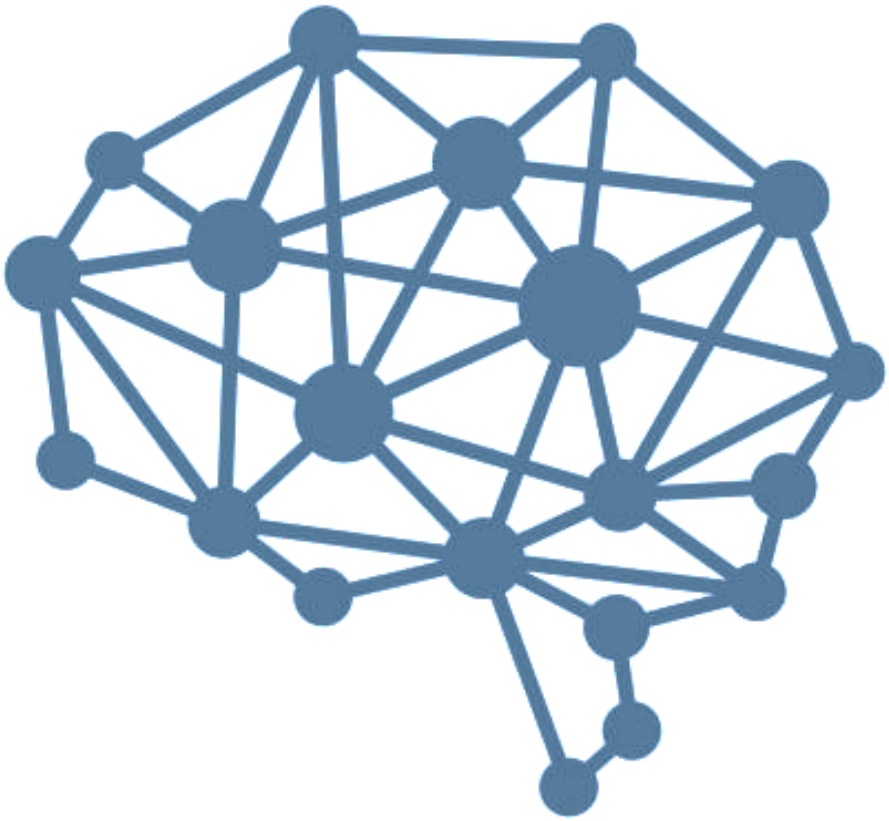
CSD. Trabajo 1

INTEGRIDAD DE LA INFORMACIÓN
BUSQUEDA DE SOLUCIONES



Autor: Javier García Ibáñez

DNI: 20952351H

Universidad Politécnica de Valencia

Índice de contenidos

Índice de Tablas

Índice de Gráficos

Introducción

El objetivo de esta práctica es implementar y analizar dos algoritmos estudiados en la asignatura para ataques a funciones de resumen. La búsqueda de contraseñas alternativas mediante construcción de tablas arcoíris y la búsqueda de mensajes diferentes con el mismo resumen mediante el algoritmo de Yuval.

Toda la programación requerida para esta práctica se realizará en lenguaje Python y mediante el editor de código Visual Studio Code.

El estudio de cada algoritmo se fundamentará en pruebas realizadas con anterioridad y cuyos resultados estarán disponibles en el siguiente repositorio GitHub.

Búsqueda de contraseñas alternativas- Arcoíris

En esta parte de la práctica se pretende estudiar el Ataque Tabla Arcoris, en adelante algoritmo arcoíris, mediante el análisis de diferentes pruebas. Este algoritmo se basa en la premisa de que, para el almacenamiento seguro de contraseñas, lo que se guarda es el resumen de estas, de manera que si se llegase a poder acceder a dicho almacenamiento no se obtuviese la clave.

Cuando el usuario introduce una clave se le aplica un resumen y se compara con el valor almacenado. Esto deja también una pequeña vulnerabilidad en caso de obtener este resumen, y es que no seria necesario obtener la clave original si no cualquiera cuyo resumen sea el mismo. Para este problema se aplica una búsqueda mediante tabla arcoíris.

Implementación

Obviando la explicación y descripción de la implementación base del algoritmo Arcoíris, pasamos a describir las funciones específicas que se han añadido a este.

Función de Resumen

En cuanto a la función resumen utilizada se ha optado, en este caso, por CRC32, evitando tener que truncar la salida, ya que esta devuelve 32 bits, lo cual se considera suficiente para abordar este problema en el ámbito de este estudio.

Además, esta era una de las funciones recomendadas en el guion de prácticas, por lo que fue la primera que se puso a prueba.

Espacio de claves considerado

En una primera implementación del algoritmo base, y a modo de experimentación juguete para observar el comportamiento del programa se utilizará el espacio de claves de las contraseñas de 6 dígitos con números del 0 al 9, lo que implica un total de un millón de claves posibles.

Más tarde se ampliará el espacio de búsqueda a claves de 5 caracteres pertenecientes al conjunto del abecedario en minúsculas sin considerar la “ñ”, lo que implicará un conjunto 26^5 (11.881.376) posibles contraseñas.

Estructura de la Tabla Arcoíris

Al igual que con el espacio de claves, el tamaño y estructura de la tabla variara en función de las pruebas a realizar, aunque siempre manteniendo la premisa de tener un número de filas mucho mayor al de columnas.

De cara a una ejecución inicial para el espacio de claves de 6 dígitos del 0 al 9, se utilizará una tabla de dimensiones 5000 x 200, obteniendo una tabla que podría englobar el millón de posibilidades que supone este espacio de claves.

Funciones recodificantes

En cuanto a las funciones recodificantes implementadas también varían en función del espacio de claves utilizado.

Función R1 – Cambio de Base (Números)

Require: el resumen de una contraseña en entero

Ensure: contraseña de 6 digitos numéricos

contraseña = resumen modulo un millón

contraseña = rellenar contraseña con ceros hasta 6 dígitos

Al trabajar en todo momento con enteros esta función se vuelve muy sencilla a la vez que efectiva, ya que al hacer modulo un millón a un resumen que puede tener un valor de hasta 2^{32} se obtiene un numero entre 0 y 999999, cubriendo todo el espacio de búsqueda.

Función R2 – Cambio de Base (Caracteres)

De forma similar a la función anterior se realiza un cambio de base, pero esta vez por agrupaciones de bits, de forma que cada una de estas agrupaciones, tras hacerle el cambio de base al tamaño de mi cadena, representen uno de los caracteres de la misma.

En esta función en concreto se cogen dígitos de 8 en 8 de los 32 que conforman el hash en binario para aplicarle el cambio de base correspondiente generando cada uno de ellos un carácter, que finalmente se unen en una cadena.

También he de recalcar que, al no poder coger 5 grupos de 8 dígitos diferentes dentro de los 32 bits del resumen, se han utilizado bits del modo (0-8, 6-14, 12-20, etc…)

Require: el resumen de una contraseña en binario

Ensure: contraseña de 5 caracteres alfabeticos

letra1 = caracteres[resumen [0:8]modulo tamaño de caracteres]

letra2 = carateres[resumen[6:14]modulo tamaño de caracteres]

letra3 = carateres[resumen[12:20]modulo tamaño de caracteres]

letra4 = carateres[resumen[18:26]modulo tamaño de caracteres]

letra5 = carateres[resumen[24:32]modulo tamaño de caracteres]

contraseña = letra1 + letra2 + letra3 + letra4 + letra5

Función R3 – Cambio de Base con movilidad de cadena por iteración

Esta función es idéntica a la anterior, aunque implementando una rotación de la cadena de caracteres por cada iteración.

Require: el resumen de una contraseña en binario

Ensure: contraseña de 5 caracteres alfabéticos

Roto mi cadena de caracteres uno a la izquierda en cada iteración

letra1 = caracteres[resumen [0:8]modulo tamaño de caracteres]

letra2 = carateres[resumen[6:14]modulo tamaño de caracteres]

letra3 = carateres[resumen[12:20]modulo tamaño de caracteres]

letra4 = carateres[resumen[18:26]modulo tamaño de caracteres]

letra5 = carateres[resumen[24:32]modulo tamaño de caracteres]

contraseña = letra1 + letra2 + letra3 + letra4 + letra5

Esta rotación permite reducir la probabilidad de colisión en la construcción de la tabla, es decir, aumentar la diversidad dentro de esta, y por tanto aumentar la probabilidad de colisión con el hash a buscar.

También hubo intención de utilizar puertas AND y OR para combinar los bits del hash con el fin de añadir diversidad, pero tras unas pocas pruebas se vio como la utilización de estas puertas hacía tender siempre a que la recodificación terminase siendo todo unos o todo cero.

Características del Ordenador

La experimentación de este algoritmo se ha realizado en un Mini PC con las siguientes características:

- Procesador: Intel de procesador de 11.ª gen N5095 (2,9GHz)
- Núcleos: 4 – Procesos: 4
- Gráfica: No contiene
- RAM: 16 GB DDR4
- Memoria: 256 GB SSD
- Sistema Operativo: Ubuntu 20.04

Experimentación y análisis

Llegados a esta parte del estudio se presentaron varios problemas durante el desarrollo del mismo. El principal fue que, debido a la falta de conocimiento acerca del funcionamiento interno del algoritmo en aquel momento, se consideró como espacio de claves todas las posibles combinaciones de ocho de los siguientes caracteres:

“abcdefghijklmnñopqrstuvwxyzABCDEFGHIJKLMNÑOPQRSTUVWXYZ0123456789”

Es decir, el conjunto del alfabeto tanto en minúsculas como en mayúsculas y los números, lo que equivale a un espacio de búsqueda de 64^8. Algo totalmente fuera del alcance de los medios computacionales de los que dispongo.

En cuanto a las pruebas realizadas para este espacio de claves fueron desechadas conformen se terminaron, ya que, evidentemente los resultados eran todo ceros, pues a pesar de utilizar tablas de tamaño 1000000 x 1000 no se cubría ni un 1% del espacio de búsqueda.

Una vez fui consciente del verdadero tamaño del espacio claves que trataba de atacar decidí separar la experimentación en dos secciones tal y como indica el guion de prácticas.

Una primera parte, la experimentación base, en la que se mostraran los resultados de las pruebas realizadas para un espacio de claves asequible, con el objetivo de familiarizarse con el algoritmo, y una segunda parte, la experimentación extendida, donde se tratará de encontrar claves equivalentes para un espacio de búsqueda bastante más complejo.

Experimentación Base

Para la experimentación base se utilizará el espacio de claves de 6 dígitos numéricos, teniendo un tamaño total de 1 millón de posibles claves.

Como función de recodificación se utilizará R1, explicada en el apartado (Funciones recodificantes).

Se utilizarán un total de 15 claves y una tabla arcoíris de 5000x200, ya que, en principio, con este tamaño se permitiría cubrir todo el espacio de búsqueda.

Claves utilizadas: {456789, 987654, 123456, 937586, 998263, 712874, 728471, 876543, 238745, 665534, 918237, 564738, 163937, 228844, 947264}

Para obtener un resultado preciso se realizarán 100 ejecuciones del programa, almacenando los resultados de manera que permita obtener una media del porcentaje de acierto que se ha tenido

Los resultados obtenidos para estas 100 ejecuciones son de 1 equivalencia para las 15 contraseñas en todas y cada una de las 100 ejecuciones, lo que representaría un 6,66% de encontrar equivalencia.

Por otro lado, el porcentaje de colisión en la tabla arcoíris es del 100% en todas las ejecuciones lo que implica que al menos hay un buen porcentaje del espacio de búsqueda cubierto.

Sin embargo, al mirar más detenidamente los datos obtenidos, he observado que la contraseña para la cual encuentra colisión es siempre la misma, la décima contraseña de la lista (665534), y siempre la misma contraseña equivalente, ella misma. Aunque la contraseña de colisión en la tabla arcoiris es siempre una diferente en cada caso.

En base a estos resultados podemos suponer que de alguna forma la función recodificación R1 tiende a construir esta contraseña con mayor frecuencia que el resto.

Experimentación extendida

Como ya se ha visto en la experimentación base, el espacio de claves utilizado anteriormente es bastante reducido, por lo que tampoco se puede aplicar un estudio exhaustivo basándonos en este.

En este apartado no solo se ampliará el espacio de claves, si no que se analizará el impacto de diferentes tamaños de tablas arcoíris y la diferencia entre dos funciones de recodificación diferentes.

El espacio de claves será, en este caso, las combinaciones de 5 caracteres del alfabeto sin tener en cuenta la ñ, es decir, 26^5 (11.881.376) posibles claves.

Para las pruebas sobre este espacio de claves se utilizarán 200 contraseñas y se probarán 16 combinaciones de tabla distintas, utilizando la función de recodificación R2.

Pruebas con función R2

Los resultados obtenidos de las pruebas utilizando la funcion R2 son los siguientes. En las tablas se muestran en forma de porcentaje el número de colisiones (Tabla 1), es decir, el número de veces que se encuentra una coincidencia dentro de la tabla arcoíris, y el número de equivalencias (Tabla 2), que refleja la cantidad de contraseñas alternativas encontradas a partir de las anteriores colisiones.

	50	**100**	**200**	**300**
5000	34,5%	75%	92%	97%
10000	57,5%	89%	96,5%	98%
30000	80%	96%	98%	99,5%
50000	90,5%	96%	99%	99,5%

Tabla 1. Porcentaje de Colisiones con R2 para diferentes tamaños de tabla.

	50	**100**	**200**	**300**
5000	3,5%	2%	1,5%	1,5%
10000	2%	3%	1,5%	0,5%
30000	3,5%	3%	2%	1,5%
50000	4,5%	3,5%	2%	1,5%

Tabla 2. Porcentaje de Equivalencias con R2 para diferentes tamaños de tabla.

En base a los resultados de la tabla, y teniendo en cuenta que el objetivo del algoritmo es encontrar el mayor número de equivalencias posibles podemos, podemos afirmar que la dimensión de tabla que mejor ha funcionado para este conjunto de pruebas es la de 50000 filas X 50 columnas.

Realmente este resultado se aleja mucho de lo que esperaba, ya que, en un principio planteaba que el mejor resultado se obtendría para un tamaño de tabla que permitiese albergar todas las claves posibles del espacio de búsqueda o al menos un gran número de ellas, sin embargo, la tabla con mayor tasa de equivalencia únicamente podría albergar entorno a un 25% de las posibles claves.

También se puede observar en las tablas, como por normal general se reduce el porcentaje de equivalencia a medida que aumenta el número de columnas, tal y como se representa en la Ilustración1.

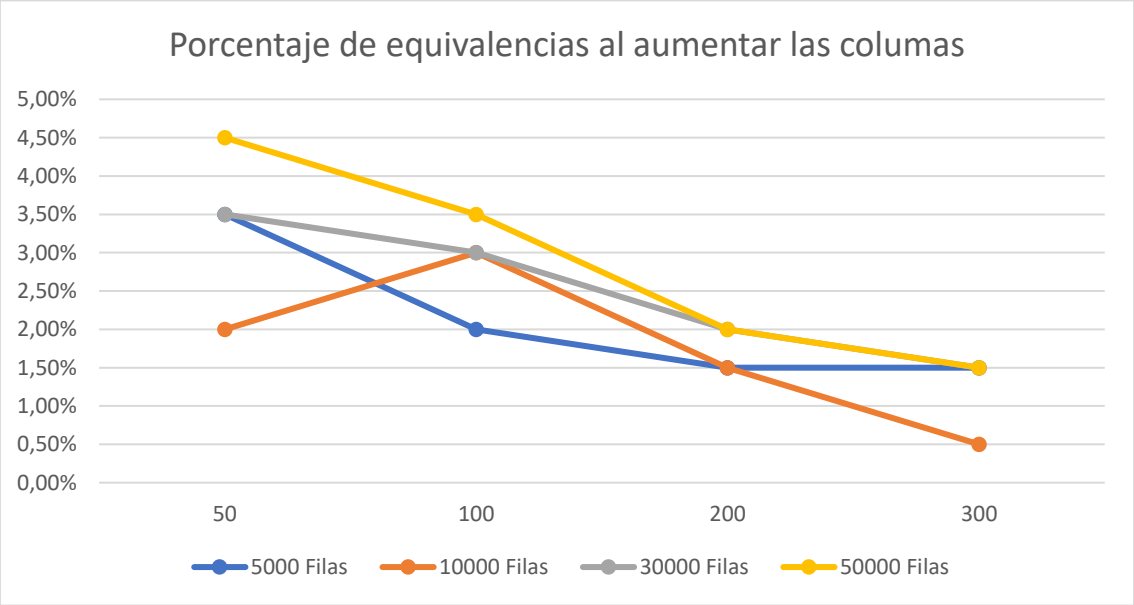


Ilustración 1. Exp extendida. Porcentaje de equivalencias al aumentar columnas

En base a esta observación, y puesto que con el porcentaje de colisiones ocurre lo contrario, es decir, aumenta en a mayor número de columnas, se puede intuir que la función recodificación utilizada permite la obtener el mismo anclaje encontrado en la tabla arcoíris por multitud de procesos diferentes.

Pruebas con función R3

Basándonos en el bajo porcentaje de equivalencia encontrado utilizando la función R2 se decide modificar esta recodificación para incrementar la variabilidad.

La idea de la nueva función es añadir una mayor diversidad a las contraseñas creadas, para ello se implementa una rotación de la cadena de caracteres basada en la iteración en la que se encuentra el programa en cada momento, es decir, si el programa se encuentra en la iteración X la cadena de caracteres rotará X módulo (tamaño de la cadena) caracteres hacia la izquierda.

De este modo un mismo hash generado en dos iteraciones diferentes no debería generar la misma contraseña. Con esto no solo debería aumentarse la diversidad de contraseñas en tabla, sino además reducir la probabilidad de llegar al mismo anclaje en la tabla por dos caminos diferentes.

	50	**100**	**200**	**300**
5000	4%	7,5%	34%	51%
10000	4%	20%	52,5%	79%
30000	14%	47%	86%	97%
50000	26,5%	61%	94%	99%

Tabla 3. Porcentaje de Colisiones con R3 para diferentes tamaños de tabla.

	50	**100**	**200**	**300**
5000	0%	0,5%	0,5%	1%
10000	0%	0%	0%	0,5%
30000	1%	2%	1%	2%
50000	0%	1,5%	0,5%	2%

Tabla 4. Porcentaje de Equivalencias con R3 para diferentes tamaños de tabla.

A la vista de los resultados se puede observar que lo planteado en la hipótesis parece no ser del todo cierto, ya que no solo se obtiene menos porcentaje de equivalencia, sino también de colisión en todos los casos.

Esto podría deberse a que, para tablas de tamaño pequeño, al utilizar un mayor grado de variabilidad no se alcanza un gran porcentaje de las posibles claves y para tamaños de tabla más grande, como por ejemplo la combinación 50000 x 300, que permitiría albergar el espacio de búsqueda completo, aunque como se puede observar en la tabla se obtiene un 99% de colisión, se estén generando una multitud de caminos diferentes para el mismo resultado.

Esta última explicación podría tener bastante lógica, ya que como se dijo en la hipótesis al mover la cadena de caracteres provocamos que el mismo hash no genere la misma contraseña en iteraciones diferentes. Por el contrario, también se está generando la casuística de que hashes diferentes generen la misma clave para iteraciones diferentes.

Búsqueda de documentos alternativos- Yuval

En esta parte de la práctica se pretende estudiar el algoritmo de Yuval mediante el análisis de diferentes pruebas. Este algoritmo se basa en la premisa de que, en los procesos de firma electrónica, el protocolo no considera el mensaje en su totalidad, sino un resumen de este, es decir, su *hash.*

Aprovechando esta vulnerabilidad se podría reutilizar la firma de un documento para firmar de forma fraudulenta otro mensaje diferente, pero con el mismo resumen.

Para la búsqueda de colisiones de resúmenes de documentos se considerarán aproximaciones basadas en la paradoja del cumpleaños, la cual establece que, de un conjunto de 23 personas, hay una probabilidad del 50,7% de que al menos dos de ellas cumplan años el mismo día. Para 57 o más personas la probabilidad es mayor del 99,666%. Esto se refiere a que de entre 57 de las 365 posibilidades, dos de ellas son iguales con casi un 100% de probabilidad.

Para aplicar esta paradoja a nuestro problema particular se partirá de una implementación base del algoritmo de Yuval.

Implementación

Obviando la explicación y descripción de la implementación base del algoritmo de Yuval, pasamos a describir las funciones específicas que se han añadido a este.

Función de Resumen

En cuanto a la función hash utilizada se ha optado por MD5, truncando la salida al valor de bits deseado en cada momento, ya que el estudio se realiza sobre la diferencia en cuanto al tamaño de este.

MD5 es un algoritmo de resumen criptográfico de 128 bits, lo que nos permite poder estudiar hasta dicha cantidad de bits.

Modificaciones del Algoritmo Base

En lo que respecta a la estructura base del algoritmo únicamente se ha modificado el bucle de búsqueda de coincidencia, de manera que, para un mismo tamaño de hash, cuando no encuentra coincidencia en los $2^{m/2}$ hashes de textos generados para dicho tamaño de hash se aumentan los textos en tamaño del diccionario a $2^{(m/2)+1}$ hasta que lo encuentre.

Esto se hace principalmente para cumplir el requisito de estudio del apartado dos de obtener al menos 15 colisiones, teniendo únicamente 15 textos originales, al asegurarnos al 100% de encontrar colisión para cada uno de ellos.

Original

Generar $t = 2^{x=m/2}$ modificaciones menores de x_l

Computar el resumen y almacenar los pares $(x_l, h(x_l))$

while no se encuentre colisión **do** {probable en t intentos}

    Generar x_i modificación menor de x y computar $h(x_i)$

    Buscar si existe x_l tal que $h(x_l) = h(x_i)$

end while

Modificación

while no se encuentre colisión **y** x < número de líneas totales de mi texto

    Generar $t = 2^{x=m/2}$ modificaciones menores de x_l

    Computar el resumen y almacenar los pares $(x_l, h(x_l))$

    while no se encuentre colisión **do** {probable en t intentos}

        Generar x_i modificación menor de x y computar $h(x_i)$

        Buscar si existe x_l tal que $h(x_l) = h(x_i)$

    end while

    x=x+1

end while

El código al completo se encuentra en el directorio de github:

El código no requiere guía ya que únicamente está compuesto de un script, sin argumentos ni interacción alguna del usuario, que genera automáticamente todos los datos utilizados en este documento.

También se podrá encontrar en el mismo directorio de github un archivo con los resultados obtenidos durante las ejecuciones.

Características del Ordenador

La experimentación de este algoritmo se ha realizado en un Mini PC con las siguientes características:

- Procesador: Intel de procesador de 11.ª gen N5095 (2,9GHz)
- Núcleos: 4 – Procesos: 4
- Gráfica: No contiene
- RAM: 16 GB DDR4
- Memoria: 256 GB SSD
- Sistema Operativo: Ubuntu 20.04

Documentos utilizados

En cuanto a los documentos base que se pretenden atacar mediante el algoritmo consisten en documentos CSV con dos columnas y 32 filas cada uno, de modo que construyendo un mensaje con cualquier combinación posible de las columnas se obtiene un texto con sentido y coherencia, pudiendo llegar a formar 2^{32} textos diferentes.

Hay un total de 16 archivos CSV, 1 para los textos lícitos y 15 para los ilícitos.

Estos documentos se encuentran también en el mismo repositorio de github que el resto del proyecto (GITHUB). Aunque se expondrá a continuación uno los probables resultados tanto de texto licito como ilícito

Lícito

Por medio de este documento, yo, Patricio Estrella me comprometo con una misión de vital importancia para nuestra comunidad submarina. Durante años he vivido en la pacífica ciudad de Fondo de Bikini disfrutando de la pacífica vida en mi roca paseando por sus hermosas calles festejando con los vecinos e ignorando por completo como la dejadez estaba consumiendo nuestra ciudad. Sin embargo, ha llegado el momento de asumir un papel más activo en el bienestar de nuestro hogar bajo el mar. Me comprometo a liderar un proyecto de limpieza de nuestros arrecifes de coral y a la conservación de nuestra preciada flora y fauna marina. Nuestra hermosa ciudad se ha visto afectada por la contaminación y el descuido y es hora de que tomemos medidas para protegerla. Trabajaré incansablemente para reunir a nuestros vecinos, organizaciones y voluntarios dispuestos a unirse a esta causa. Firmo este documento con determinación y la esperanza de que cada uno de nosotros pueda contribuir a la protección y preservación de Fondo de Bikini el lugar que todos llamamos hogar. El futuro de nuestra ciudad depende de nuestras acciones y estoy decidido a liderar el camino hacia un futuro más limpio y brillante bajo el mar. ¡Hagámoslo juntos! Firmado Patricio Estrella.

Ilícito

Yo, Patricio Estrella, me comprometo solemnemente a dejar todos mis bienes, tanto tangibles como intangibles, a mi gran amigo Bob Esponja, con efecto partir del próximo día Martes 31 de octubre. Con pleno conocimiento y entendimiento de mis acciones, establezco este compromiso como un acto de generosidad y amor hacia mi amigo Bob Esponja como muestra de mi aprecio y gratitud hacia él. Confío en que mis bienes serán utilizados de manera sabia y beneficiosa para Bob Esponja y la comunidad en general. Esta promesa refleja mi deseo sincero de contribuir al bienestar de Bob Esponja y dejar un legado que perdure en el tiempo. En este viernes 20 de octubre, yo, Patricio Estrella, dejo constancia de este compromiso ante testigos y con pleno conocimiento de sus implicaciones. Véase a continuación una lista de aquellos bienes que pasan a ser propiedad Bob Esponja: Mi hamaca favorita, mi televisión de 24 pulgadas, mi sofá de tela, la alfombra de lino, mi herradura de la suerte, mi cepillo de dientes, todos mis ahorros (15 dólares), y, por último, mi casaRoca.

Función de generación de textos alternativos

Para la generación de los textos alternativos se utilizan todas las posibles combinaciones de las X primeras filas de los documentos CSV de manera que si se quiere obtener 2^{20} textos se utilizaran todas las combinaciones de las 20 primeras líneas.

Este proceso se realiza mediante un iterador de 0 a 2^{x}, que posteriormente se convierte a binario de manera que sus dígitos representaran la columna escogida de cada fila.

```
iteracion = 0
for digito in binario
    texto = texto + csv[iteracion][digito]
    iteracion+1
end for
```

Se debe tener en cuenta que para que todos los textos generados tengan sentido por si mismos, todos deben tener el total de líneas posibles, lo que implica que el binario a partir del cual se construye el texto debe tener siempre 32 dígitos, rellenando para esto, siempre con ceros a la derecha hasta obtener un total de 32.

Ilustremos esto con un ejemplo de un caso concreto:

Si queremos generar 2^{10} textos deberíamos iterar desde el cero (0000000000) hasta el 2^{10} (1111111111) pero completando a la derecha con tantos ceros como líneas totales tenga el CSV de nuestro texto.

Para el binario -> 1010010011

Se utilizaría -> 10100100110000000000000000000000

De esta forma, una vez construida la parte del texto variable completará el resto del mismo utilizando siempre la primera columna.

Experimentación y análisis

Una vez explicado el funcionamiento y las peculiaridades del algoritmo a estudiar, podemos proceder con la experimentación y el análisis de este.

Para la realización de las pruebas se han tenido en cuenta una serie de consideraciones que creo necesario conocer antes de poder comenzar con el análisis de resultados.

Para las pruebas tanto de la implementación base como de la extendida se ha considerado que para cada texto original debe encontrarse debe encontrase una colisión, por lo de que de no encontrarse en el conjunto de $2^{m/2}$ textos, se incrementara en uno el exponente repitiendo la prueba hasta que se encuentre colisión, tal y como se explica en el subapartado de modificaciones del algoritmo base.

Desde ahora y en lo que resta de documento se hará referencia a los CSV de los cuales surgen los diferentes textos como conjunto de textos, ya que esto es lo que representan en este estudio.

La experimentación y análisis de resultados se dividirá en dos secciones, separando la experimentación base de la extendida, tal y como propone el guion de prácticas.

En ambas se realizarán las mismas pruebas, aunque variando el tamaño del conjunto de tests, es decir, el número de ejecuciones del programa para distintos conjuntos de textos ilícitos.

El estudio consistirá, principalmente, en la fluctuación del tiempo de búsqueda de la primera colisión para un conjunto de textos en función del hash utilizado, teniendo en cuenta que al aumentar el hash también se aumenta el numero de textos por los que se comienza la búsqueda y el tiempo de ejecución tiene en cuenta la construcción del diccionario de hashes de textos ilícitos.

En cuanto al dominio de estudio escogido, será para ambas experimentaciones de resúmenes desde 24 a 48 bits aumentando de 4 en 4. El motivo de esta elección se basa en la facilidad que supone visualmente aumentar el número de bits del hash indicando únicamente el numero de caracteres que se quiere en hexadecimal.

De esta forma pues al indicar que se aumente en 1 numero de caracteres en hexadecimal del hash se estará indicando a su vez que se quieren aumentar 4 bits.

En cuanto al rango de estudio, se estableció el limite en 48 bits puesto que, aunque únicamente mantengamos en memoria el un listado de equivalencias (binario - hash), cuando el hash supera esta cifra el listado estaría almacenando un total de 2^{26} como mínimo lo cual supera los 6 millones de elementos y en cuanto necesitase utilizar 2^{27} estaríamos hablando de mas de 12 millones, lo que, tras varias pruebas, descubrí que desbordaba por completo mi memoria RAM.

Experimentación Base

La experimentación base, en mi caso, consistirá en utilizar cinco conjuntos de textos ilícitos diferentes a comparar con un único conjunto de textos lícitos.

Al utilizar 5 ilícitos diferentes se podrá cotejar una media entre los resultados de las 5 ejecuciones, permitiendo evitar obtener un resultado poco fiable por un simple golpe de mala suerte.

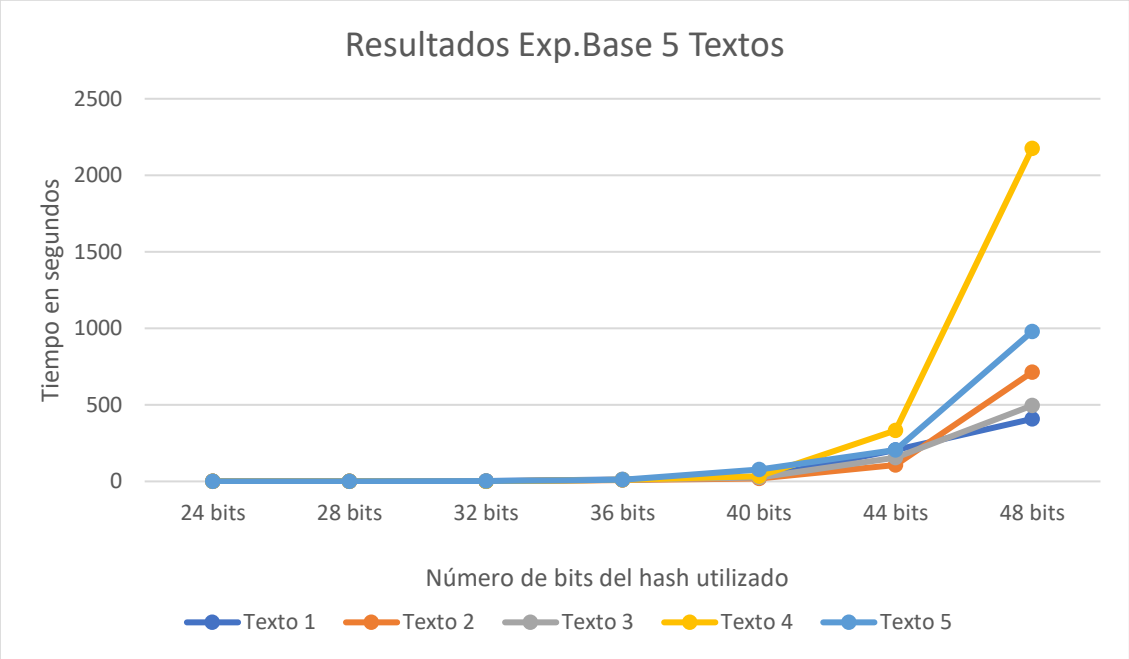


Ilustración 2. Grafico. Resultados experimentación base 5 textos

En esta primera grafica ya se puede observar el motivo de utilizar 5 conjuntos de textos en lugar de uno solo. El “Texto 4” tiene un tiempo de ejecución muy alto en comparación con los demás, por lo que de haber utilizado únicamente este texto nuestros resultados estarían muy desvirtuados.

Al hacer una media de los resultados este error se suaviza bastante, tal y como se puede observar en la siguiente grafica.

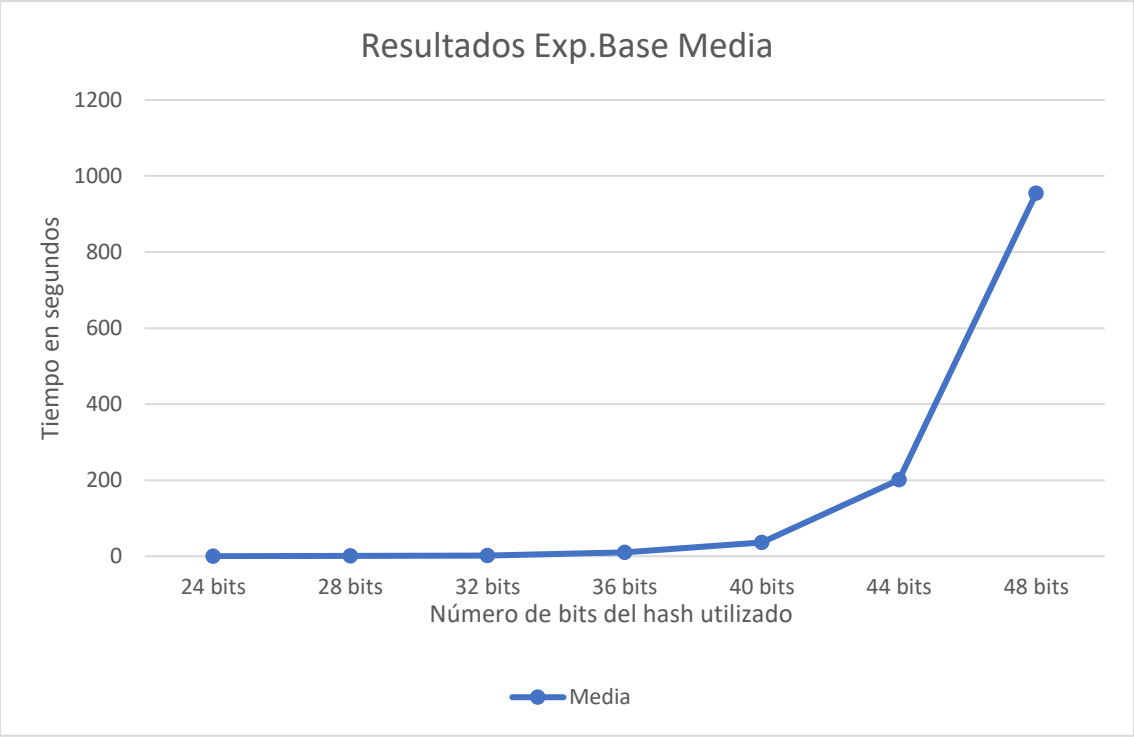


Ilustración 3. Grafico. Resultados experimentación base (Media)

Utilizando ahora la grafica generada por la media de los resultados, se puede observar un crecimiento exponencial, ya que es de esta forma como crecen tanto el diccionario a construir como el posterior espacio de búsqueda.

También considero interesante mostrar una tabla comparativa que muestra la cantidad de textos necesarios para encontrar la coincidencia en función del tamaño de hash, ya que permite ver claramente como se aplicaría la paradoja del cumpleaños.

Según la paradoja del cumpleaños para obtener colisión con un 50% de probabilidad el número de mensajes debe ser igual a $2^{m/2}$, siendo m el tamaño del hash en bits.

Véase a continuación una tabla aclarativa de los textos necesarios para encontrar coincidencia en función del tamaño del hash, siendo los valores mostrados el exponente de dos para cada caso.

	BASE	Texto 1	Texto 2	Texto 3	Texto 4	Texto 5
24 bits	12	13	13	12	12	12
28 bits	14	15	15	16	15	16
32 bits	17	16	16	16	16	16
36 bits	18	19	19	19	19	19
40 bits	20	20	20	20	21	22
44 bits	22	23	22	23	24	23
48 bits	24	24	25	24	26	25

Tabla 5. Textos necesarios para obtener colisión en función del tamaño de hash

A simple vista y de forma muy general, se puede observar que aproximadamente en la mitad de los casos se encuentra colisión para un numero de textos igual a $2^{tamañoHash/2}$, cumpliéndose de cierta forma el 50% de probabilidad que asegura la paradoja del cumpleaños.

También se puede observar en esta tabla el motivo del elevado tiempo de ejecución que presenta el texto 4 para hash de tamaño 48 bits, ya que al contrario que el resto textos, este necesita llegar a 2^{26}.

Experimentación Extendida

Para la experimentación extendida, en este caso se repetirán las pruebas, esta vez con 15 conjuntos de textos, lo que permitirá generar unos resultados mucho mas precisos que los anteriores.

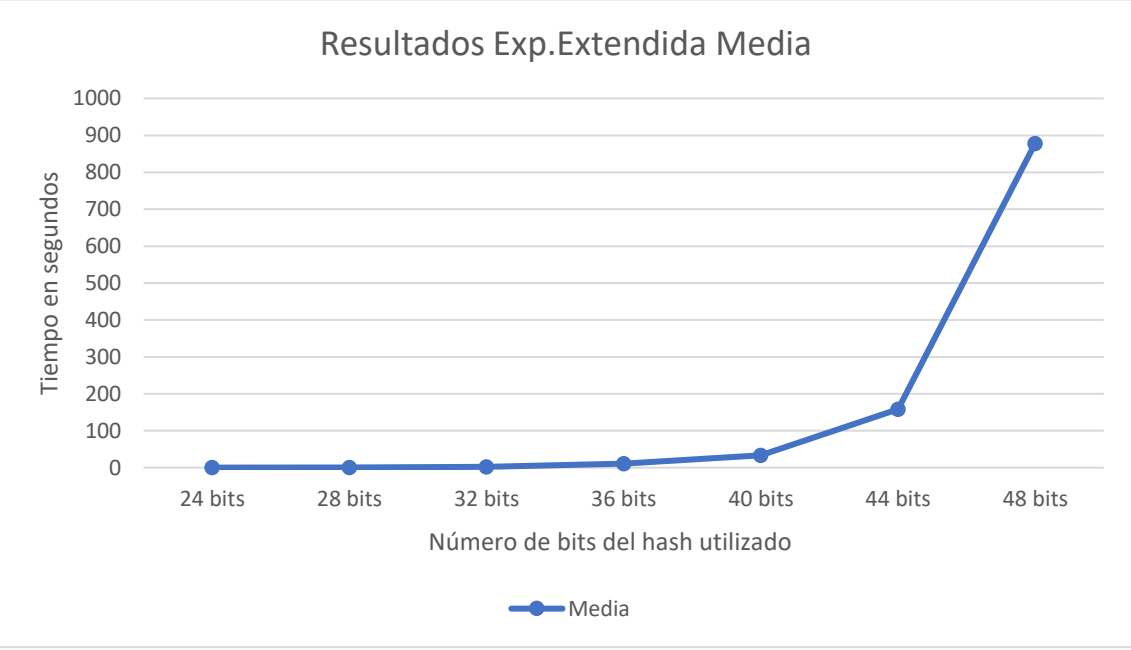


Ilustración 4. Grafico. Resultados experimentación extendida (Media)

En la tabla anterior se puede observar la media obtenida a partir de los 15 conjuntos de textos utilizados para estas pruebas. Si es cierto que esta grafica por separado no arroja demasiada información, por lo que, en la siguiente grafica se puede observar una comparativa de los resultados medios obtenidos con 5 textos y con 15.

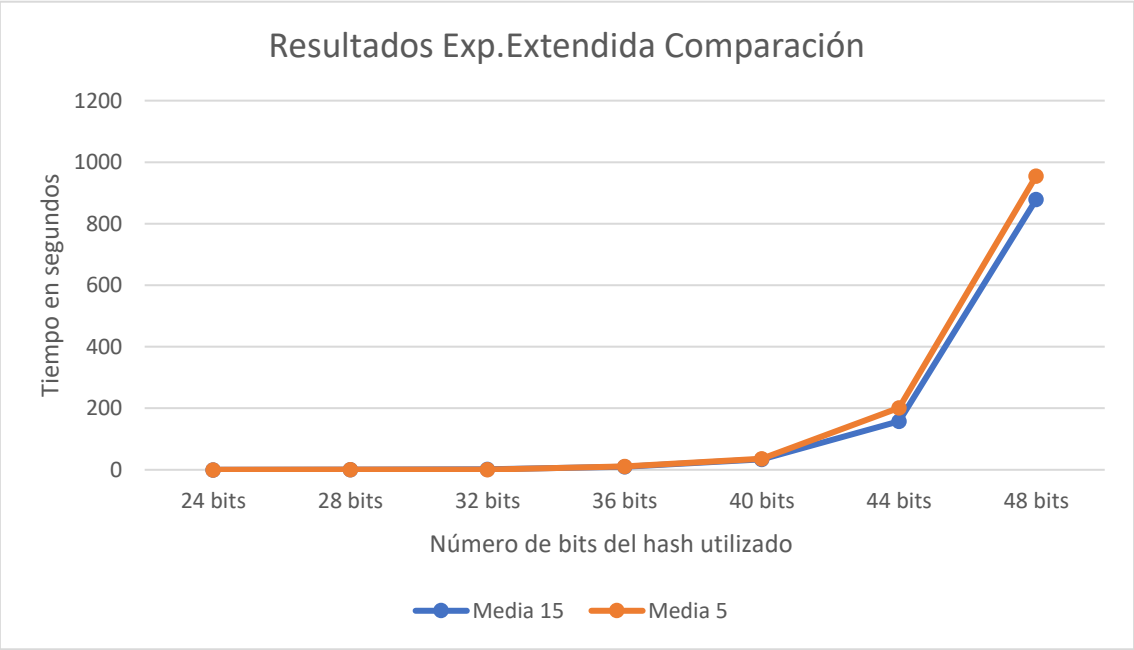


Ilustración 5. Grafico. Resultados experimentación extendida (Comparación)

Se observa en esta grafica como los tiempos de ejecución obtenidos mediante la media de 15 conjuntos de textos es ligeramente inferior a la obtenida con únicamente 5 textos, lo cual indica que los resultados que obteníamos antes estaban significativamente alterados al alza.

Implementación alternativa

Como ya se menciona en el apartado de experimentación, el motivo de realizar el estudio únicamente hasta tamaño de hash 48 se debe a la limitada capacidad de memoria RAM de la que dispongo, ya que, al tratar de almacenar 2^{27} hashes, esta se desborda.

Por esta razón he decidido enfocar la implementación alternativa como una modificación del algoritmo base de forma que permita trabajar con cualquier cantidad de textos sin importar las limitaciones de memoria.

Para esto se incluirá un condicional que indicará que para conjuntos de textos superiores a 2^{25} se utilizará una segunda función a la que he llamado "*yuvalAtomico*".

Esta función se encarga de comprobar los mismos textos que se harían de manera normal, pero particionando el conjunto inicial en X subconjuntos de tamaño fijo 2^{25}.

Explicación de la modificación

Para la explicación de esta modificación vamos a partir de que el tamaño del conjunto de textos depende del exponente que acompañe al 2, ya que este será el número de líneas a tener en cuenta. Sabiendo esto de aquí en adelante y durante el resto de la explicación se hablará directamente del exponente del 2 sin nombrarlo exponente.

Así pues, esta función procede de la siguiente forma:

1. Se obtiene el valor de cuanto supera el tamaño permitido p.g.(dif = 27 - 25).
2. Se realiza Yuval con conjuntos de tamaño 25, 2^{dif} veces.
3. Se termina por completo este ciclo en cuanto se encuentra una coincidencia.

La dificultad aquí está en la generación del binario, ya que este habrá que adaptarlo, construyendo primero el binario correspondiente de entre 0 y 2^{25} con exactamente 25 dígitos, luego se le añadirá a la izquierda el valor de la iteración actual en binario para finalizar rellenando con cerosa a la derecha hasta rellenar los 32 bits.

Explicado mediante un ejemplo práctico:

Se quiere ejecutar Yuval para 2^{28} textos.

Se obtiene que se supera el limite de antes marcado por 3, por lo que se realizaran un total de 2^3 (8) comparaciones con conjuntos de 2^{25} textos.

Supongamos que nos encontramos en la iteración 5 de las 8 a realizar y en una combinación cualquiera de las 2^{25} posibles.

por ejemplo (11010010101110111010011011)

Este número se completaría con el binario correspondiente a la iteración actual, en este caso, 5 (101), quedando tal que: 101 – 11010010101110111010011011.

Finalmente se completaría el numero con ceros a la derecha hasta obtener 32 bits, resultando en: 101 – 11010010101110111010011011 – 0000 pudiendo obtener el texto resultante de la combinación: 10111010010101101110100110110000.

Experimentación

En cuanto a la experimentación relativa a esta ampliación se han realizado las mismas pruebas que en los casos anteriores, pero esta vez pudiendo llegar hasta el hash de tamaño 52 bits, explorando conjuntos de textos de 2^{27}, cosa que antes no era posible en mi ordenador.

Resultados de 5 textos

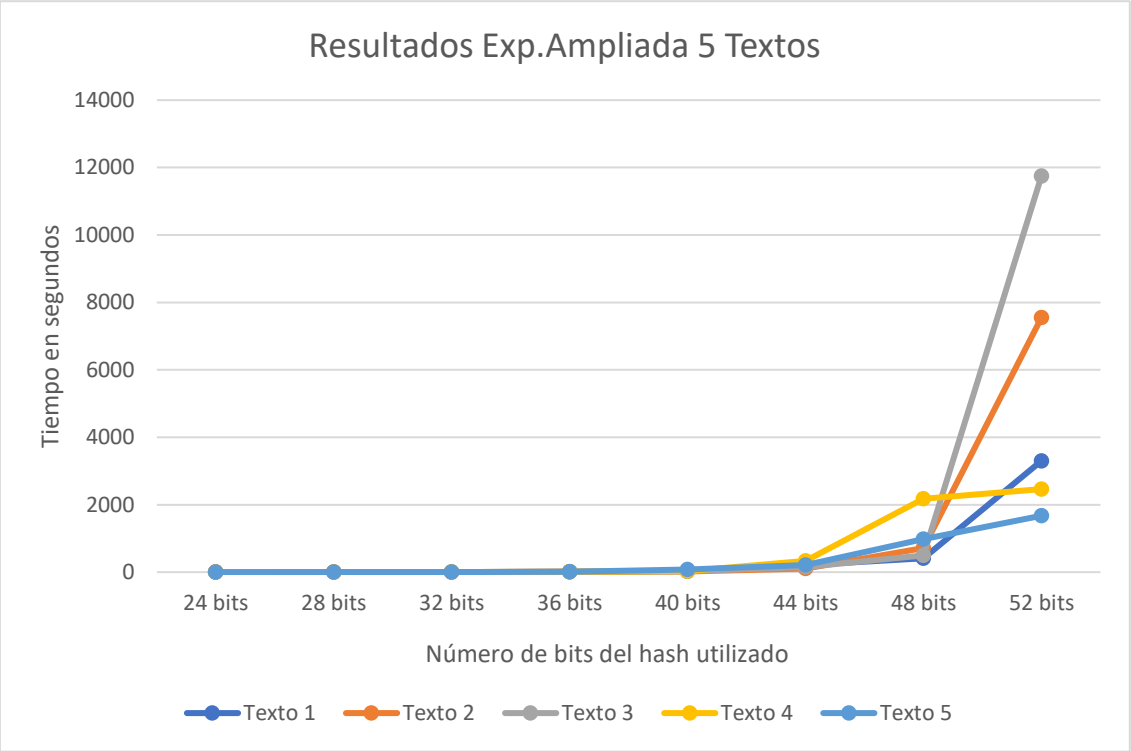


Ilustración 6. Grafico. Resultados experimentación ampliada 5 textos

En este grafico se observa claramente la gran velocidad a la que escala el tiempo respecto al tamaño del hash. Si nos fijamos en resultado del texto 4 para 48 bits de hash, que antes considerábamos anómalo debido a la gran diferencia con el resto de datos, vemos, no solo que ahora parece una diferencia minúscula, si no que para el siguiente tamaño de hash por pura estocástica obtiene un tiempo muy similar al obtenido para un hash más pequeño.

Resultados de 15 textos

Al igual que en el estudio anterior para el análisis de los resultados obtenidos por los 15 textos, se mostrará únicamente una comparación entre la media obtenida para 5 textos y la obtenida para 15.

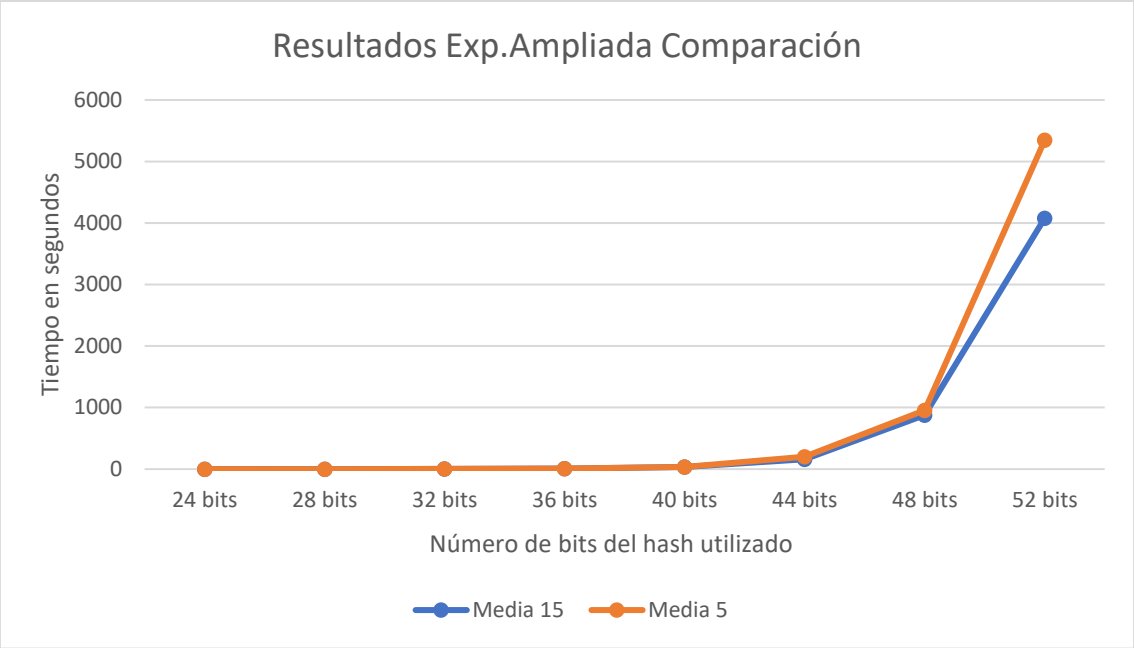


Ilustración 7. Grafico. Resultados experimentación ampliada (Comparación)

Se sigue observando como la media obtenida por 5 textos es mas elevada que la obtenida para 15, aunque en este caso se debe a los 11750 segundos de ejecución que registra el texto 3.

Otras pruebas

La ejecución de pruebas para tamaños superiores de hash ha sido inviable, ya que esta implementación permite ejecutar grandes cantidades de texto, pero no reduce el tiempo de ejecución de estas.

Solo se ha podido realizar una la ejecución para un texto con hash de 56 bits y ha registrado un tiempo de 135.696,3133 segundos, es decir, más de 37 horas, en un único texto, por lo que repetir esta ejecución para otro texto es inviable.

Otras ideas de ampliación

También se plantea la idea de generar ambos diccionarios completos antes de comparar, lo que aceleraría muchísimo la comparación. Pero esto solo seria asequible para valores de hash muy pequeños, por lo que descarte si quiera ponerlo a prueba.

La idea de implementar paralelización también fue descartada tras preguntar al profesor la validez de esta como ampliación.

Conclusiones

Quiero aprovechar este apartado para dar una visión personal acerca del estudio llevado a cabo más allá de comentar los resultados obtenidos, ya que esto ya se ha hecho en los respectivos apartados de experimentación de cada uno de los algoritmos estudiados.

He de confesar que, en un principio, la idea de que fuese posible romper una función de resumen y driblar con esto una contraseña o falsificar un texto con la firma de otro, me aterraba.

No obstante, tras tratar de analizar estos algoritmos y enfrentarme no solo a los problemas ocasionados hasta conseguir obtener un código funcional, sino también a los enormes tiempos de ejecución requeridos para conseguir romper funciones de resumen de únicamente 56bits, he podido comprender la seguridad que suponen realmente estas funciones con únicamente resúmenes de 128 bits, por no hablar de aquellas que superan este tamaño.

Repositorio GITHUB

En este repositorio se encuentra el código completo de ambos algoritmos, así como los resultados obtenidos para cada una de las pruebas mostradas en este documento

Repositorio GitHub: https://github.com/Javi-GI/Practica1CSD/tree/main/Practica_1

Bibliografía

Guion de la práctica, https://poliformat.upv.es/access/content/group/DOC_34876_2023/Trabajos/MUCC%20-%20Criptologia%20-%20Integridad.pdf

Tema 2, Criptología. Integridad de la Información, https://poliformat.upv.es/access/content/group/DOC_34876_2023/Transparencias/MUCC%20-%20Criptolog%C3%ADa%20Integridad.pdf

Manual CRC32, https://www.php.net/manual/es/function.crc32.php

MD5,https://es.wikipedia.org/wiki/MD5#:~:text=En%20criptograf%C3%ADa%2C%20MD5%20(abreviatura%20de,archivo%20no%20haya%20sido%20modificado.

Apéndices

arcoirisbase.py

```
from zlib import crc32
import random
import pandas as pd
import time
import csv

caracteresCadena = '0123456789'

def cargarCSV(ficherito):
    textoCargado = []
    with open(ficherito, newline='') as csvfile:
        texto = csv.reader(csvfile, delimiter=';')
        for row in texto:
            textoCargado.append(row)
    return textoCargado

def h(input):
    input = bytes(input, encoding='utf-8')
    result = crc32(input)
    return result

def r(x, index):
    Password = x%1000000
    return str(Password)


def busqueda(tabla, listaPassword, t, n):
    resultados = []
    for password in listaPassword:
        resultado = [0,0,0]
        pw = password[0]
        #print(pw)
        original = pw
        p0 = h(original)
        p = p0
        #print(p0)

        for i in range(t):
            if p in tabla:
                break
            p = h(r(p, i))
```

```
        if i >= t - 1:
            print("No se encontro coincidencia.")

        else:
            pwd = tabla[p]
            print("La contraseña coincidente es: " + pwd)
            tiempo_inicio = time.time()
            tiempo_transcurrido = 0
            hPassword = h(pwd)
            i = 0
            while hPassword != p0 and i < n*t: #tiempo_transcurrido < tiempo_limite:
                password = r(hPassword, i)
                hPassword = h(password)
                tiempo_transcurrido = time.time() - tiempo_inicio
                i = i+1

            if i < n*t: #tiempo_transcurrido < tiempo_limite:
                print("Encontrada coincidencia en hash " + str(hPassword) + " con contraseña
" + password + " ----------------------------------")
                resultado = [original,pwd,password,hPassword]

            else:
                print("No se encontro en el tiempo permitido.")
                resultado = [original,pwd,0,0]
        resultados.append(resultado)

    return resultados

def main():
    listaResultados = []

    for i in range(100):
        t= 200
        n= 5000
        tabla = {}
        print("------ Nueva prueba ------")
        while len(tabla) < n:
            #Password al azar
            pi = ''.join(random.choice(caracteresCadena) for _ in range(6))
            p = pi

            for j in range(t - 1):
                p = r(h(p), j)

            tabla[h(p)] = pi
            #print(len(tabla))
        print("------ Tabla Creada ------")
```

```
    passwordsList = cargarCSV("basePassword.csv")

    #print(passwordsList)

    listaResultados.append(["------ Tamaño de tabla " + str(n) +  " X " + str(t) + " ------"])
    listaResultados.append(["Original","Colision","Equivalente","Hash"])
    resultados = busqueda(tabla, passwordsList, t, n)
    listaResultados = listaResultados + resultados

    outputArcoiris = 'outputArcoirisBasePrueba.csv'
    with open(outputArcoiris, 'w', newline="") as file:
      csvwriter = csv.writer(file)
      csvwriter.writerows(listaResultados)

main()
```

arcoíris.py

```
from zlib import crc32
import random
import pandas as pd
import time
import csv

caracteresCadena = 'abcdefghijklmnopqrstuvwxyz'
#abcdefghijklmnopqrstuvwxyz

def cargarCSV(ficherito):
    textoCargado = []
    with open(ficherito, newline='') as csvfile:
        texto = csv.reader(csvfile, delimiter=';')

        for row in texto:
            textoCargado.append(row)

    return textoCargado

def h(input):
    input = bytes(input, encoding='utf-8')
    result = crc32(input)
    return result


def r(x, index):
    caracteres = caracteresCadena
    password = (bin(x)[2:]).zfill(32)

    letra1 = caracteres[int(password[0:8], 2)%len(caracteres)]
    letra2 = caracteres[int(password[6:14], 2)%len(caracteres)]
    letra3 = caracteres[int(password[12:20], 2)%len(caracteres)]
    letra4 = caracteres[int(password[18:26], 2)%len(caracteres)]
    letra5 = caracteres[int(password[24:32], 2)%len(caracteres)]

    Password = letra1 + letra2 + letra3 + letra4 + letra5

    return Password

def busqueda(tabla, listaPassword, t, n):
    resultados = []
    for password in listaPassword:
        resultado = [0,0,0]
        pw = password[0]
```

```
        #print(pw)
        original = pw
        p0 = h(original)
        p = p0
        #print(p0)

        for i in range(t):
            if p in tabla:
                break
            p = h(r(p, i))
            #print("p: " + str(p))

        if i >= t - 1:
            print("No se encontro coincidencia.")

        else:
            pwd = tabla[p]
            print("La contraseña coincidente es: " + pwd)
            tiempo_inicio = time.time()
            tiempo_transcurrido = 0
            hPassword = h(pwd)
            i = 0
            while hPassword != p0 and i < n*t: #tiempo_transcurrido < tiempo_limite:
                password = r(hPassword, i)
                hPassword = h(password)
                tiempo_transcurrido = time.time() - tiempo_inicio
                i = i+1

            if i < n*t: #tiempo_transcurrido < tiempo_limite:
                print("Encontrada coincidencia en hash " + str(hPassword) + " con contraseña
" + password + " ----------------------------------")
                resultado = [original,pwd,password,hPassword]

            else:
                print("No se encontro en el tiempo permitido.")
                resultado = [original,pwd,0,0]
        resultados.append(resultado)


    return resultados

def main():
    listaResultados = []
```

```
    columnas = [50, 50, 50, 50, 100, 100, 100, 100, 200, 200, 200, 200, 300, 300, 300,
300]
    filas = [5000, 10000, 30000, 50000, 5000, 10000, 30000, 50000, 5000, 10000, 30000,
50000, 5000, 10000, 30000, 50000]

    for i in range(10,len(filas)):
        t= columnas[i]
        n= filas[i]
        tabla = {}
        print("------ Nueva prueba ------")
        while len(tabla) < n:
            #Password al azar
            pi = ''.join(random.choice(caracteresCadena) for _ in range(6))
            p = pi

            for j in range(t - 1):
                p = r(h(p), j)

            tabla[h(p)] = pi
            #print(len(tabla))
        print("------ Tabla Creada ------")
        #print(tabla)

        passwordsList = cargarCSV("password.csv")

        #print(passwordsList)

        listaResultados.append(["----- Tamaño de tabla " + str(n) +  " X " + str(t) + " ------"])
        listaResultados.append(["Original","Colision","Equivalente","Hash"])
        resultados = busqueda(tabla, passwordsList, t, n)
        listaResultados = listaResultados + resultados

        outputArcoiris = 'outputArcoirisPrueba.csv'
        with open(outputArcoiris, 'w', newline="") as file:
            csvwriter = csv.writer(file)
            csvwriter.writerows(listaResultados)

main()
```

yuval.py

```
import pandas as pd
import csv
import hashlib
import time

def cargarCSV(ficherito):
    textoCargado = []
    with open(ficherito, newline='') as csvfile:
        texto = csv.reader(csvfile, delimiter=';')

        for row in texto:
            textoCargado.append(row)

    return textoCargado

def convertToBin(numero, tam):
    binario = bin(numero)[2:]
    binario = binario.zfill(tam)
    binario = binario.ljust(32, "0")
    binarioFinal = bin(int(binario,2))[2:].zfill(len(binario))
    return binario

def cargarTexto(csvTexto, binIt):
    texto = " "
    it = 0
    for x in binIt:
        #print(x)
        texto = texto + csvTexto[it][int(x)]
        it=it+1
    return texto

def generarTextosIlicitos(ilicitoCSV, tam, num):
    textNum = pow(2,num)
    ilicitoDic = {}
    for i in range(0,textNum):
        binIt = convertToBin(i, num)
        textoIlicito = cargarTexto(ilicitoCSV, binIt)
        huella = hash_variable_length(textoIlicito, tam)
        ilicitoDic[huella] = binIt

    return ilicitoDic
```

```
def hash_variable_length(data, length_digit):
    md5_hash = hashlib.md5(data.encode()).hexdigest()
    # Trunca el hash al número de bytes especificado
    truncated_hash = md5_hash[:length_digit]

    return truncated_hash

def comparar(licitoCSV, ilicitoCSV, tam, num):
    startTime = time.perf_counter()
    listaIlicitos = generarTextosIlicitos (ilicitoCSV, tam, num)
    print("Textos Ilicitos Generados: " + str(num))
    dicNum = len(listaIlicitos)
    cantidad = 0
    textNum = pow(2,num)
    resultados = [0,0,0,0.0]
    tiempo = 0.0
    for x in range(0,textNum): #recorre textos licitos
        binDic = convertToBin(x, num)
        textoLicito = cargarTexto(licitoCSV, binDic)
        huella = hash_variable_length(textoLicito,tam)
        del textoLicito
        if huella in listaIlicitos:
            endTime = time.perf_counter()
            tiempo = endTime-startTime
            resultados=[num, binDic, listaIlicitos[huella],tiempo]
            print("Hay coincidencia: Licito: " + binDic + " e Ilicito: " + listaIlicitos[huella] + "
con hash: " + huella + " en tiempo: " + str(tiempo))
            break
    del listaIlicitos
    return resultados

def doMedia(resultados, div):
    suma = 0.0
    for x in range(0, div):
        suma = suma + resultados[x][3]
    media = suma/div
    return media

def main():
    num = 20
    resultadosList=[]
    tituloLicito = "licito.csv"
    licitoCSV = cargarCSV(tituloLicito)

    for x in range(6,13):
        resultados = []
        print("Pruebas para Hash: " + str(x))
```

```
        resultadosList.append(['-------- Resultados para Hash: ' + str(x*4) + ' --------'])
        resultadosList.append(["Tamanio Texto", "Licito", "Ilicito", "Tiempo"])

        for y in range(1,16):
            print("Texto " + str(y))
            tituloIlicito = "ilicito" + str(y) + ".csv"
            ilicitoCSV = cargarCSV(tituloIlicito)
            encontrado = False
            num = x*2
            while not encontrado and num < 33:
                resultado = comparar(licitoCSV,ilicitoCSV,x,num)
                if resultado[3] != 0.0:
                    resultados.append(resultado)
                    #print(resultado)
                    resultadosList.append(resultado)
                    encontrado = True
                num = num + 1
            del ilicitoCSV
        mediacinco = doMedia(resultados, 5)
        media = doMedia(resultados, len(resultados))

        print("La media es: " + str(media))

        resultadosList.append(["La media para 5 es de: " + str(mediacinco)])
        resultadosList.append(["La media total es de: " + str(media)])
        resultadosList.append(" ")
        outputYuval = 'outputPrueba.csv'
        with open(outputYuval, 'w', newline="") as file:
            csvwriter = csv.writer(file)
            csvwriter.writerows(resultadosList)

main()
```

yuvalAtomico.py

```
import pandas as pd
import csv
import hashlib
import time

def cargarCSV(ficherito):
    textoCargado = []
    with open(ficherito, newline='') as csvfile:
        texto = csv.reader(csvfile, delimiter=';')

        for row in texto:
            textoCargado.append(row)

    return textoCargado

def convertToBinAtomico(numero, tam, dif, ini):
    binario = bin(numero)[2:]
    binario = binario.zfill(tam-dif)
    binario = binario.ljust(32-dif, "0")
    binario = str(ini) + binario
    binarioFinal = bin(int(binario,2))[2:].zfill(len(binario))
    return binarioFinal

def convertToBin(numero, tam):
    binario = bin(numero)[2:]
    binario = binario.zfill(tam)
    binario = binario.ljust(32, "0")
    binarioFinal = bin(int(binario,2))[2:].zfill(len(binario))
    return binarioFinal

def cargarTexto(csvTexto, binIt):
    texto = " "
    it = 0
    for x in binIt:
        #print(x)
        texto = texto + csvTexto[it][int(x)]
        it=it+1
    return texto

def generarTextosIlicitos(ilicitoCSV, tam, num):
    textNum = pow(2,num)
    ilicitoDic = {}
    for i in range(0,textNum):
        binIt = convertToBin(i, num)
        textoIlicito = cargarTexto(ilicitoCSV, binIt)
```

```
        huella = hash_variable_length(textoIlicito, tam)
        ilicitoDic[huella] = binIt

    return ilicitoDic

def generarTextosIlicitosAtomico(ilicitoCSV, tam, num, it):
    textNum = pow(2,num)
    dif = num - 6
    ilicitoDic = {}
    ini = bin(it)[2:]
    ini = ini.zfill(dif)
    for i in range(0,textNum):
        binIt = convertToBinAtomico(i, num, dif, ini)
        textoIlicito = cargarTexto(ilicitoCSV, binIt)
        huella = hash_variable_length(textoIlicito, tam)
        ilicitoDic[huella] = binIt

    return ilicitoDic

def hash_variable_length(data, length_digit):
    md5_hash = hashlib.md5(data.encode()).hexdigest()
    # Trunca el hash al número de bytes especificado
    truncated_hash = md5_hash[:length_digit]

    return truncated_hash

def comparar(licitoCSV, ilicitoCSV, tam, num):
    startTime = time.perf_counter()
    listaIlicitos = generarTextosIlicitos (ilicitoCSV, tam, num)
    print("Textos Ilicitos Generados: " + str(num))
    textNum = pow(2,num)
    resultados = [0,0,0,0.0]
    tiempo = 0.0
    for x in range(0,textNum): #recorre textos licitos
        binDic = convertToBin(x, num)
        textoLicito = cargarTexto(licitoCSV, binDic)
        huella = hash_variable_length(textoLicito,tam)
        del textoLicito
        if huella in listaIlicitos:
            endTime = time.perf_counter()
            tiempo = endTime-startTime
            resultados=[num, binDic, listaIlicitos[huella],tiempo]
            print("Hay coincidencia: Licito: " + binDic + " e Ilicito: " + listaIlicitos[huella] + "
con hash: " + huella + " en tiempo: " + str(tiempo))
            break
    del listaIlicitos
    return resultados
```

```
def compararAtomico(licitoCSV, ilicitoCSV, tam, num):
    if(num >= 26):
        dif = num-6
        startTime = time.perf_counter()
        for i in range(0, pow(2,dif)):
            print("Texto " + str(i+1) + " de tamaño total " + str(num))
            listaIlicitos = generarTextosIlicitosAtomico(ilicitoCSV, tam, num, i)
            print("Textos Ilicitos Generados: " + str(num))
            textNum = pow(2,num)
            resultados = [0,0,0,0.0]
            tiempo = 0.0
            for x in range(0,textNum): #recorre textos licitos
                binDic = convertToBin(x, num)
                textoLicito = cargarTexto(licitoCSV, binDic)
                huella = hash_variable_length(textoLicito,tam)
                del textoLicito
                if huella in listaIlicitos:
                    endTime = time.perf_counter()
                    tiempo = endTime-startTime
                    resultados=[num, binDic, listaIlicitos[huella],tiempo]
                    print("Hay coincidencia: Licito: " + binDic + " e Ilicito: " + listaIlicitos[huella]
+ " con hash: " + huella + " en tiempo: " + str(tiempo))
                    del listaIlicitos
                    return resultados
        del listaIlicitos
        return resultados

    else:
        resultados = comparar(licitoCSV,ilicitoCSV,tam,num)

    return resultados

def doMedia(resultados, div):
    suma = 0.0
    for x in range(0, div):
        suma = suma + resultados[x][3]
    media = suma/div
    return media

def main():
    num = 20
    resultadosList=[]
    tituloLicito = "licito.csv"
    licitoCSV = cargarCSV(tituloLicito)

    for x in range(13,17):
```

```
        resultados = []
        print("Pruebas para Hash: " + str(x))

        resultadosList.append(['-------- Resultados para Hash: ' + str(x*4) + ' --------'])
        resultadosList.append(["Tamanio Texto", "Licito", "Ilicito", "Tiempo"])

        for y in range(1,16):
            print("Texto " + str(y))
            tituloIlicito = "ilicito" + str(y) + ".csv"
            ilicitoCSV = cargarCSV(tituloIlicito)
            encontrado = False
            num = x*2
            while not encontrado and num < 33:
                resultado = compararAtomico(licitoCSV,ilicitoCSV,x,num)
                if resultado[3] != 0.0:
                    resultados.append(resultado)
                    #print(resultado)
                    resultadosList.append(resultado)
                    encontrado = True
                num = num + 1
            del ilicitoCSV
        mediacinco = doMedia(resultados, 5)
        media = doMedia(resultados, len(resultados))

        print("La media es: " + str(media))

        resultadosList.append(["La media para 5 es de: " + str(mediacinco)])
        resultadosList.append(["La media total es de: " + str(media)])
        resultadosList.append(" ")
        outputYuval = 'outputAtomico.csv'
        with open(outputYuval, 'w', newline="") as file:
            csvwriter = csv.writer(file)
            csvwriter.writerows(resultadosList)

main()
```