



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

 etsinf

Escola Tècnica
Superior d'Enginyeria
Informàtica

Escuela Técnica Superior de Ingeniería Informática
Universidad Politécnica de Valencia

Adivinando passwords
Una propuesta para su búsqueda eficiente
TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Alejandro Mor Michael
Tutor: Damián López Rodríguez

Curso 2018 - 2019

Resumen

El acceso a los sistemas informáticos está desde siempre ligado a la utilización de palabras de paso o passwords. Por motivos de seguridad, los passwords se han almacenado de forma oculta en los sistemas, siendo habitualmente el resultado de la aplicación de una función resumen -o hash- sobre el password. Dichas funciones resumen tienen una gran relevancia para el mantenimiento seguro de los passwords. También son invertibles, con una probabilidad de colisión inversamente proporcional de forma exponencial al número de bits del resumen. Una aproximación para encontrar dichas colisiones se basa en la construcción de las denominadas tablas del arco iris, que emplean una aproximación *time-memory trade-off* (TMTO), mostrándose eficientes a la hora de encontrar colisiones y posibilitando el acceso no autorizado a los sistemas.

Palabras clave: password, hash, tabla del arco iris

Resum

L'accés als sistemes informàtics ha estat des-de sempre lligat a l'ús de paraules de pas o passwords. Per motius de seguretat, els passwords són emmagatzemats de forma oculta als sistemes, seguint habitualment el resultat de l'aplicació d'una funció resum -o hash- sobre el password. Aquestes funcions resum tenen una gran rellevància a l'hora de mantindre els passwords segurament. També són invertibles, amb una probabilitat de col·lisió inversament proporcional exponencialment al nombre de bits del resum. Una aproximació per a encontrar dites col·lisions són les denominades taules *rainbow*, que fan ús d'una aproximació *time-memory trade-off* (TMTO), mostrant-se eficients a l'hora d'encontrar col·lisions i possibilitant l'accés no autoritzat als sistemes.

Paraules clau: password, hash, taula rainbow

Abstract

Access to computer systems has always been tied to the use of passwords. For security reasons, passwords are stored in an occult manner, being usually the result of a hash function on the password. Said hash functions are highly relevant for safe-keeping passwords. They are also reversible, having a collision probability inversely proportional exponentially to the number of bits in the hash. An approximation for finding such collisions is based in the generation of the so called rainbow tables, which make use of a time-memory trade-off (TMTO), showing efficiency when looking for those collisions and allowing unauthorised access to the systems.

Key words: password, hash, rainbow table

Índice general

Índice general	V	
Índice de figuras	VII	
Índice de tablas	VII	
<hr/>		
1 Introducción	Más extenso... uso habitual, breve historia, por qué proporcionan seguridad, líneas de trabajo futuro (sha-3), cómo se evitan hoy ataques de este tipo; por qué es interesante	1
1.1 Motivación	1	1
1.2 Objetivos	aún así estudiar el problema	1
1.3 Estructura de la memoria	2
2 Buscando confidencialidad: funciones resumen		3
2.1 Definición	3
2.2 Propiedades	3
2.3 Contramedidas de una función <i>hash</i> para inutilizar el ataque del arco iris	4
2.4 Función atacada: CRC-32	5
3 Antecesores al ataque del arco iris		7
3.1 Primera aproximación: <i>time-memory trade-off</i>	7
3.2 Puntos distinguidos	9
4 El ataque del arco iris		11
4.1 Generación de tablas del arco iris	12
4.2 Generación de contraseñas aleatorias	17
4.3 Primeras pruebas	17
4.4 En busca de la configuración perfecta	21
4.5 Resultados finales	25
5 Conclusiones		31
Bibliografía		37
<hr/>		
Apéndice		
A Configuración del sistema		39

Índice de figuras

3.1	Computación esquematizada de las tablas de Hellman en su TMTO	8
4.1	Porcentajes de éxito para las tablas empleando R1	18
4.2	Porcentajes de éxito para las tablas empleando R2	19
4.3	Porcentajes de éxitos para las tablas empleando R1 + R2	19
4.4	Porcentajes de éxito para las tablas empleando el patrón grande . . .	20
4.5	Porcentajes de éxito para las tablas empleando el patrón pequeño .	21
4.6	Porcentajes de colisiones para las tablas empleando R1	23
4.7	Porcentajes de colisiones para las tablas empleando R2	23
4.8	Porcentajes de colisiones para las tablas empleando R1 + R2	24
4.9	Porcentajes de colisiones para las tablas empleando el patrón grande .	24
4.10	Porcentajes de colisiones para las tablas empleando el patrón pe- queño	25
4.11	Porcentajes de éxito para las tablas que emplean R1	26
4.12	Porcentajes de éxito para las tablas que emplean R2	27
4.13	Porcentajes de éxito para las tablas que emplean R1 + R2	27
4.14	Porcentajes de éxito para las tablas que emplean el patrón grande .	28
4.15	Porcentajes de éxito para las tablas que emplean el patrón pequeño	29

Índice de tablas

CAPÍTULO 1

Introducción

Un poco brusca y muy breve la intro... alguien que no este en el tema lo encontrará raro...

El uso de funciones resumen en criptografía está basado en la complejidad de encontrar colisiones para un resumen dado. Dicha complejidad hace inviables los ataques basados en fuerza bruta, por lo que si se pretende obtener resultados satisfactorios, han de llegar necesariamente desde mejores aproximaciones. Es por ello que con el tiempo han surgido varias de estas aproximaciones capaces de mejorar los resultados de la fuerza bruta de forma significante, haciendo viable la búsqueda de colisiones para un resumen ya sabido.

1.1 Motivación

El hecho de que las contraseñas traten de ocultarse empleando técnicas criptográficas no ha supuesto un impedimento para su obtención no autorizada. En la mayoría de casos, se podría decir que tan sólo ha servido para retrasar lo inevitable, ya que ha sido posible implementar ataques que han conseguido hacerse con ellas, demostrando fallos de seguridad en las técnicas criptográficas empleadas. En concreto, la obtención de contraseñas para el acceso a un sistema operativo siempre ha resultado ser de mucho interés para atacantes malintencionados, debido a la magnitud de la posible recompensa una vez conseguido llevar a cabo su ataque con éxito. [El ataque del arco iris](#) aquí presentado demuestra la debilidad de algunas funciones resumen empleadas para dicha ocultación de contraseñas, así como la utilidad que puede tener realizar una implementación como la que se va a presentar.

[Todavía no has dicho nada de lo que es un ataque arco Iris](#)

1.2 Objetivos

El objetivo principal de este trabajo consiste en la obtención del mayor número posible de contraseñas almacenadas en un sistema operativo. Para ello, la secuencia de pasos a realizar es la siguiente:

[Más general... \(estudiar posibles mejoras en eficiencia de sistemas de búsqueda de colisiones en funciones hash\)](#)

1. Generación de diversas tablas del arco iris, teniendo cada una diferentes parámetros.

2. Generación de 1.000 resúmenes *hash* aleatorios, simulando ser las contraseñas almacenadas en el sistema operativo.
3. Experimentación con las tablas sobre las contraseñas, implementando el ataque del arco iris con cada una de ellas para obtener los resultados correspondientes.
4. Interpretación de los resultados obtenidos para determinar las mejores tablas en cuanto a calidad de resultados y tamaño de tabla.

1.3 Estructura de la memoria

En primer lugar se introducirá el concepto de funciones resumen o funciones *hash*, su funcionamiento, aplicaciones, debilidades y fortalezas frente a este tipo de ataque. Seguidamente se presentarán las técnicas originales que con el tiempo dieron lugar al ataque del arco iris y sus primeras implementaciones. A continuación se dará una extensa explicación sobre el ataque del arco iris en sí, con todos los factores que se han de tener en cuenta a la hora de su desarrollo e implementación, seguido de la aplicación del ataque del arco iris llevada a cabo en este proyecto, la cual será presentada en detalle, indicando los factores tenidos en cuenta para ello y relacionándolos con lo anteriormente explicado. A ello le seguirá una experimentación y su posterior interpretación, para concluir cuál resulta ser la mejor aproximación. Finalmente, se realizarán las últimas conclusiones obtenidas después del desarrollo completo del trabajo.

CAPÍTULO 2

Buscando confidencialidad: funciones resumen

2.1 Definición

Las funciones resumen, también denominadas funciones *hash*, son aquellas que actúan sobre mensajes de longitud arbitraria, tomándolos como entrada y produciendo como salida una cadena de longitud fija. A dicha salida producida por una función *hash* se le denomina un resumen -o valor- *hash*, también pudiendo referirse a ella simplemente como un *hash*.

[Algo de historia, esquemas de construcción, funciones clásicas, uso de estas, estado actual... busca Menezes “handbook of applied cryptography”..](#)

2.2 Propiedades

Una de las principales características de los valores *hash* producidos por una función resumen es que, al ser de longitud fija, no suelen ser únicos para cada mensaje de entrada, por lo cual suelen existir colisiones dentro de una misma función *hash*. Una colisión se produce cuando para dos mensajes distintos, sean m_1 y m_2 , el valor *hash* de ambos es el mismo. Para una función *hash* que produzca como salida cadenas binarias de longitud n , de acuerdo con una distribución uniforme, existen un total de 2^n valores *hash* diferentes, por lo cual la probabilidad de que dos mensajes colisionen es igual a $\frac{1}{2^n}$.

En su aplicación a la criptografía, las funciones resumen son consideradas probablemente seguras cuando es inviable obtener una colisión para un *hash* conocido. Es decir, si un atacante obtiene un *hash*, sea h , no le es posible obtener un mensaje m que produzca el mismo valor *hash* h de manera eficiente. En este caso, la inviabilidad se define como la imposibilidad de obtener una colisión en menor tiempo de lo que lo haría un ataque de fuerza bruta. Siguiendo esta lógica, cualquier función *hash* para la cual exista una forma eficiente de obtener colisiones (es decir, más rápida que la fuerza bruta) será considerada insegura o rota.

Algunas otras características importantes de las funciones *hash* son las siguientes:

- **Computación eficiente:** obtener un resumen *hash* para un mensaje de entrada dado se realiza rápidamente. Aunque parezca una necesidad obvia, es de vital importancia para el uso de funciones *hash* en protocolos criptográficos.
- **Determinismo:** un mensaje de entrada dado siempre producirá la misma cadena de salida.
- **Uniformidad:** los posibles valores *hash* de una función resumen deben de tener la misma probabilidad de ser generados.
- **Altamente susceptibles:** dados un mensaje m y su correspondiente valor *hash* h_m , alterar algunos bits de m genera un valor *hash* h'_m el cual es tan diferente a h_m que no parecen estar relacionados, es decir, comparando ambos valores *hash* no es posible apreciar que provienen de mensajes muy similares.

A la hora de determinar el nivel de seguridad correspondiente a una función *hash*, es suficiente con determinar si dicha función es resistente a los siguientes tipos de ataques:

1. **Pre-imagen:** Encontrado un valor *hash* h , es inviable encontrar el mensaje m del cual ha sido generado.
2. **Segunda pre-imagen:** Dado un mensaje de entrada m , es inviable averiguar otro mensaje de entrada diferente m' de tal forma que el valor *hash* producido por ambos mensajes sea el mismo.
3. **Colisión:** Resulta inviable encontrar dos mensajes diferentes, m_1 y m_2 , los cuales generan el mismo valor *hash*. [Collision resistant hash function CRHF](#)

Cabe remarcar que en cuanto a estas resistencias, la diferencia entre la resistencia a segunda pre-imagen y a colisión se debe a que en la segunda pre-imagen, el primer mensaje ha sido interceptado por un atacante, mientras que en la colisión no se ha obtenido ningún mensaje de entrada.

[Las contramedidas no es algo propio de la función resumen, es añadido en la estrategia del uso](#)

2.3 Contramedidas de una función *hash* para inutilizar el ataque del arco iris

Aunque el ataque del arco iris tiene una efectividad muy alta cuando se implementa de la forma correcta, existen escenarios en los cuales resulta inviable implementar el ataque o directamente imposible:

- **Salting:** la técnica conocida como *salting* consiste en añadir una cadena aleatoria no secreta (*salt*) a la contraseña antes de ser resumida o directamente al valor *hash* resultante. Por ejemplo, si se partiera con la contraseña p , el *salt* s y la operación de concatenación $+$, sería posible obtener como valores *hash* tanto $h = \text{HASH}(p + s)$ como $h = \text{HASH}(p) + s$. Para llevar a cabo

el ataque del arco iris con éxito frente a funciones *hash* que hacen uso de esta técnica es necesario generar tablas del arco iris para cada *salt* posible, lo cual para tamaños de *salt* pequeños no supone un problema serio, pero en cuanto se alcanza cierto tamaño de *salt* resulta inviable implementar el ataque debido a la enorme cantidad de tablas necesarias.

- **Estiramiento:** esta técnica hace uso del *salting*, concatenándolo a la contraseña antes de generar el valor *hash* correspondiente. Tras obtener dicho *hash* se repite el mismo proceso un número de veces determinado, esta vez concatenando el *salt* al valor *hash* antes obtenido. De esta forma, además de requerir tablas del arco iris para cada *salt* posible, las tablas han de replicar el bucle de estiramiento, lo cual como os fácil observar puede resultar en la inviabilidad del ataque.

2.4 Función atacada: CRC-32

Para este proyecto se ha escogido atacar *hashes* de contraseñas generados utilizando la función *hash* CRC-32. Las siglas CRC provienen de *cyclic redundancy check*, o lo que es lo mismo verificación por redundancia cíclica, mientras que el número 32 indica que las cadenas de salida resultantes tienen una longitud fija de 32 bits. Esta función pertenece a la familia de las funciones CRC, el uso de las cuales es bastante popular debido a su fácil implementación y rapidez de computación, entre otras cosas.

Cabe destacar que son de sobra conocidas las debilidades que presenta esta función *hash* a la hora de ocultar contraseñas frente a ataques criptográficos como el ataque del arco iris. De tal forma, se presupone que se obtendrán resultados satisfactorios al implementar el ataque frente a esta función, lo cual servirá para representar una implementación base, la cual es perfectamente escalable, por ejemplo eligiendo atacar una función *hash* diferente con mayor longitud de cadenas como la función MD5.

Md5 también es muy débil, la idea está más bien en enfatizar más si cabe a escalabilidad de los resultados del proyecto y por qué se consigue esta escalabilidad

CAPÍTULO 3

Antecesores al ataque del arco iris

Martin E. Hellman [1] introdujo por primera vez el concepto del *time-memory trade-off* (TMTO), o intercambio tiempo-memoria. Aunque esto supuso una mejora mayúscula en comparación a los ataques por fuerza bruta, él mismo admitía en su artículo que había espacio para mejorar todavía más. Poco después Rivest [2] consiguió hacer realidad las expectativas de Hellman mediante el uso de puntos distinguidos. Aunque investigando la bibliografía se pueden encontrar más artículos que suponen mejoras y nuevos métodos, estas dos aproximaciones serán consideradas como las más importantes antes de llegar al ataque del arco iris.

De dónde has sacado esto? Te has comido mucho...

3.1 Primera aproximación: *time-memory trade-off*

En julio de 1980 era publicado por Martin E. Hellman el primer artículo introduciendo el concepto del intercambio tiempo-memoria (dicho intercambio será referido a partir de ahora como TMTO). En dicho artículo, Hellman indica como el TMTO se puede aplicar con el mismo objetivo que en este proyecto: obtener contraseñas de forma no autorizada. Para este fin emplea estructuras de datos en forma de tablas, que, aunque no las llame como tal, bien podrían ser consideradas una primera aproximación de las tablas del arco iris empleadas en el ataque homónimo.

Para ello, su método emplea los siguientes parámetros:

- P : conjunto finito de posibles contraseñas.
- C : función *hash* que transforma contraseñas de P en valores *hash*.
- R : función de reconstrucción que transforma *hashes* obtenidos con C en posibles contraseñas de P .
- m : número de filas de la tabla.
- t : número de columnas de la tabla.

Teniendo en cuenta todos estos parámetros, para formar una tabla de dimensiones $m \times t$, se realizará el mismo procedimiento para cada fila i en el rango

1, ..., m . Se comenzará tomando una contraseña inicial en P , denominada p_{i_0} , de la cual será generado un valor *hash*, el cual será reconstruido aplicándole la función R . Una vez hecho esto, se habrá obtenido una nueva contraseña p_{i_1} , diferente a la anterior por requerimientos de la función de reconstrucción R . Este proceso será repetido un total de t veces en cada fila, generando por último el valor *hash* correspondiente a la última contraseña (p_{i_m}) obtenida por R , llegando a obtener al final la tabla indicada de m filas y t columnas. Si se denomina f al proceso de aplicar a una contraseña una función *hash* seguida de una función de reconstrucción, una representación visual de la generación de la tabla podría ser el siguiente:

$$\begin{array}{ccccccc}
 p_{1_0} & \xrightarrow{f} & p_{1_1} & \xrightarrow{f} & \dots & \xrightarrow{f} & p_{1_t} \xrightarrow{C} h_{1_t} \\
 p_{2_0} & \xrightarrow{f} & p_{2_1} & \xrightarrow{f} & \dots & \xrightarrow{f} & p_{2_t} \xrightarrow{C} h_{2_t} \\
 p_{3_0} & \xrightarrow{f} & p_{3_1} & \xrightarrow{f} & \dots & \xrightarrow{f} & p_{3_t} \xrightarrow{C} h_{3_t} \\
 & \vdots & & & & \vdots & \\
 p_{m_0} & \xrightarrow{f} & p_{m_1} & \xrightarrow{f} & \dots & \xrightarrow{f} & p_{m_t} \xrightarrow{C} h_{m_t}
 \end{array}$$

Figura 3.1: Computación esquematizada de las tablas de Hellman en su TMTO

Una vez finalizado el proceso, no se almacenará la tabla en su totalidad, sino que tan sólo se almacenan la primera y última entradas de cada fila, formando una tupla. De tal forma, tomando el ejemplo anterior, se almacenarán las tuplas $\{p_{1_0}, h_{1_t}\}$, $\{p_{2_0}, h_{2_t}\}$, $\{p_{3_0}, h_{3_t}\}$, ..., $\{p_{m_0}, h_{m_t}\}$, con el objetivo de ahorrar espacio en memoria.

A la hora de tratar de obtener contraseñas se realizará el siguiente procedimiento. Habiendo obtenido un *hash* almacenado en el sistema, denominado h , el primer paso será compararlo con los *hashes* en las últimas columnas de la tabla, tras lo cual se pueden dar dos posibles situaciones, dependiendo de si h colisiona con alguna de las últimas columnas:

- **h colisiona:** este caso puede darse si la contraseña correspondiente a h se encuentra en la columna anterior de la tabla, o puede resultar en una falsa alarma. En ambos casos, como se sabe en qué fila ha aparecido h , se tomará su punto de inicio y desde él se repetirá el proceso de generación de la tabla, hasta llegar a la columna anterior a h , para la cual habrá de comprobarse si permite el acceso al sistema.
 - **Acceso permitido:** efectivamente, se ha obtenido con éxito una de las contraseñas almacenadas en el sistema.

- **Acceso denegado:** falsa alarma. Esto sucede debido a que un valor *hash* puede ser obtenido aplicando la función correspondiente a diferentes contraseñas, por lo cual finalmente no se ha obtenido una contraseña válida.
- ***h* no colisiona:** en este caso, se aplicará la función de reconstrucción a *h*, seguida de la función *hash* empleada, obteniendo un nuevo *hash* *h'*, el cual vuelve a compararse con las últimas columnas de la tabla.

Puede ocurrir también que durante el proceso de generación de la tabla se generen colisiones en las últimas columnas, es decir, que dos o más de las últimas columnas acaben teniendo el mismo valor *hash*. Esto contribuye a reducir el número de filas únicas de la tabla, aunque si se consigue mantener por debajo de un porcentaje pequeño no debería de tener un impacto negativo significante en la mejora brindada por el TMTO.

Hablando del cual, el intercambio tiempo-memoria que tiene lugar aquí depende de los valores asignados tanto a *m* como a *t*. Una posible forma de pensar en dichos valores es tener en cuenta el número posible de *hashes* distintos que puede generar la función resumen atacada. Sea este número denominado *N*, una aproximación lógica sería que *m* y *t* cumplan la siguiente relación: $m \times t = N$. Dentro de esta relación, podrían darse los casos extremos de $m = N, t = 1$, o el contrario, $m = 1, t = N$, aunque dichos casos estarían implementando un ataque más parecido a la fuerza bruta en lugar de hacer uso del TMTO. En la época en la que Hellman ideó esta aproximación, en lugar de emplear tan sólo una tabla con dimensiones $m \times t = N$ su método hacía uso de diversas tablas de menor dimensionalidad para reducir las limitaciones establecidas por la memoria del sistema.

En el intercambio en este caso, un mayor número de filas respecto al número de columnas supondrá un menor tiempo de computación para la generación de la tabla, pero requerirá más espacio en memoria para almacenar los resultados, ya que se obtendrán más tuplas con la primera y última entrada de cada fila. En cambio, aumentar el valor de *t* mientras se reduce el de *m* resultará en una reducción en el espacio requerido en memoria, aunque el tiempo de computación necesario para obtener la tabla será mayor, ya que para cada fila se repetirá en procedimiento visto en la Figura 3.1 más veces.

Un caso práctico... ¿cuál sería la necesidad de espacio considerando la función del resumen de X bits?

3.2 Puntos distinguidos

Tomando el método de Hellman [1] mediante el uso del TMTO, Rivest [2] implementó una mejora que reducía el tiempo de búsqueda a la hora de tratar de obtener contraseñas. Dicha mejora está basada en la implementación de puntos distinguidos en la tabla.

Los puntos distinguidos de Rivest se dan al establecer ciertas condiciones para las últimas entradas de la tabla, es decir, los elementos contenidos en las últimas columnas. Dichas condiciones quedan a la elección del atacante, pudiendo ser empezar o terminar con un número determinado de ceros o unos, por ejemplo.

Esta aproximación tiene un matiz a la hora de generar la tabla, y es que como las últimas entradas deben cumplir las condiciones establecidas, ahora en lugar de obtener dichas entradas después de t pasos, se habrá de continuar hasta generar una entrada que satisfaga dichas condiciones. Por otra parte, la gran ventaja que brinda este método es una reducción significativa en cuanto al tiempo de búsqueda en las tablas, ya que el número de posibles colisiones de últimas entradas de la tabla se ve notablemente reducido, lo cual resulta también en un mayor rango de contraseñas cubierto por la tabla, aumentando la probabilidad de éxito de la misma.

CAPÍTULO 4

El ataque del arco iris

Lo que describes no es más que la aproximación de Hellman... aplicado a un caso particular

En 2003, Philippe Oechslin [3] publicó en primer artículo en el cual aparecía el concepto de la tabla del arco iris. Dicha tabla consiste en una implementación más compleja del intercambio tiempo-memoria de Hellman, el cual se ha visto en la sección 3.1. Parte de la mayor complejidad se debe al aumento del tamaño de la tabla, ya que las limitaciones en cuanto a la memoria para almacenar las tablas que se daban lugar cuando Hellman publicó su artículo no ocurren en este caso.

Es momento de detallar la implementación llevada a cabo del ataque del arco iris. Antes de nada, se han de especificar el dominio de contraseñas y la función *hash* empleados. Ya se ha visto anteriormente en qué consiste una función *hash*, por lo cual no es requerido volver a explicarlo.

Un dominio de contraseñas se puede describir como todo el rango de posibles contraseñas, las cuales han de cumplir ciertas condiciones, como longitud de la contraseña o caracteres permitidos. En el caso de este proyecto se ha escogido:

Consideraciones de esta experimentación: formato passwd, función hash, tamaños de tabla...

- **Dominio:** tendrán una longitud de seis caracteres, los cuales podrán ser tan sólo números. Esto conforma un dominio con un total de un millón de contraseñas, las cuales se encuentran en el rango '000000', ...,'999999', conformando un total de un millón de contraseñas.
- **Función hash:** la función CRC-32, la cual ya ha sido explicada anteriormente en la sección 2.4.

El motivo que ha llevado a elegir tal dominio y función *hash* no es otro buscar una reducción en los tiempos de generación de tablas del arco iris y búsqueda en las propias tablas.

Una ventaja de el ataque del arco iris es la escalabilidad que posee, ya que la forma en que se desarrolla el ataque va a ser la misma sea contra la función CRC-32 o contra cualquier otra que genere *hashes* de longitud mayor, como por ejemplo la función *hash* MD5. Lo mismo se puede aplicar a el dominio de contraseñas: aunque más posibles contraseñas existan en el dominio (letras minúsculas y/o mayúsculas, símbolos, ...), siempre serán tratadas de la misma forma, tan sólo resultando en un aumento del tiempo requerido para romperlas. La memoria necesaria para almacenar las tablas también aumentará tanto empleando funciones resumen con mayores valores *hash* como aplicando el ataque a dominios de

contraseñas mayores, ya que las entradas de las tablas del arco iris contendrán valores de mayor tamaño.

4.1 Generación de tablas del arco iris

Como se ha visto en el apartado anterior, el dominio de contraseñas a ser atacadas consta de un total de un millón de contraseñas, el cual va a pasar a ser representado por el parámetro N . Como también sucediera anteriormente en la sección 3.1 al introducir el intercambio tiempo-memoria (TMTD), en la generación de tablas al implementar el ataque del arco iris se ha de tener en cuenta el tamaño de dichas tablas, el cual depende de el número total de contraseñas. Recuérdese que el tamaño de una tabla en el TMTD del arco iris viene dado por el número de filas (m) y de columnas (t) que posee. Una de las mayores diferencias de este ataque respecto al TMTD de Hellman es que así como en el TMTD se hacía uso de diversas tablas de menor dimensión, en el ataque del arco iris basta con generar una sola tabla de mayor dimensionalidad. En este caso, el tamaño de la tabla debería de cumplir:

$$m \times t \approx 10^6 = N \quad (\text{Fórmula 1})$$

Teniendo esto en cuenta, se procederá a generar diferentes tablas de tamaños diversos, la mayoría cumpliendo con la relación establecida en (Fórmula 1), aunque algunas tablas tendrán tamaños por encima o por debajo de N , para explorar más posibilidades y observar la eficiencia de las tablas con diferentes tamaños. Un total de 74 tamaños de tabla diferentes serán empleados, siendo los posibles valores para m y t los siguientes:

- $m \in \{250, 500, 1000, 2000, 2500, 5000, 10000, 20000, 25000, 50000, 100000, 200000, 250000, 500000, 1000000\}$
- $t \in \{1, 2, 4, 5, 10, 20, 40, 50, 100, 200, 400, 500, 1000\}$

Queda todavía por especificar la función de reconstrucción a emplear. En la implementación aquí desarrollada existirán dos funciones de reconstrucción diferentes las cuales serán denominadas **R1** y **R2**. El hecho de poder emplear más de una función de reconstrucción diferente en las tablas del arco iris constituye una de las diferencias fundamentales respecto al TMTD de Hellman, ya que las tablas empleadas en aquel momento tan sólo hacían uso de una función de reconstrucción.

Ambas funciones de reconstrucción (**R1** y **R2**) tienen el mismo comportamiento en cuanto a la entrada que reciben y la salida que producen: tomar como entrada un valor *hash* generado por la función CRC-32, generando como salida una posible contraseña en el rango '000000', ..., '999999'. Cada una de estas funciones actúa de la siguiente forma, tomando un valor *hash* h y produciendo una reconstrucción r :

- **R1:** tomando h en forma numérica, esta función le aplicará la operación módulo un millón, con el objetivo de obtener los seis últimos números de h .
- **R2:** tomando h en forma hexadecimal, esta función inicialmente eliminará los dos primeros caracteres de h , los cuales siempre serán '0x'. Seguidamente, mientras el *hash* resultante tenga una longitud mayor a seis caracteres, se irá eliminando el primero de estos caracteres, hasta obtener una longitud igual a seis. Por último, todas las letras que queden en h serán sustituidas por números. Estos números no serán seleccionados al azar, sino que corresponderán con los números de h en forma numérica, comenzando desde el final, y tras cada sustitución se tomará el número anterior.

Para enseñar de forma más clara cómo funcionan tanto **R1** como **R2**, se puede tomar como ejemplo la contraseña '112233', la cual será denominada p . En este caso, los valores *hash* de p obtenidos tras aplicarle la función CRC-32 serán, en forma numérica (h_{num}) y en forma hexadecimal (h_{hex}): $h_{num} = '3570655599'$ y $h_{hex} = '0xd4d3e16f'$.

Aplicando **R1**, tomando p , obteniendo $h_{num} = '3570655599'$ y aplicándole la operación módulo un millón, el valor reconstruido resultante será $r = '655599'$, el cual corresponde con una posible contraseña dentro del dominio establecido.

La aplicación de **R2** sería llevada a cabo de la siguiente forma:

1. Tomar p y obtener $h_{num} = '3570655599'$ y $h_{hex} = '0xd4d3e16f'$.
2. Eliminar los dos primeros caracteres de h_{hex} , resultando en $h_{hex} = 'd4d3e16f'$.
3. Mientras la longitud de h_{hex} sea mayor que seis, eliminar su primer carácter, resultando finalmente en $h_{hex} = 'd3e16f'$
4. Recorrer los caracteres de h_{hex} desde el principio, y aquellos que sean letras se reemplazan por números de h_{num} comenzando por el final. Tras este paso se tendrá $r = '939165'$, ya que se habrán sustituido las letras 'd', 'e' y 'f' por los números 9, 9 y 5, los cuales se encuentran en la última, penúltima y antepenúltima posición de h_{num} , respectivamente.

Una vez se ha especificado el dominio de contraseñas, las función *hash* y la función de reconstrucción a emplear, es momento de establecer qué tamaños de tablas serán elegidos y generar dichas tablas. Para ello, como se ha visto en la Figura 3.1, una vez establecidos los parámetros m (filas) y t (columnas), los pasos a seguir en este caso serán:

1. Tomar una contraseña inicial p

2. Obtener h , el valor *hash* de p
 - 2.1. Reconstruir h , obteniendo r
 - 2.2. Generar h de nuevo, siendo el valor *hash* de r
 - 2.3. Repetir t veces
3. Guardar p y el último h obtenido en la tabla
4. Repetir m veces

La tabla del arco iris resultante será almacenada en memoria, aunque con unas dimensiones reducidas respecto a las empleadas en su generación. Mientras que el número de filas de cada tabla seguirá siendo m , el número de columnas se ve reducido, guardando tan sólo dos columnas por cada fila, las cuales corresponderán con el primer y último elementos obtenidos durante la construcción de la tabla. No es necesario almacenar la tabla en su totalidad, ya que al emplear funciones de reconstrucción deterministas -es decir, que siempre generan el mismo resultado para cada entrada específica- únicamente se requiere saber desde qué contraseña inicial se ha partido para volver a generar toda la fila siguiendo el proceso descrito en los pasos 1, 2, 3 y 4 arriba vistos. La contraseña inicial de la primera fila siempre será $p_0 = '000000'$, en la segunda fila se comenzará por $p_1 = '000001'$, y así sucesivamente hasta llegar al final de la tabla.

Recuérdese que se está tratando de romper contraseñas almacenadas en forma de valores *hash*. La manera de conseguirlo será si, durante el proceso de generación de una tabla, se llega a obtener dicho *hash* que se está tratando de romper, ya que eso significará que la entrada anterior en la tabla será su contraseña correspondiente. A la hora de buscar contraseñas en una tabla, antes que nada se buscará en las entradas guardadas en memoria, ya que si el valor *hash* se encuentra en una de las últimas entradas de la tabla no hay necesidad de volver a generarla, debido a que se puede concluir que es posible obtener su contraseña correspondiente. En caso de no encontrar el valor *hash* en una de las últimas entradas de la tabla, se realizará el proceso de búsqueda volviendo a generar la tabla, ya que cabe la posibilidad de que el valor *hash* a romper se encuentre en una de las entradas entre la primera y la última, las cuales no han sido almacenadas en memoria. Si en algún momento durante esta búsqueda se da con el valor *hash* a romper, se concluye la búsqueda con éxito. El caso opuesto significará que la tabla no ha sido capaz de romper esa contraseña.

En cuanto a el tamaño de la tabla, puede observarse que cuantas más filas posea, mayor probabilidad existirá de romper una contraseña sin necesidad de generar la tabla de nuevo, mientras que cuantas más columnas tenga una tabla, generará un mayor número de posibles contraseñas en la búsqueda, por lo cual aumentará la probabilidad de romper una contraseña con éxito. Viendo estas propiedades, es muy tentador generar tablas del máximo tamaño posible, ya que en principio parece que parten con ventaja respecto a tablas más pequeñas. Bien, no todo son ventajas con un mayor tamaño, ya que si bien la probabilidad de éxito aumenta, también lo hace en principio el porcentaje de colisiones en una misma tabla. Una colisión se da cuando para filas diferentes de una tabla, el valor *hash* generado en la última entrada es el mismo, resultando en una reducción del número

No pueden considerarse tablas muy grandes si la función hash tiene un tamaño medio, no hablemos de si tiene un tamaño grande

de contraseñas únicas cubiertas por la tabla. También en cuanto al tamaño, si bien en este caso es factible construir tablas que contengan todas las contraseñas, ya que son en total $N = 10^6$, en el momento en que se esté tratando con un dominio de contraseñas de mayor tamaño esta posibilidad queda eliminada, ya que sería inviable obtener dichas tablas debido al tiempo requerido en su construcción y/o la memoria requerida para almacenarlas. En este caso las tablas con un millón de entradas no tardan más de cinco minutos en ser generadas y ocupan un total de 22.0 MB en memoria, siendo perfectamente viables.

En cuanto a las funciones de reconstrucción, se experimentará con diferentes combinaciones de las mismas, con el objetivo de determinar cuál de ellas brinda los mejores resultados. Cada combinación de dichas funciones de reconstrucción corresponderá a una configuración de tabla diferente, pudiendo ser una de las siguientes configuraciones:

- **Módulo un millón (R1):** se empleará la función R1 para reconstruir los valores *hash* para todas las entradas de la tabla.
- **Hexadecimal (R2):** se empleará la función R2 para reconstruir los valores *hash* para todas las entradas de la tabla.
- **Módulo un millón y hexadecimal (R1 + R2):** se empleará R1 para las entradas pares (comenzando por cero) y R2 será utilizada en las entradas impares.
- **Patrón pequeño (R1 + R2 + R2):** se empleará R1 en la primera reconstrucción de cada fila, mientras que para las siguientes dos reconstrucciones se hará uso de R2. Acto seguido se volverá a usar R1, repitiendo este patrón hasta alcanzar el final la fila correspondiente.
- **Patrón grande (R1 + R2 + R2 + R1 + R1 + R2 + R2 + R2):** al igual que en el anterior patrón, se comenzará empleado R1 para la primera entrada, continuando con la combinación establecida para las siguientes siete entradas, hasta alcanzar el final de la fila correspondiente.

El objetivo de generar tablas con diferentes combinaciones de las funciones de reconstrucción consiste en tratar de minimizar el número de colisiones. En principio se espera que las tablas que emplean una sola función de reconstrucción tengan un mayor número de colisiones respecto a aquellas tablas que combinen las funciones de reconstrucción. La forma de medir las colisiones que posea una tabla es sencilla, ya que es suficiente recorrer sus últimas columnas y determinar cuántas de ellas son únicas, es decir, cuántas aparecen tan sólo una vez. Una vez obtenido este valor, llámese c , sería suficiente con restarlo al número total de filas de la tabla, m , y el resultado será el número de colisiones de la tabla.

Para cada una de las configuraciones de tabla establecidas se generarán tablas de los 74 tamaños diferentes, obteniendo un total de $74 \cdot 5 = 370$ tablas en total, lo cual deberían de ser suficientes tablas para establecer cuál es la mejor combinación de tamaño de tabla y funciones de reconstrucción.

En cuanto han sido determinados los parámetros que configurarán las tablas del arco iris, queda a elección del atacante escoger la estructura de datos en par-

ticular que será la empleada para representarla. A la hora de optar por una estructura de datos u otra, se ha de tener en cuenta que no se va a almacenar en memoria la tabla en su totalidad, es decir, todas las $m \times t$ entradas, sino que únicamente se almacenarán la primera y última entradas de cada fila de la tabla, o lo que es lo mismo, la primera y última columnas de cada fila. Dentro de todas las posibilidades lo ideal sería, una vez que se haya desarrollado el método de construcción de tablas, generar aquellas que se desee mediante diferentes estructuras de datos, para una vez obtenidas todas ellas comparar los tiempos necesarios en su construcción y en el acceso a las mismas, así como el espacio que ocupan en memoria. En el caso de la implementación aquí desarrollada, se compararon tablas del arco iris representadas mediante listas de tuplas y diccionarios. Cada una de estas estructuras de datos representa las tablas de la siguiente forma:

- **Lista de tuplas:** Cada entrada de la lista corresponde a una fila de la tabla y contiene una tupla o pareja de elementos. El primero de estos elementos es la contraseña inicial de la fila correspondiente, mientras que el segundo elemento corresponde al valor *hash* obtenido en la última entrada de la fila correspondiente. De esta manera, empleando la Figura 3.1 como ejemplo, la lista de tuplas que equivaldría a esta tabla del arco iris correspondería con la siguiente representación: $[(p_{1_0}, h_{1_t}), (p_{2_0}, h_{2_t}), (p_{3_0}, h_{3_t}), \dots, (p_{m_0}, h_{m_t})]$.

No dejan de ser detalles de implementación, cada uno escoger los que más rabia le dan

- **Diccionario:** esta estructura de datos viene como anillo al dedo a la hora de representar tablas del arco iris, debido a su estructura interna siendo cada entrada compuesta de una clave y su valor correspondiente, tal que *clave* : *valor*. De tal forma, en cada entrada del diccionario correspondiente a una fila de la tabla, la *clave* será igual a la primera contraseña de dicha fila, mientras que el *valor* corresponderá con el resumen *hash* generado en la última columna de dicha fila. De nuevo, si se toma como ejemplo la Figura 3.1, el diccionario resultante que representaría a la tabla del arco iris correspondiente sería el siguiente: $\{p_{1_0} : h_{1_t}, p_{2_0} : h_{2_t}, p_{3_0} : h_{3_t}, \dots, p_{m_0} : h_{m_t}\}$.

Tras construir las primeras tablas del arco iris de diversos tamaños, habiendo generado dos copias de cada una, siendo una copia representada por una lista de tuplas y la otra por un diccionario, se experimentó con ellas midiendo los tiempos empleados en la generación y búsqueda, así como la memoria ocupada. Tras realizar dicha experimentación, se optó por elegir el diccionario como estructura de datos para todas las futuras tablas. Esto fue debido a que, si bien los tiempos requeridos por cada una de las estructuras de datos resultaron ser extremadamente similares, las tablas representadas por un diccionario ocupaban cerca de la mitad del espacio en memoria que aquellas de igual tamaño pero guardadas en forma de lista de tuplas.

Una vez establecido todo lo necesario para construir las tablas del arco iris, se puede proceder a generar tablas de tamaños determinados y dejarlas preparadas para atacar las contraseñas almacenadas en el sistema.

4.2 Generación de contraseñas aleatorias

Para simular las contraseñas almacenadas en un sistema operativo en forma de valores *hash*, se generarán un total de 1.000 *hashes* aleatorios. Para su obtención se escogerá de forma aleatoria 1.000 contraseñas en el dominio establecido (N), generando el valor *hash* correspondiente de cada una y almacenándolo en una lista. Dicha lista será almacenada en memoria al finalizar el proceso de obtención del los 1.000 valores *hash*. Las contraseñas correspondientes a estos 1.000 *hashes* serán las que todas las tablas tratarán de romper, es decir, obtener la contraseña original únicamente conociendo el valor *hash* correspondiente.

4.3 Primeras pruebas

Una vez generadas todas las tablas del arco iris y las contraseñas a atacar, es el momento de ponerse manos a la obra. Inicialmente no se experimentó con los 74 tamaños de tabla, sino con 25 de ellos, los cuales se apreciarán a continuación. Dichos tamaños iniciales formaron parte de la primera implementación del ataque del arco iris sobre las contraseñas anteriormente generadas, tras la cual se ajustaron las dimensiones de las tablas empleadas. A cada tabla empleada en el ataque del arco iris a las contraseñas se le asigna un porcentaje de éxito. Dicho porcentaje de éxito para cada tabla se mide dividiendo el número de contraseñas que ha sido capaz de romper entre el número total de contraseñas a romper, 1.000 en este caso. Para todos los resultados se empleará el mismo código de color:

- **rojo**: porcentaje de éxito en el rango de 0, . . . , 69'99 %.
- **amarillo**: porcentaje de éxito en el rango de 70, . . . , 84'99 %.
- **azul**: porcentaje de éxito en el rango de 85, . . . , 94'99 %.
- **verde**: porcentaje de éxito en el rango de 95, . . . , 100 %.

Idealmente se desea que las tablas del arco iris generadas sean capaces de romper todas las contraseñas almacenadas en el sistema. Si bien esto es posible en este caso para el dominio de contraseñas elegido, si se expandiese dicho dominio resultaría más complicado, por lo cual si una tabla consigue romper al menos 950 contraseñas de las 1.000 generadas aleatoriamente será clasificada como idónea. Así mismo, una tabla que no llegue a romper tantas contraseñas pero consiga al menos adivinar 850 será clasificada como aceptable, ya que significará que en la mayoría de los casos será capaz de acceder al sistema. Las tablas que no consigan llegar a ese nivel no serían válidas para ser empleadas en un ataque, aunque si al menos consiguen romper 700 contraseñas puede considerarse que se han quedado cerca de ser aceptables, mientras que una tabla que ni siquiera consiga romper 700 contraseñas debería de ser descartada de inmediato, ya que tendría una probabilidad de adivinar una contraseña similar a sacar cara o cruz tirando una moneda al aire.

Los resultados de la primera prueba, para cada combinación de funciones de reconstrucción, pasan a verse a continuación.

Comenzando por aquellas tablas que emplean la función de reconstrucción con la operación módulo un millón (**R1**):

	1	2	4	5	10	20	40	50	100	200	400	500	1000	1000000
<i>t</i>														100.00%
														91.90%
1														
2														
4														
5														
10														
20														
40														
50														
100														
200														
400														
500														
1000	63.50%													
	1000	2000	2500	5000	10000	20000	25000	50000	100000	200000	250000	500000	1000000	<i>m</i>

Figura 4.1: Porcentajes de éxito para las tablas empleando R1

Puede observarse que en este caso se obtienen resultados bastante satisfactorios. Tan sólo una de las tablas sería descartada, aquella con dimensiones 1000×1000 , mientras que después de esa solamente hay otra que no llega a ser aceptable, aunque se queda realmente cerca de conseguirlo. Todas las tablas con 2500 filas o más serán consideradas como aceptables o idóneas, llegando incluso a obtener un éxito del 100 % en varios casos, si bien dichas tablas tienen unas dimensiones que exceden el dominio elegido. Por ejemplo, la primera tabla que consigue un porcentaje de éxito perfecto tiene unas dimensiones de $250.000 \times 100 = 25.000.000 > 1.000.000 = N$. Es interesante también observar como para las tablas con mayor número de filas, si se intenta mantener unas dimensiones dentro del dominio N resultan peores que tablas con menos filas pero más profundidad. Por ejemplo, la tabla con dimensiones 200.000×5 tiene menor porcentaje de éxito que la tabla con dimensiones 20.000×50 , teniendo un 10 % de las filas, pero 10 veces la profundidad, lo cual resulta clave para su mejor resultado.

Pasando ahora a las tablas que emplean la función de reconstrucción empleando el *hash* en hexadecimal (**R2**) se obtienen los siguientes resultados:

	1													100.00%
	2													91.10%
	4													89.40%
	5													88.80%
	10													93.00%
<i>t</i>	20													95.50%
	40													97.10%
	50													97.10%
	100													98.90% 99.70%
	200													100.00% 100.00% 100.00%
	400													99.90% 99.90% 100.00%
	500													74.90%
	1000													57.10%
	1000	2000	2500	5000	10000	20000	25000	50000	100000	200000	250000	500000	1000000	<i>m</i>

Figura 4.2: Porcentajes de éxito para las tablas empleando R2

Observando en un primer vistazo la distribución de los colores a lo largo de la tabla se puede apreciar que los resultados aquí obtenidos son peores que los anteriores con **R1**. La tabla que ya se esperaba que fuera a ser descartada ($m = 1.000$) rinde peor que antes, y se han generado menos tablas idóneas que en los resultados anteriores. De nuevo, las tablas con un número de entradas muy elevado siguen alcanzando el 100 % de porcentaje de éxito. Con todo esto siguen siendo resultados satisfactorios, ya que hay múltiples tablas con un porcentaje de éxito mayor al 95 %.

En cuanto a las tablas que entrelazan ambas funciones de reconstrucción (**R1 + R2**) se obtiene:

La idea es más bien establecer una línea base, detectar donde es susceptible la mejora, y intentar modificar el proceso de construcción de las tablas para tener mejores resultados con menos tamaño

	1													100.00%
	2													97.00%
	4													96.20%
	5													97.00%
	10													98.30%
<i>t</i>	20													99.50%
	40													99.70%
	50													99.60% 100.00% 100.00%
	100													100.00% 100.00% 100.00%
	200													99.90%
	400													98.10%
	500													95.40%
	1000													63.10%
	1000	2000	2500	5000	10000	20000	25000	50000	100000	200000	250000	500000	1000000	<i>m</i>

Figura 4.3: Porcentajes de éxitos para las tablas empleando R1 + R2

Estos son los mejores resultados obtenidos hasta el momento debido a que, aunque todavía existe la misma tabla descartable con 1.000 entradas, el resto de tablas del arco iris generadas son idóneas, habiendo obtenido incluso un mayor número de tablas que han conseguido romper todas las contraseñas. Entrando

en detalla en las tablas perfecta, además de las tablas más grandes de las cuales ya se esperaban estos resultados, se puede observar que la tabla con dimensiones 50.000×50 es la de menor tamaño entre ellas, lo cual significa que sería la que menos memoria requiriese para ser almacenada. De esta tabla se podría decir también que sería, dentro de aquellas con un éxito del 100 %, una de las que menos tiempo requeriría en su generación, ya que si bien posee un número elevado de columnas, sus pocas filas en comparación a las demás reducirán el tiempo para construirla.

La mejora de estos resultados respecto a los anteriores es debido a que al combinar ambas funciones de reconstrucción, el número de colisiones de las tablas se reduce de manera significativa. Como consecuencia, las tablas son capaces de cubrir un mayor rango de contraseñas, obteniendo mejores resultados como ya se ha visto.

Para las tablas que emplean el denominado patrón grande ($R1 + R2 + R2 + R1 + R1 + R2 + R2 + R2$) se obtienen los siguientes resultados:

t													100.00%	
	1	2	4	5	10	20	40	50	100	200	400	500		
1000	60.40%													
1000	60.40%	1000	2000	2500	5000	10000	20000	25000	50000	100000	200000	250000	500000	1000000
														m

Figura 4.4: Porcentajes de éxito para las tablas empleando el patrón grande

Estos resultados son peores que los obtenidos anteriormente ($R1 + R2$). Empleando el patrón establecido se esperaba en principio una reducción en el número de colisiones todavía mayor que en cualquiera de las demás combinaciones de funciones de reconstrucción, esperando como consecuencia una mejora en los resultados. Si bien el número de colisiones se ha visto reducido, aunque no en la medida esperada, es aparente que los resultados no suponen una mejora respecto a las tablas de la Figura 4.3, aunque sí que rinden mejor que las tablas que emplean únicamente una de las funciones de reconstrucción. En este caso también se obtienen tablas capaces de romper todas las contraseñas, las cuales ya vienen siendo habituales para los tamaños correspondientes, aunque se pueden observar un mayor número de tablas aceptables pero no idóneas.

Por último, en cuanto a las tablas que emplean el denominado patrón pequeño, los resultados obtenidos corresponden con los siguientes:

	1												100.00%
	2												83.70%
	4												82.60%
	5												86.00%
	10												88.80%
	20												92.90%
t	40												96.20%
	50												97.80%
	100												97.80% 99.40% 99.80%
	200												99.90% 100.00%
	400												100.00% 100.00% 100.00%
	500												100.00% 100.00% 100.00%
	1000												71.30%
	1000	2000	2500	5000	10000	20000	25000	50000	100000	200000	250000	500000	1000000
													m

Figura 4.5: Porcentajes de éxito para las tablas empleando el patrón pequeño

Por primera vez no se han generado tablas que vayan a ser descartadas. Incluso la tabla que habitualmente tiene el peor rendimiento ($m = 1.000$) en esta ocasión ha quedado cerca de ser considerada aceptable. De nuevo, aunque las colisiones para cada tabla se han visto reducidas en un pequeño porcentaje, no ha sido suficiente para mejorar los resultados obtenido entrelazando las funciones de reconstrucción (**R1 + R2**). El esquema general vuelve a ocurrir aquí también, con las tablas esperadas teniendo un porcentaje de éxito del 100 %.

Tras esta primera tanda de pruebas, es fácilmente observable que, para tablas que superen las 10.000 filas, el porcentaje de éxito no baja del 88.0 %. Dichas tablas serán capaces de romper la gran mayoría de las contraseñas almacenadas en el sistema. Si bien esta es la intención del ataque del arco iris, resultaría interesante indagar en tablas de menor tamaño, para así poder determinar si realmente es necesario crear tablas que contengan tantas filas, o si por el contrario existen ciertas configuraciones que son capaces de brindar resultados igual de buenos que las tablas más grandes. Encontrar dichas configuraciones es de gran interés, ya que de ser posible conseguirlo significaría una reducción en los requerimientos tanto de tiempo como de memoria, algo que siempre es beneficioso en este tipo de ataques.

4.4 En busca de la configuración perfecta

Si bien en las pruebas anteriores la tabla con un menor número de filas posiblemente sea $m = 1.000$ filas, en la búsqueda de las tablas con resultados similares a los mejores obtenidos anteriormente pero con menor tamaño será necesario emplear tamaños de tabla diferentes, expandiendo la búsqueda con tamaños menores a los vistos anteriormente. De tal forma, la tabla con menos filas pasará a tener $m = 250$, mientras que la tabla más grande tendrá $m = 10,000$ filas. En concreto, se generarán tantas tablas como posibles permutaciones existan de los siguientes conjuntos de m y t :

- $m \in \{250, 500, 750, 1.000, 1.500, 2.000, 2.500, 5.000, 10.000\}$
- $t \in \{50, 100, 200, 400, 500, 1.000\}$

En total se generarán $6 \cdot 9 = 54$ tablas para cada combinación de funciones de reconstrucción. Tras generar todas estas tablas, pero antes de obtener su rendimiento, resulta interesante observar las colisiones dichas tablas en las diferentes combinaciones de funciones de reconstrucción. Para ello se obtendrá el porcentaje de colisiones de cada tabla, obteniendo primero el número de colisiones como ya se ha explicado anteriormente, y acto seguido dividiéndolo entre el número de filas de la tabla (m). Para dichos porcentajes, al igual que se ha hecho en la sección anterior con los porcentajes de éxito, se empleará un código de color, el cual será el siguiente:

- **rojo**: porcentaje de colisiones en el rango de 70, . . . , 100 %.
- **amarillo**: porcentaje de colisiones en el rango de 50, . . . , 69'99 %.
- **azul**: porcentaje de colisiones en el rango de 25, . . . , 49'99 %.
- **verde**: porcentaje de colisiones en el rango de 0, . . . , 24'99 %.

Interpretando este código de color podría decirse que una tabla que tengan colisiones en un 70 por ciento de sus filas o más no serían las más ideales para ser empleadas en la búsqueda de contraseñas, ya que tantas colisiones se traducen en una reducción muy significante de las contraseñas cubiertas por esa tabla. Si este porcentaje se reduce un poco, mientras una tabla cubra cerca de la mitad de sus contraseñas, serán sus dimensiones las que determinen su utilidad, ya que si la tabla es lo suficientemente grande será capaz de compensar la pérdida de contraseñas cubiertas con su tamaño. Cualquier tabla que cubra más de la mitad de sus contraseñas será aceptable en este aspecto, aunque de nuevo sus dimensiones serán cruciales para determinar su efectividad atacando contraseñas. Por último, todas aquellas tablas que cubran tres cuartas partes o más de las contraseñas que se pretenden serán en principio las más efectivas.

Comenzando al igual que antes por las tablas que tan sólo emplean la reconstrucción con la operación módulo un millón (**R1**), se obtiene:

	50	0.80%	1.80%	2.40%	2.60%	4.27%	5.15%	5.80%	10.16%	19.08%
	100	2.00%	2.60%	4.00%	5.00%	7.47%	9.55%	10.76%	18.96%	32.57%
	200	3.60%	4.60%	6.80%	8.90%	13.00%	16.00%	17.92%	30.06%	47.18%
	400	5.20%	8.40%	13.07%	16.50%	23.47%	28.55%	31.96%	48.74%	65.83%
	500	6.00%	10.00%	15.20%	19.40%	27.00%	32.55%	36.68%	54.74%	71.14%
	1000	9.20%	13.60%	18.93%	24.00%	33.20%	40.35%	45.68%	65.02%	79.64%
	250	500	750	1000	1500	2000	2500	5000	10000	
							m			

Figura 4.6: Porcentajes de colisiones para las tablas empleando R1

Como era de esperar y va a ser habitual en estos resultados, las tablas con menos filas tendrán porcentajes de colisiones menores respecto a las tablas de mayor tamaño. Las únicas tablas con un porcentaje de colisiones muy restrictivo ocurren con unas dimensiones superiores al dominio N , lo cual no es sorprendente en tales casos. Se puede concluir que el uso exclusivo de **R1**, en cuanto a colisiones, parece prometer resultados satisfactorios a la hora de atacar contraseñas.

Para las tablas que emplean únicamente la función de reconstrucción utilizando el *hash* en hexadecimal (**R2**) se obtienen los siguientes resultados:

	50	0.00%	1.60%	2.00%	2.60%	4.20%	5.70%	6.64%	12.52%	22.60%
	100	0.40%	2.60%	4.27%	5.50%	8.13%	10.15%	12.32%	22.50%	36.42%
	200	2.80%	7.20%	9.47%	12.40%	16.27%	18.80%	21.60%	34.72%	51.27%
	400	11.60%	18.00%	21.73%	27.10%	34.13%	38.75%	43.28%	58.74%	73.00%
	500	15.60%	23.60%	27.60%	33.40%	41.07%	46.50%	51.68%	67.52%	79.88%
	1000	18.00%	28.80%	35.87%	42.80%	53.67%	60.95%	66.76%	81.62%	90.47%
	250	500	750	1000	1500	2000	2500	5000	10000	
							m			

Figura 4.7: Porcentajes de colisiones para las tablas empleando R2

En esta ocasión, aunque para las tablas de menor tamaño se obtiene un menor número de colisiones, en cuanto aumentan las dimensiones de las tablas los resultados empeoran respecto a los anteriores. Esta es la razón por la cual, como se ha visto en las primeras pruebas, estas tablas tienen un peor rendimiento que las que emplean únicamente **R1**, aunque aún así seguían siendo buenos resultados. En este caso se han generado más tablas con peores porcentajes de colisiones, la más grande de ellas superando el 90 por ciento de colisiones, lo que se traduce en menos de 1.000 contraseñas cubiertas por una tabla que en principio se esperaba que cubriese 10.000. También hay más tablas por encima del cincuenta por ciento, cubriendo algo menos de la mitad de las contraseñas previstas, lo cual considerando su tamaño puede llegar a comprometer su rendimiento de manera significativa.

Pasando ahora a las tablas que combinan ambas funciones de reconstrucción se obtiene:

<i>t</i>	50	0.80%	1.00%	1.73%	2.30%	3.93%	5.30%	6.32%	11.48%	20.74%
	100	0.80%	1.60%	3.07%	4.30%	6.80%	8.80%	10.96%	19.48%	33.78%
	200	2.00%	4.80%	6.67%	8.30%	12.60%	16.65%	20.00%	32.16%	49.08%
	400	2.40%	9.00%	12.80%	17.80%	23.00%	29.05%	33.24%	47.44%	63.71%
	500	3.20%	11.00%	16.13%	21.80%	27.33%	33.60%	38.24%	52.96%	68.54%
	1000	9.60%	21.40%	29.20%	36.90%	46.00%	53.40%	59.24%	74.32%	85.00%
	250	500	750	1000	1500	2000	2500	5000	10000	
							<i>m</i>			

Figura 4.8: Porcentajes de colisiones para las tablas empleando R1 + R2

Aunque se ha mejorado respecto a las tablas que emplean únicamente **R2**, en comparación a las primeras tablas que hacen uso tan sólo de **R1** existen más colisiones en estas tablas, aunque en general estos resultados no son mucho peores. Este pequeño empeoramiento resulta sorprendente, ya que se esperaba que al combinar ambas funciones de reconstrucción el porcentaje de colisiones fuera reducido drásticamente. En línea con lo anteriormente visto siguen tanto la distribución de colores a lo largo de la tabla como los altos porcentajes de las tablas de mayor tamaño. Aún con todo, estos resultados se podrían clasificar como satisfactorios.

En cuanto a las tablas que emplean el denominado patrón grande (**R1 + R2 + R2 + R1 + R1 + R2 + R2 + R2**) se observan los siguientes resultados:

<i>t</i>	50	0.00%	0.40%	0.67%	0.80%	1.27%	1.70%	2.04%	3.18%	6.70%
	100	1.20%	1.20%	1.60%	2.30%	3.93%	4.80%	5.60%	11.20%	20.63%
	200	2.40%	4.60%	7.87%	9.50%	14.67%	17.55%	20.28%	33.68%	50.76%
	400	4.00%	8.00%	12.80%	16.50%	23.53%	28.25%	32.44%	48.94%	65.97%
	500	3.60%	4.60%	7.87%	10.20%	15.73%	18.80%	22.12%	36.64%	54.28%
	1000	10.40%	21.80%	30.80%	36.80%	45.33%	51.30%	55.88%	70.64%	82.47%
	250	500	750	1000	1500	2000	2500	5000	10000	
							<i>m</i>			

Figura 4.9: Porcentajes de colisiones para las tablas empleando el patrón grande

Puede observarse como las tablas con esta configuración poseen menor número de colisiones que las anteriores en la Figura 4.8 (**R1 + R2**), lo cual era de esperar, ya que el objetivo de establecer este patrón era forzar una reducción en el número de colisiones. Lo sorprendente en este caso es que siguen habiendo porcentajes de colisiones mayores que en la Figura 4.6 (**R1**). Esto resulta totalmente inesperado, ya que la función de reconstrucción **R1** parece en principio la más simple, dando a entender que sería la que más colisiones produciría, empeorando los resultados. Una cosa queda clara, y es que el objetivo inicial de emplear este patrón no se ha cumplido, aunque resta por ver la efectividad de estas tablas a la hora de tratar de romper contraseñas.

Por último, las tablas que emplean el denominado patrón pequeño (**R1 + R2 + R2**) generan los siguientes resultados:

<i>t</i>	50	0.80%	0.60%	0.80%	1.10%	1.73%	2.00%	2.68%	4.86%	9.30%
	100	1.20%	1.40%	2.13%	2.70%	3.53%	4.35%	5.56%	9.08%	16.16%
	200	1.60%	2.00%	2.80%	3.80%	5.40%	6.75%	8.44%	14.32%	25.46%
	400	2.00%	4.00%	5.07%	6.70%	9.87%	12.70%	15.44%	24.70%	39.64%
	500	1.60%	3.80%	5.73%	7.40%	11.47%	14.65%	17.80%	29.54%	45.17%
	1000	5.60%	11.80%	16.13%	18.90%	25.87%	30.75%	35.08%	51.08%	67.09%
	250	500	750	1000	1500	2000	2500	5000	10000	
										<i>m</i>

Figura 4.10: Porcentajes de colisiones para las tablas empleando el patrón pequeño

De nuevo se obtienen unos resultados que no cumplen con lo inicialmente esperado en ciertos aspectos. Con el uso de este patrón se esperaba reducir el número de colisiones respecto a tabla que empleasen tan sólo **R1** o **R2**, lo cual como puede observarse ocurre con claridad. Lo inesperado en este caso es que estas tablas mejoren en cuanto al porcentaje de colisiones respecto al patrón grande, siendo esta mejora más que notable. Las tablas que emplean este patrón pequeño poseen el menor número de colisiones, llegando incluso a no existir ninguna tabla con más de un 67 % de colisiones. Tan sólo nueve tablas de las 54 totales en este caso superan el 25 por ciento de colisiones, y de estas únicamente dos pasan del 50 por ciento. Sin duda alguna estos son los mejores resultados en cuanto a colisiones se refiere.

4.5 Resultados finales

Habiendo obtenido los porcentajes de colisiones para las tablas con cada una de las combinaciones de funciones de reconstrucción es el momento de realizar el ataque del arco iris con ellas. De esta forma se espera determinar qué tamaño de tabla es el ideal en cuanto a contraseñas rotas, tiempo empleado en su generación y memoria requerida para su almacenamiento. En principio lo más lógico sería esperar que las tablas con menor número de colisiones fueran las que mejor rendimiento tuvieran, ya que cuantas menos colisiones se den lugar en una tabla mayor será el número de contraseñas cubiertas por esa tabla. Siguiendo esta lógica, las tablas con mayor éxito deberían de ser las que hacen uso del patrón pequeño (**R1 + R2 + R2**), ya que presentan una diferencia significante en sus porcentajes de colisiones, como se ha visto en la Figura 4.10.

Como ya se ha hecho anteriormente en la sección 4.3, cada una de las tablas que se pretende analizar tendrá la tarea de atacar las mismas contraseñas aleatorias generadas en la sección 4.2, obteniendo el porcentaje de contraseñas rotas respecto al total de contraseñas a romper.

Comenzando como es habitual por las tablas que tan sólo hacen uso de la función de reconstrucción con la operación módulo un millón (**R1**) se obtienen los siguientes resultados:

<i>t</i>	50	14.60%	27.90%	39.00%	46.60%	59.40%	67.80%	73.80%	87.40%	94.70%
	100	22.10%	41.90%	55.80%	65.80%	77.20%	81.60%	85.90%	92.30%	96.40%
	200	24.00%	46.20%	61.20%	71.30%	81.60%	87.70%	91.00%	95.20%	97.70%
	400	23.70%	45.00%	60.50%	69.90%	82.10%	88.50%	91.80%	96.00%	98.30%
	500	23.50%	44.30%	58.20%	67.00%	78.30%	84.10%	87.90%	94.60%	97.90%
	1000	22.70%	42.40%	55.10%	63.50%	72.70%	77.60%	80.70%	86.20%	89.30%
	250	500	750	1000	1500	2000	2500	5000	10000	
							<i>m</i>			

Figura 4.11: Porcentajes de éxito para las tablas que emplean **R1**

Como era de esperar, las tablas de menor tamaño no realizan un buen trabajo a la hora de adivinar contraseñas. Aunque dichas tablas tenían porcentajes de colisiones realmente bajos, puede observarse ahora como esa característica no es suficiente para romper muchas contraseñas. Hace falta que las tablas posean al menos 1.000 filas para comenzar a ser efectivas, y una vez superan las 2.000 filas los resultados encontrados son en su mayoría satisfactorios. En esta ocasión no se han generado tablas con efectividad perfecta, es decir, con un porcentaje de éxito del 100 %, aunque algunas tablas se han quedado a las puertas de conseguirlo.

Es interesante también observar que la profundidad ideal para las tablas con esta configuración es de 400 columnas para las tablas con más de 1.000 filas, lo cual muestra como una mayor profundidad de tabla no se traduce en un porcentaje de éxito más elevado. Esto es debido a que cuanto más se aumenta la profundidad de una tabla más veces se genera el valor *hash* de una contraseña para ser reconstruido a continuación, con lo cual aumenta la probabilidad de generar colisiones en la tabla, reduciendo el número de contraseñas cubiertas por dicha tabla, resultando en un menor porcentaje de éxito.

Viendo estos resultados, si se pretende emplear la tabla de menor tamaño posible que sea aceptable a la hora de romper contraseñas, habría de emplearse la tabla de dimensiones $m = 2.000$, $t = 200$, mientras que los mejores resultados los brinda la tabla de dimensiones $m = 10.000$, $t = 400$.

Pasando ahora a las tablas que emplean únicamente la función de reconstrucción con el *hash* en hexadecimal (**R2**) se obtienen los siguientes resultados:

	50	15.90%	29.70%	40.20%	48.80%	61.20%	69.70%	76.00%	88.60%	94.40%
<i>t</i>	100	22.20%	41.70%	56.20%	66.20%	76.90%	82.80%	87.00%	93.40%	97.10%
	200	24.10%	44.40%	61.20%	70.30%	82.10%	87.30%	90.20%	94.30%	97.40%
	400	22.10%	41.00%	57.00%	64.70%	74.60%	78.90%	82.10%	90.10%	93.80%
	500	21.10%	38.20%	53.10%	60.80%	69.30%	74.90%	77.60%	81.90%	85.30%
	1000	20.50%	35.60%	48.00%	57.10%	69.40%	73.80%	76.10%	79.30%	79.60%
		250	500	750	1000	1500	2000	2500	5000	10000
								<i>m</i>		

Figura 4.12: Porcentajes de éxito para las tablas que emplean R2

Como se puede observar, aunque para las tablas más pequeñas se han obtenido mejores resultados, en cuanto las dimensiones aumentan los resultados empeoran respecto a los anteriores. En esta ocasión se han generado el mismo número de tablas con un porcentaje de éxito por debajo del 70%, pero existen más tablas que no llegan a ser aceptables, así como menos tablas con un buen rendimiento. Incluso la mejor de estas tablas rinde peor que la mejor de las anteriores (97'40 < 98'30). Se puede asumir que este empeoramiento es debido al mayor porcentaje de colisiones de estas tablas (Figura 4.7) respecto a las anteriores (Figura 4.6). Otra diferencia que se puede apreciar al comparar estos resultados con los obtenidos en la Figura 4.11 es que, en esta ocasión, la profundidad de tabla ideal parece ser $t = 200$, lo cual es algo razonable debido al mayor número de colisiones, ya que como se ha explicado antes, mayor profundidad equivale a más colisiones.

En esta ocasión, de nuevo, la tabla de menor tamaño con un rendimiento aceptable ha de tener 2.000 filas.

Atacando las contraseñas con las tablas que entrelazan ambas funciones de reconstrucción (**R1 + R2**) se obtiene:

	50	15.50%	29.40%	42.50%	51.50%	65.80%	75.90%	82.40%	96.10%	98.80%
<i>t</i>	100	23.80%	45.40%	63.30%	74.50%	88.30%	93.80%	96.20%	99.40%	99.90%
	200	24.40%	47.30%	68.20%	83.60%	94.40%	97.20%	98.10%	99.90%	100.00%
	400	24.40%	45.30%	64.10%	77.30%	92.60%	96.80%	98.10%	99.80%	100.00%
	500	24.20%	44.50%	62.30%	74.40%	89.90%	95.40%	97.70%	99.60%	100.00%
	1000	22.60%	39.30%	53.10%	63.10%	77.30%	85.90%	89.90%	95.40%	97.90%
		250	500	750	1000	1500	2000	2500	5000	10000
								<i>m</i>		

Figura 4.13: Porcentajes de éxito para las tablas que emplean R1 + R2

Como ya ocurriera en las primeras pruebas realizadas en la sección 4.3, las tablas que emplean la secuencia de funciones de reconstrucción **R1 + R2** consiguen romper el mayor número de contraseñas. El número de tablas con un rendimiento pobre ha reducido, mientras que ha aumentado de forma significativa la cantidad de tablas idóneas, algunas de ellas llegando incluso a ser capaces de romper todas las contraseñas almacenadas. Al igual que ha ocurrido con las tablas anteriores

(Figura 4.12), la profundidad ideal es de $t = 200$, lo cual permite un ahorro de tiempo más que notable en el proceso de construcción de las tablas.

En este caso se ha visto reducido el número mínimo de filas necesarias para que una tabla sea considerada como aceptable, $m = 1500$. También se ha podido ver que con una configuración de funciones de reconstrucción apropiada no es necesario generar tablas de más de 10.000 filas, lo cual permite reducir los requerimientos de memoria significativamente.

Observando los resultados obtenidos con tablas que emplean el denominado patrón grande (**R1 + R2 + R2 + R1 + R1 + R2 + R2 + R2**) se obtiene:

	50	4.60%	8.90%	14.20%	19.60%	27.40%	33.00%	40.20%	63.10%	82.80%
<i>t</i>	100	12.20%	22.90%	32.70%	41.20%	54.50%	62.90%	70.40%	87.40%	95.90%
	200	20.70%	38.10%	51.50%	61.40%	74.50%	81.50%	86.70%	96.00%	98.80%
	400	23.40%	44.10%	58.80%	68.90%	82.70%	89.00%	92.20%	98.00%	99.30%
	500	23.90%	46.10%	62.00%	73.40%	85.10%	90.40%	94.00%	98.30%	99.60%
	1000	22.20%	38.90%	51.00%	60.40%	74.10%	80.80%	84.30%	92.60%	95.50%
		250	500	750	1000	1500	2000	2500	5000	10000
		<i>m</i>								

Figura 4.14: Porcentajes de éxito para las tablas que emplean el patrón grande

Aunque en esta ocasión no se ha conseguido mejorar respecto a los resultados anteriores, sí que se aprecia un mejor rendimiento que las tablas que hacen uso de tan sólo una función de reconstrucción. A pesar de generar más tablas con malos porcentajes de éxito, el número de tablas idóneas se ha visto aumentado respecto a las Figuras 4.11 y 4.12, dos de las cuales se han quedado muy cerca de romper todas las contraseñas. Puede apreciarse también algo que no había ocurrido hasta este momento. Para las tablas de las figuras anteriores, aquellas que superan las 2.000 filas son, como mínimo, aceptables. Por contrario, en este caso existe una tabla con 5.000 filas con un porcentaje de éxito por debajo del 70 por ciento. También cambia el valor ideal del parámetro t , el cual en esta ocasión es de 500 columnas. Esto se le atribuye a el patrón establecido, ya que al desarrollarse durante ocho entradas consecutivas requiere de una profundidad de tabla mayor para obtener buenos resultados.

Al igual que en los resultados de la Figura 4.13, una tabla con un mínimo de 1.500 filas daría un buen rendimiento mientras mantiene un tamaño reducido.

Por último, realizando el ataque con las tablas que emplean el denominado patrón pequeño (**R1 + R2 + R2**) pueden obtenerse los siguientes resultados:

	50	8.20%	17.40%	25.20%	31.50%	41.10%	49.80%	55.60%	76.60%	90.30%
	100	18.20%	33.70%	46.90%	56.00%	68.50%	76.60%	81.40%	92.50%	97.80%
	200	22.40%	42.40%	58.50%	69.80%	81.90%	87.40%	90.60%	96.70%	99.10%
	400	24.50%	46.60%	61.90%	71.90%	82.20%	87.70%	90.10%	96.60%	98.90%
	500	24.60%	47.70%	63.40%	71.50%	82.20%	87.20%	91.00%	96.60%	98.60%
	1000	23.60%	43.90%	60.60%	71.30%	83.20%	89.50%	91.50%	95.50%	97.80%
		250	500	750	1000	1500	2000	2500	5000	10000
							m			

Figura 4.15: Porcentajes de éxito para las tablas que emplean el patrón pequeño

Estos resultados son mejores que los obtenidos con el patrón grande, aunque no superan a los obtenidos en la Figura 4.13. Tan sólo resultan peores en un apartado respecto al patrón grande, y es en el tamaño mínimo de tabla para obtener resultados satisfactorios, ya que se ha pasado de requerir 1.500 filas a 2.000, aunque la diferencia de almacenamiento en memoria es nimia. De nuevo ha habido una tabla muy cerca de romper las 1.000 contraseñas almacenadas. Hay un hecho característico en estas tablas, el cual corresponde con la profundidad ideal. Si bien para cualquiera de los resultados anteriores existe cierto número de columnas que presenta el mejor rendimiento, en este para no se aprecia un único valor de t que brinde los mejores resultados para cada valor de m . Observando los resultados con detenimiento se puede apreciar que, dependiendo del número de filas, la cantidad ideal de columnas será $t = 200$ o $t = 1.000$ para las tablas con un mínimo de 1.000 filas. Como último apunte, en este caso ocurre igual que en las Figuras 4.11 y 4.12 a la hora de buscar la tabla de tamaño mínimo con resultados aceptables, lo cual se da con $m = 2.000$ filas.

CAPÍTULO 5

Conclusiones

Tras observar la idea general de la primera implementación de Hellman [1] y tener en cuenta la mejora propuesta por Rivest [2], el ataque del arco iris lleva estos conceptos más allá, con el objetivo de mejorar los resultados obtenidos anteriormente y también adaptar estas implementaciones a las nuevas necesidades establecidas por los avances informáticos.

Tras establecer el funcionamiento general en cuanto a la construcción de las tablas se refiere, el paso siguiente corresponde a determinar una o varias funciones de reconstrucción a emplear durante dicha construcción. Tras varias pruebas con diversas funciones se optó por emplear las ya explicadas **R1** y **R2**. En un principio tan sólo se iban a dar lugar dos configuraciones de tabla diferentes, empleando exclusivamente una de las funciones de reconstrucción en cada configuración. Tras la obtención de los resultados generados a raíz de implementar el ataque del arco iris con estas dos configuraciones surgió la idea de combinar ambas funciones de reconstrucción, con el objetivo en mente de reducir el número de colisiones de las tablas, permitiendo aumentar la cantidad de contraseñas cubiertas por una tabla. En cuanto a esto, la primera propuesta fue entrelazar ambas funciones (**R1 + R2**), mientras que una vez generadas estas tablas se trató de reducir todavía más las colisiones existentes, estableciendo el denominado patrón grande (**R1 + R2 + R2 + R1 + R1 + R2 + R2 + R2**), el cual consiguió reducir la cantidad de colisiones, como se muestra en la Figura 4.9. La configuración restante con las tablas que hacen uso del denominado patrón pequeño (**R1 + R2 + R2**) se planteó una vez obtenida la primera tanda de resultados, ya que el patrón grande no fue capaz de generar mejores resultados que los obtenidos hasta ese momento.

Habiendo implementado el ataque del arco iris con un total de 370 tablas diferentes, variando las dimensiones y el uso de funciones de reconstrucción de las mismas, se ha podido observar el rendimiento en cuanto a la cantidad de contraseñas rotas se refiere.

En una primera instancia se optó por implementar tablas con dimensiones no menores de $m = 1.000$ filas, debido a que generar tablas con un menor número de filas habría de compensarse con una mayor profundidad para cubrir todo el dominio $N = 1,000,000$, lo cual podría resultar en un aumento notable del número de colisiones en dichas tablas. Tras realizar estos primeros ataques y obtener sus resultados en la sección 4.3 es posible establecer unas primeras conclusiones sobre ellos. Para empezar, prestando atención antes que nada a la calidad de los

resultados en sí, es decir, priorizando las tablas con porcentajes de éxito mayores, queda claro que la mejor combinación de funciones de reconstrucción se da empleando sucesivamente **R1** y **R2**, como puede observarse en la Figura 4.3. La segunda conclusión clara que puede establecerse es que, una vez se emplean tablas de 5.000 filas o más, cualquiera de las tablas empleadas en este primer ataque consigue romper la mayoría de las contraseñas. En el caso de esta implementación en concreto, esto sucede con tal cantidad de filas debido a que el dominio N no resulta ser excesivamente elevado. También cabe destacar que, aunque para estos primeros ataques se emplearon tablas llegando incluso a tener un millón de filas que cubrían todas las contraseñas o incluso superaban el dominio establecido, esta práctica no será factible en aplicaciones con un dominio mayor. El motivo de generar dichas tablas en esta ocasión fue el tratar de llegar lo más cerca posible al límite de filas. Es importante indicar que dichas tablas realmente no resultan prácticas en este ataque, ya que aunque consiguen romper todas las contraseñas almacenadas, existen otras tablas de menor tamaño con las cuales se dan los mismos resultados por lo cual serían dichas tablas más pequeñas las empleadas para realizar el ataque del arco iris, ya que se ahorraría una cantidad de tiempo considerable entre la generación de la tabla y la búsqueda de las contraseñas. Por último, es fácilmente observable que las tablas que hacen uso exclusivamente de la función de reconstrucción **R2** brindan los peores resultados, aunque sean satisfactorios en general. Esto fue un hecho sorprendente en primer lugar, ya que se esperaba que esta función de reconstrucción fuera capaz de generar tablas con menor cantidad de colisiones, lo cual no ocurrió finalmente como se puede observar en la Figura 4.7.

Tras esta primera tanda de resultados, viendo como ya se ha dicho los elevados porcentajes de éxito para tablas con al menos 5.000 filas, el foco de atención se centró en buscar las tablas de menor tamaño posible que proporcionaran resultados similares a las tablas de mayores dimensiones en cuanto a la calidad de los mismos se refiere. Para ello, se estableció el número mínimo de filas en 250, mientras que el máximo quedó en 10.000 filas. Cabe destacar que de las tablas con un número de filas reducido no se esperaba un gran rendimiento, debido a que cubrían una cantidad de contraseñas inferior a el dominio N . Aún así, generarlas y obtener sus resultados ha resultado ser necesario para poder establecer una frontera con claridad en las búsquedas por la tabla más eficiente de construir con mejores resultados. Habiendo alcanzado este punto no se estimó necesario establecer una nueva configuración para las tablas, por lo cual el proceso de obtención de resultados fue idéntico al empleado en la sección 4.3.

Una vez generadas todas las tablas necesarias con los nuevos tamaños, antes de realizar el ataque del arco iris con cada una de ellas se optó por obtener el número de colisiones para cada una de ellas. Esto sirvió para poder reflejar una de las dos medidas de calidad de las configuraciones de las tablas, siendo la segunda de estas medidas el porcentaje de éxito. Analizando los resultados en cuanto a los porcentajes de colisiones se refiere, se aprecia que la función de reconstrucción **R2** es la peor en este aspecto de manera considerable, mientras que el patrón pequeño (Figura 4.10) genera las tablas con el menor número de colisiones, lo cual en principio podría emplearse como argumento para predecir que estas tablas conseguirían romper la mayor cantidad de contraseñas.

Tras implementar el ataque del arco iris con las nuevas tablas y representar los resultados obtenidos, de nuevo las configuraciones establecidas podrían clasificarse de la siguiente manera empleando como criterio de clasificación la cantidad de contraseñas rotas:

1. **R1 + R2**
2. **Patrón pequeño**
3. **Patrón grande**
4. **R1**
5. **R2**

Una vez observada esta clasificación de los resultados se puede tratar de establecer los motivos por los cuales se da lugar dicha clasificación para esta implementación en concreto. De entre ellos, los dos motivos con mayor peso son los siguientes:

- **Variedad en la reconstrucción:** Como queda reflejado, emplear únicamente una función de reconstrucción brinda los peores resultados. Tras esta observación, es coherente pensar que emplear una combinación de ambas funciones asegurará un mejor rendimiento, lo cual efectivamente ocurre en este caso. A la hora de elegir una combinación de funciones de reconstrucción se ha de tener en cuenta el comportamiento de cada función por separado, siendo el porcentaje de colisiones el aspecto más importante. Esto queda reflejado si se comparan los resultados obtenidos con los dos patrones establecidos, ya que el patrón pequeño obtiene mejores resultados al hacer un menor uso de la función de reconstrucción **R2** que el patrón grande, lo cual le permite cubrir una mayor cantidad de contraseñas, aunque no tantas como al emplear sucesivamente ambas funciones (**R1 + R2**). Siendo esta última combinación la mejor de todas, queda demostrado que no se requiere una combinación más compleja para obtener mejores resultados.
- **Colisiones en las tablas:** Aunque este aspecto por sí sólo no es el más importante, es obvio que tiene un impacto significativo a la hora de romper contraseñas. Si de hecho para obtener mejores resultados únicamente hiciera falta obtener tablas con el menor número de colisiones posible, el patrón pequeño quedaría por encima de cualquier otra de las configuraciones, ya que existe una diferencia más que notable en cuanto al porcentaje de colisiones obtenido (Figura 4.10) respecto a las demás configuraciones. Dicho esto, la importancia de mantener un número reducido de colisiones queda reflejada en las tablas que hacen uso exclusivamente de la función de reconstrucción **R2** (Figura 4.7), ya que dichas tablas son las que poseen un mayor número de colisiones y a la vez presentan los peores resultados (Figura 4.12). La clave del éxito de una tabla en cuanto a su porcentaje de colisiones se refiere reside en ser capaz de cubrir un rango suficiente de contraseñas aún teniendo colisiones que reduzcan dicho rango.

Una vez analizados los resultados obtenidos, se debe tener en cuenta que este es un caso particular de esta implementación, es decir, estos resultados se han dado de esta forma debido a la elección de la función *hash* y dominio de contraseñas a atacar. Si se eligiera atacar una función *hash* que produjera *hashes* de mayor tamaño, manteniendo las mismas funciones de reconstrucción y configuraciones vistas, probablemente los porcentajes de colisiones de las tablas se verían reducidos, ya que con valores *hash* más largos disminuye la probabilidad de colisiones en las tablas. Este también sería el caso si se atacara un dominio de contraseñas de mayor longitud aunque se mantuviese el uso de la función *hash* CRC-32, ya que dicha función siempre genera valores *hash* de 32 bits de longitud, por lo cual las funciones de reconstrucción seguirían recibiendo como entrada cadenas de 32 bits de longitud pero producirían como salida contraseñas de dicha longitud mayor, reduciendo de esta forma la probabilidad de generar colisiones.

Otro de los aspectos brevemente comentados anteriormente sobre el cual se puede expandir es la escalabilidad del ataque del arco iris. Si para la implementación desarrollada en este caso se decidiera atacar una función *hash* diferente o un dominio de contraseñas que contuviera un mayor número de ellas, los cambios necesarios en la implementación serían mínimos, siendo la diferencia más notable un aumento en los tiempos empleados en construcción de tablas y búsqueda de contraseñas en las mismas. Este es uno de los motivos por el cual el ataque del arco iris ha resultado ser tan eficiente, ya que es suficiente con implementarlo una primera vez para una función *hash* y dominio concretos para tener la capacidad de implementar el ataque en una variedad de condiciones distintas, requiriendo generalmente pequeños cambios para cada condición específica, siendo una condición una tupla de función *hash* y dominio de contraseñas a atacar.

En cuanto al concepto de las funciones de reconstrucción cabe destacar varios aspectos de vital importancia para su correcto comportamiento en el ataque del arco iris. Principalmente, se requieren dos características fundamentales que cualquier función de reconstrucción ha de conocer de antemano, sin las cuales le será imposible realizar un buen trabajo:

- **Longitud de las cadenas de entrada y salida:** un aspecto crucial para poder establecer el método de reconstrucción de valores *hash* a contraseñas. En el caso de no tener conocimiento de dichas longitudes se correría el riesgo de producir contraseñas no válidas para el dominio establecido. Además, puede resultar de gran utilidad, ya que se pueden establecer relaciones entre los tamaños de las cadenas de entrada y salida que faciliten la obtención de la función de reconstrucción.
- **Caracteres permitidos del dominio de contraseñas:** al igual que ocurre con el tamaño de las cadenas, la función de reconstrucción requerirá tener conocimiento de los caracteres que las conformen, para imposibilitar el caso de generar una contraseña que quede fuera de los límites establecidos por el dominio. Un claro ejemplo podría ser que, tomando el dominio de contraseñas *N* de la implementación vista en el capítulo anterior, cualquier función de reconstrucción empleada en ese caso jamás debería producir una cadena con uno o más de sus caracteres distintos de los números del 0 al 9.

Otro aspecto que resulta también importante aunque no lo es tanto que los anteriores es la velocidad de reconstrucción de valores *hash* a contraseñas, ya que es una operación que se va a aplicar tan a menudo como la función resumen elegida durante cualquiera de las fases del ataque del arco iris. También es importante destacar que el hecho de que una función de reconstrucción que sea peor que otra diferente para una implementación específica (es decir, que genere peores resultados que otra) no significa que vaya a ser siempre así. Podría encontrarse otra implementación diferente, con otra función *hash* y dominio de contraseñas para los cuales la función que anteriormente brindaba peores resultados funcione mejor. Por este motivo generalmente resulta beneficioso tratar de utilizar más de una función de reconstrucción para una misma implementación, ya que de esta forma se puede determinar la mejor función o configuración de funciones, como se ha realizado en este proyecto.

La estructura de datos empleada para representar la tabla también resulta una decisión importante, aunque el grado de importancia dependerá del tamaño del dominio de contraseñas que se pretende atacar. En la implementación aquí presentada, al ser el dominio de un tamaño considerado pequeño, incluso las tablas más grandes no han llegado a tardar más de tres minutos tanto en su generación como al atacar las contraseñas, por lo cual no se ha requerido buscar la mayor eficiencia posible. En el caso de realizar el ataque contra un dominio de contraseñas de tamaño considerable, el cual con la implementación actual pudiera multiplicar los tiempos requeridos, se habría de fijar como objetivo adicional encontrar la estructura de datos más eficiente. De nuevo, esto entra dentro de la escalabilidad del ataque del arco iris, ya que por mucho que la estructura de datos varíe, la forma de llevar a cabo el ataque permanecerá siendo la misma.

Tras todo esto, la efectividad del ataque del arco iris debería haber quedado suficientemente clara. Siempre que se realice el ataque frente a un sistema que almacene valores *hash* de las contraseñas originales mediante el uso de una función *hash* que no utilice técnicas como el *salting* o el estiramiento de contraseñas (explicados en la sección 2.3), las tablas del arco iris serán capaces de adivinar la gran mayoría de las contraseñas. Una de las claves para dicho éxito recae en emplear funciones de reconstrucción adecuadas, para lo cual es necesario dedicar el tiempo adecuado a la experimentación con diversas de estas funciones, así como los diferentes parámetros que configuran una tabla en concreto. Como ha sido siempre habitual en la criptografía, a medida que se han ido reforzando las medidas de seguridad frente a ataques informáticos también se ha conseguido mejorar la efectividad de los mismos, lo cual motiva a un refuerzo mayor de la seguridad de los sistemas, formando un ciclo el cual continúa hoy en día. Si bien como ya se ha dicho, actualmente resulta sencillo inutilizar el ataque del arco iris, todavía existen sistemas que siguen siendo vulnerables a sus tablas, como por ejemplo todos los sistemas operativos de Microsoft Windows hasta llegar a Windows 7. También, aunque el objetivo principal del ataque del arco iris siempre es romper las máximas contraseñas posibles, se le puede dar un uso diferente, el cual consiste en atacar dos funciones *hash* diferentes haciendo uso de la misma configuración de tablas del arco iris, para así poder comparar la seguridad de ambas funciones. Esto puede ser útil en aquellos casos en los cuales por especificaciones de diseño u otros motivos no sea posible hacer uso del *salting* o el estiramiento. Incluso se podría hacer uso de una implementación en particular del ataque del arco iris

para comparar la eficiencia de diferentes estructuras de datos, ya que estarán sometidas a las mismas operaciones. Todos estos aspectos sirven para observar que, aunque en un principio es normal pensar en el ataque del arco iris como un ataque informático más, se le puede dar diferentes usos dependiendo del objetivo que se esté tratando de cumplir.

Bibliografía

- [1] M. E. Hellman. A cryptanalytic time-memory trade-off. *IEEE Transactions on Information Theory*, VOL. IT-26, NO. 4, julio 1980.
- [2] D.Denning. *Cryptography and Data Security*, p.100, Addison-Wesley, 1982.
- [3] Oechslin P. (2003) Making a Faster Cryptanalytic Time-Memory Trade-Off, Boneh D. (eds) *Advances in Cryptology - CRYPTO 2003*, Lecture Notes in Computer Science, vol 2729. Springer, Berlin, Germany.

APÉNDICE A

Configuración del sistema

Por determinar