontrar la base del mensaje de manera mucho más

fácil. Un ejemplo, es que estos algoritmos ni siquiera logran mantener el código de hash. Esto quiere decir que se ha logrado encontrar otro mensaje que genere el mismo código. Esto genera que no sea seguro, ya que, si mi mensaje que envió no se mantiene, sino que otro lo modifica y no me entero porque mi código no cambia, quiere decir que es necesario cambiar al algoritmo que genera el código. Por lo tanto, la mayoría a de algoritmos que ya se consideran obsoletos, es por el problema de que no son capaces de mantener el mensaje oculto, es decir, se puede suplantar el mensaje o se puede encontrar muy fácilmente.

2.2 Mining

El mining es un proceso por el cual se validan y registran transacciones en el blockchain (Una base de datos segura e inmutable). La minería se centra en la resolución de problemas matemáticos complejos para confirmar transacciones y agregar nuevos bloques a la blockchain. Para esto se usan computadores especializados en realizar cálculos. Tiene varias etapas.

1. Validación de transacciones: Cuando se realiza una transacción se envía a todos los nodos de la blockchain para su validación.
2. Recopilación de transacciones en bloques: Luego de ser validada, la transacción se agrega a un pool de transacciones pendientes. Los mineros las recopilan y agrupan en bloques.
3. Resolución de problemas matemáticos: Una vez que se ha creado un bloque de transacciones, los mineros deben resolver un problema matemático complejo. La resolución de este problema implica realizar una gran cantidad de cálculos utilizando la potencia de procesamiento de sus computadores.
4. Verificación del bloque: Una vez que se ha resuelto el problema matemático, el bloque de transacciones es verificado y validado por otros nodos de la red. Si el bloque es aceptado, se agrega a la cadena de bloques existente y se recompensa al minero con una cierta cantidad de la criptomoneda que está minando.

Estos 4 pasos se repiten constantemente en la blockchain. Cada que se agrega un nuevo bloque, se crean nuevas unidades de la criptomoneda que está siendo minada y se otorga una recompensa al minero que lo resolvió.

2.3 Sal en rainbow tables

2.2.1 Problema de seguridad asociado

Antes de empezar a hablar del problema de seguridad sería necesario definir que es una Rainbow table. Esta tabla, no tiene nada que ver con su nombre. Sin embargo, esta tabla tiene que ver con una variedad. Dentro de este “arcoíris”, se encuentran una variedad de cogidos. Estos códigos son el hash de todas las posibles combinaciones de contraseñas. Estas combinaciones recordemos de algoritmos, anteriores que dependen mucho de las características que se buscan. Esto quiere decir, que, en esta tabla, se guardan los hashes de

las combinaciones que son necesarias para evaluar una contraseña. Luego de haber definido lo que es esta Rainbow tablé, ahora si vamos a definí cual es el problema relacionado con esta tabla. En este caso, ya se tocó este un poco, y es que es relacionado con las contraseñas almacenadas. El problema el cual está relacionado con la tabla, es el espionaje si se quiere decir. Esto quiere decir, que el problema para el cual es utilizado el tabal es para atacar contraseñas. Esto se deba que cuando se tienen todas las combinaciones posibles de contraseña con su hash respectivo. Es probable que se pueda hallar el valor y así se drice que se está atacando la contraseña. Por lo que el problema de seguridad relacionado con esta tabla es el de espionaje o de robo de contraseña o de ataque de contraseñas o de robo de contraseñas de acuerdo con su contexto (AuditoriadeCodigo).

2.2.2 Cómo ayuda la sal a resolver este problema

Ahora al hablar del tema de la sal, es necesario definir antes que es la sal. La sal es una cadena aleatoria, dependiendo del sistema, de bits que se genera de acuerdo con unos parámetros a o unas características. Ahora bien, si se quiere pensar la sal es como una contraseña que genera el sistema. Ahora bien, luego de haber dicho este es necesario definir como ayuda la sal a mitigar el problema antes mencionado. Para esto es necesario usar un ejemplo. La idea de que se pueda revelar una contraseña recae en la idea de cuánto tiempo y cuantos recursos tiene un ataque ante para evaluar todos los caracteres de la contraseña de una persona, ya que como lo mencionamos antes el atacante tiene que encontrar todas las combinaciones y compara el hash guardado con los hashes de estas combinaciones. Como se sabe este proceso gasta mucho tiempo y recursos de cierta manera. Sin embargo, no es imposible que se encuentre con cierta cantidad de suerte. Ahora bien, que pasaría se a la contraseña se le agrega un tipo de cadena aleatoria y es este el hash que se guarda, no solo del de la contraseña sola. Esto lo que generaría es que el atacante no solo tenga que encontrar la contraseña, sino que también tenga que buscar la sal, la cual tiende a ser única en cada contraseña. Sumándole a esto que la combinación puede cambiar por cada sal que se tenga y dependiendo de su longitud. Por lo tanto, la sal ayuda a mitigar este problema al añadirle un nivel más si se quiere pensar, a evaluar o encontrar una contraseña.

Referencias

1. Callaghan, P. (s/f). *Why you should use SHA-256 in evidence authentication*. Pagefreezer.com. Recuperado el 3 de mayo de 2023, de https://blog.pagefreezer.com/sha-256-benefits-evidence-authentication

2. *Hash algorithm comparison: MD5, SHA-1, SHA-2 & SHA-3*. (2022, febrero 15). Code Signing Store; CodeSigningStore. https://codesigningstore.com/hash-algorithm-comparison
3. Auditor. (2019, agosto 2). *Rainbow tables, creación y uso (cracking hash)*. Auditoría de código. https://auditoriadecodigo.com/rainbow-tables-como-se-crean-y-usan-las-tablas-de-cracking-de-hash-mas-potentes-explicado-facil-o-casi-facil/
4. Euny Hong. (2017, October 17). How does bitcoin mining work? Investopedia. https://www.investopedia.com/tech/how-does-bitcoin-mining-work/
5. Aumasson, J.-P., Neves, S., Wilcox-O'Hearn, Z., & Winnerlein, C. (2013). BLAKE2: Simpler, Smaller, Fast as MD5. En *Applied Cryptography and Network Security* (pp. 119–135). Springer Berlin Heidelberg.
6. Mammedov, D. (2023, enero 15). SHA-3. *www.wiki3.es-es.nina.az*. https://www.wiki3.es-es.nina.az/SHA-3.html
7. (S/f). Libertex.org. Recuperado el 4 de mayo de 2023, de https://libertex.org/es/blog/sha256
8. *No title*. (s/f). Github.Io. Recuperado el 4 de mayo de 2023, de https://xilinx.github.io/Vitis_Libraries/security/2020.1/guide_L1/internals/blake2b.html