UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escuela Tica Superior de Ingeniernformca
Universidad Politica 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:** criptografía, identificación, password, función resumen, tabla del arco iris

# Resum

L'accés al sistemes informàtics ha estat des-de sempre lligat a l'ús de paraules de pas o passwords. Per motius de seguretat, els passwords son 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 son invertibles, amb una probabilitat de col·lisió inversament proporcional exponencialment al nombre de bits del resum. Una aproximació per a encontrar dites col·lisions son 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:** criptografia, indentificació, password, funció resum, 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:** criptography, identification, password, hash function, rainbow table

# Índice general

# Índice de figuras



# Índice de tablas

# Índice de algoritmos

# CAPÍTULO 1
# Introducción

La criptografía se puede definir como el estudio de técnicas matemáticas relacionadas con el mantenimiento seguro de cualquier tipo de información[1]. Tiene una historia muy extensa, existiendo pruebas de sus primeros usos hace 4000 años, pasando por las primeras implementaciones en los sistemas informáticos hasta llegar al momento actual. Ligado al aumento del acceso a los sistemas informáticos surgió una nueva demanda de acceso a medios de protección de información digital, lo cual sirvió como estímulo para la investigación y obtención de nuevas técnicas y métodos de mantenimiento seguro de la información almacenada en los sistemas informáticos.

## 1.1 La criptografía como piedra angular

Existen infinidad de motivos interesantes para proteger información digital privada de forma segura, pudiendo ser los más destacados los siguientes:

1. **Confidencialidad**: mantener la información privada oculta excepto para quienes tengan autorización para su obtención. Hoy en día, es habitual emplear algoritmos matemáticos con el fin de hacer la información ininteligible para usuarios no autorizados.
2. **Autenticación**: sinónimo habitual de identificación, pudiéndose aplicar tanto a entidades interesadas en información como a la propia información.
3. **Integridad de datos**: relacionado con la modificación no autorizada de información. Es necesario poseer la habilidad de detectar cualquier alteración de los datos por un usuario no autorizado si se desea garantizar la integridad de dichos datos.
4. **No repudio**: prevenir la negación de la realización de acciones previas, o lo que es lo mismo asegurar la realización de acciones previas.
5. **Firma**: ligar información específica a una entidad de confianza.

Los métodos criptográficos empleados habitualmente consisten en el cifrado de mensajes y el uso de la firma electrónica. Con los objetivos vistos arriba en mente, una de las herramientas más comunes para llevar a cabe estos métodos son las funciones resumen.

## 1.2 Características y usos de las funciones resumen

Aunque existen una gran variedad de técnicas criptográficas con el objetivo en mente de garantizar los puntos anteriores, en este proyecto el foco ha sido puesto mayoritariamente en la utilización de una de ellas: las funciones resumen. En concreto, se buscará introducir una implementación que suponga una mejora respecto a procedimientos anteriores empleados para la búsqueda de colisiones para este tipo de funciones.

### 1.2.1. Definición

También conocidas como funciones *hash*, estas funciones 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*.

Una función resumen $h$ cumple que, para una cadena binaria de salida $x$ de longitud igual a $n$ bits, la probabilidad de que un mensaje de entrada $m$ aleatorio resulte en un valor *hash* igual que $x$ es de $2^{-n}$. Esto es lo que se conoce como una colisión, lo cual representa el punto central de este proyecto y se explicará en detalle más adelante.

### 1.2.2. Propiedades

Las funciones resumen poseen diversas características además de las vistas en el anterior apartado, las cuales resultan de gran utilidad para su uso. Las más notables 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 de la misma longitud son equiprobables, es decir, tienen 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.

Para el uso criptográfico, existen también otro tipo de características que sirven para determinar el nivel de seguridad de una función resumen en particular,

las cuales son conocidas como las resistencias. Existen tres tipos diferentes, y dependiendo de a cuáles de ellas sea resistente una función *hash* se podrá establecer su grado de seguridad:

1. **Resistencia a pre-imagen**: conociendo un valor *hash* $y$ sin conocer su mensaje de entrada original, es computacionalmente inviable encontrar un mensaje de entrada $m$ que genere $y$ como valor *hash*, es decir $h(m) = y$. En este caso, el mensaje $m$ es considerado como la pre-imagen.

2. **Resistencia a segunda pre-imagen**: conociendo un mensaje de entrada $m$, es computacionalmente inviable encontrar otro mensaje de entrada diferente $m'$ de tal forma que el valor *hash* producido por ambos mensajes sea el mismo, tal y como $h(m) = h(m')$. En este caso, el mensaje $m'$ es considerado como la segunda pre-imagen.

3. **Resistencia a colisiones**: Resulta computacionalmente inviable encontrar dos mensajes de entrada diferentes entre sí, $m_1$ y $m_2$, los cuales generan el mismo valor *hash*, de tal forma que $h(m_1) = h(m_2)$.

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.

Como ya ha sido indicado anteriormente, las colisiones constituyen uno de los puntos fundamentales de este proyecto. Su obtención para una función resumen puede representar una vulnerabilidad, pudiendo de esta forma sortear la barrera que representan.

### 1.2.3. Usos habituales

Aunque existen una gran variedad de aplicaciones de las funciones resumen, dos de los más populares son los vistos a continuación:

- **Firma electrónica**: este tipo de protocolos son empleados para verificar la autenticidad de mensajes o documentos digitales. Las funciones *hash* constituyen una parte fundamental para llevar a cabo este objetivo, como se puede ver en el siguiente ejemplo, el cual representa un intercambio de información digital de un mensaje $x$ entre dos entidades, desde $E1$ para $E2$, empleando una función resumen $h$ y una firma electrónica $F$:

  1. Ambas entidades $E1$ y $E2$ se ponen de acuerdo en la función resumen $h$ a emplear.
  2. La entidad firmante, $E1$, desea mandar un mensaje $x$ a la entidad $E2$. Para ello, manda una tupla $t$ que consiste en el mensaje $x$ por una parte, y por la otra la firma del valor *hash* de $x$, de tal forma que $t = \{x, F[h(x)]\}$.

3. La entidad receptora, $E2$, al recibir la tupla $t$ verifica que ha sido mandada por $E1$ obteniendo el valor *hash* del mensaje $x$, de tal manera que si coincide con lo recibido todo ha ido según lo esperado, mientras que si no es así puede indicar un uso ilegítimo del sistema.

- **Acceso a un sistema mediante una contraseña**: sea un sistema operativo o la cuenta de usuario de una página web, el uso de contraseñas para ganar acceso a un entorno funciona mediante la utilización de las funciones resumen. Al asociar una contraseña a un usuario, la información almacenada en el sistema no es la contraseña como tal, sino su valor *hash*. En el caso de tratar de acceder a un sistema que emplea una función resumen $h$, al introducir una contraseña $p$ se calcula su valor *hash* $h(p)$, y se comprueba que sea igual que el valor asociado al usuario en cuestión. Si coinciden, se garantiza el acceso, de lo contrario siendo denegado.

En ambos casos, el método de superar la barrera impuesta por la función resumen es el mismo: encontrar una colisión para el valor *hash* deseado.

## 1.3 Motivación

La obtención de colisiones dentro de las funciones resumen sirve para vulnerar y en muchos casos inutilizar la protección que tratan de proporcionar.

Aunque inicialmente pueda resultar una tarea que requiera grandes cantidades de tiempo y memoria, existen métodos que consiguen reducir estos requerimientos con el fin de obtener dichas colisiones de manera eficiente. Dichos métodos se han ido mejorando con el paso del tiempo, quedando a su vez inutilizados por las medidas de seguridad que surgían como contramedida. Así mismo, las funciones *hash* han evolucionado acordemente, tratando de resultar más seguras contra estos métodos.

Estos avances han llevado al momento actual, en el cual se tiene acceso a una gran variedad de técnicas de obtención de colisiones para funciones *hash*, algunas más eficientes que otras, así como la disponibilidad de diversas funciones resumen. Obtener un esquema capaz de obtener colisiones para una función *hash* en un tiempo eficiente es aplicable a diferentes ámbitos, desde la obtención no autorizada de información hasta la experimentación con diferentes funciones resumen para determinar sus grados de seguridad frente a estos ataques.

Cualquier avance en el campo de la obtención de colisiones para funciones resumen permitirá una evolución de las mismas. En este sentido se plantea este trabajo.

## 1.4 Objetivos

El objetivo principal de este trabajo consiste en diseñar un método eficiente de búsqueda de colisiones para funciones resumen. De esta manera se obtendría una implementación capaz de atacar cualquier función *hash* con el objetivo de

encontrar colisiones dentro de la misma, pudiendo ser empleada para diversos fines relacionados con el uso de estas funciones. Así mismo, la implementación desarrollada tendrá como objetivo adicional representar una mejora respecto a métodos anteriores, siguiendo un esquema ya establecido que consigue obtener dicha mejora. Dentro del mismo, se tratará de encontrar la implementación más eficiente, lo cual requerirá una amplia fase de experimentación.

## 1.5 Estructura de la memoria

Habiendo establecido una base introduciendo conceptos fundamentales tanto de la criptografía como de las funciones resumen, los siguientes capítulos se distribuirán de la siguiente forma:

- En el segundo capítulo se expondrán diferentes métodos de obtención de colisiones para funciones resumen, los cuales constituirán una base sobre la cual cimentar los capítulos venideros.
- En el tercer capítulo tendrá lugar la explicación en detalle del método empleado en este proyecto para la búsqueda de colisiones de funciones *hash*, relacionándolo con las aproximaciones vistas en el anterior capítulo y sirviendo como preámbulo a la experimentación desarrollada.
- En el capítulo cuatro se entrará en detalle de la implementación desarrollada para obtener un método eficiente de búsqueda de colisiones para funciones resumen, exponiendo las especificaciones establecidas y los resultados obtenidos a lo largo de todo el desarrollo.
- En el capítulo final se realizarán las conclusiones finales sobre todo lo visto, analizando los resultados obtenidos en el anterior capítulo y estableciendo posibles trabajos futuros.

# CAPÍTULO 2
# Aproximaciones en la búsqueda de colisiones en funciones resumen

El dicho popular "hecha la ley, hecha la trampa" también es aplicable en el campo de la criptografía, y dentro de todas sus posibilidades, las funciones resumen no quedan exentas. Aunque desde la aparición de estas funciones se han desarrollado infinidad de ataques diferentes, cada uno basándose en diferentes propiedades, los más relevantes para este proyecto vienen dados a continuación.

## 2.1 La paradoja del cumpleaños

La paradoja del cumpleaños tiene lugar en teoría de la probabilidad, y recibe su nombre del siguiente suceso. Si se toma un grupo de $p$ personas, existe una probabilidad de que dos de ellas tengan la misma fecha de nacimiento en cuanto al día y el mes, sin tener en cuenta el año. Es evidente que si se dispone de 367 personas, existe un 100 % de probabilidad de que al menos dos de ellas coincidan en este aspecto, ya que existen un total de 366 fechas de nacimiento diferente, incluyendo el 29 de febrero. La paradoja del cumpleaños toma forma cuando se dispone de un número menor de personas, ya que con tan sólo 70 personas, la probabilidad de que dos de ellas tengan el mismo cumpleaños asciende hasta el 99.9 %, mientras que con únicamente 23 personas existe una probabilidad del 50 % de que tenga lugar esta ocurrencia, como puede observarse en [2].

Extrapolando la paradoja del cumpleaños a las funciones *hash*, puede tomarse como ejemplo una función $h$ que genere resúmenes con una longitud de $n$ bits. En este caso, el número total de valores *hash* diferentes que pueden obtenerse mediante $h$ es de $2^n$. Sabiendo esto, si se pretendiera adivinar un valor *hash* aleatorio de $h$, sería necesario llevar a cabo $\mathcal{O}(2^n)$ operaciones, considerando la obtención de un resumen *hash* como una operación. Debido a la paradoja del cumpleaños, normalmente no va a ser necesario realizar tantos pasos, ya que tras realizar $\mathcal{O}(2^{\frac{n}{2}})$ pasos muy probablemente se habrá obtenido el valor deseado [1]. Esto ocurre siguiendo una propiedad fundamental de la paradoja del cumpleaños, la cual especifica que en un problema de este tipo, contando con $n$ elementos o candidatos, tras realizar $\mathcal{O}(\sqrt{n})$ selecciones existe una probabilidad muy alta de encontrar una coincidencia.

De esta manera surgieron los ataques del cumpleaños, basados en la paradoja homónima. Este tipo de ataques a funciones *hash* tan sólo tienen en cuenta el número de bits que poseen los valores *hash* y el tiempo empleado en la obtención de uno de estos valores.

### 2.1.1. El algoritmo de Yuval aplicado al ataque del cumpleaños

Una de las implementaciones más populares del ataque del cumpleaños vino dada por Yuval, consiguiendo con éxito ser capaz de encontrar colisiones en las funciones *hash* de aquel momento. La forma de llevar a cabo este ataque contra un sistema de firma electrónica se muestra en el algoritmo 2.1.

**Algorithm 2.1** Algoritmo de Yuval aplicando el ataque del cumpleaños

**Entrada:** mensaje legítimo $m_1$, mensaje fraudulento $m_2$, función *hash* $h$ que genera resúmenes de $n$ bits de longitud, $t = 0$.
**Salida:** $m_1'$, $m_2'$ resultantes de aplicar modificaciones menores a $m_1$, $m_2$, tal que $h(m_1') = h(m_2')$ (de esta forma la firma funcionaría tanto con $m_1'$ como con $m_2'$).

**while** $t < 2^{\frac{n}{2}}$ **do**
  Generar mensajes fraudulentos $m_1'$, resultados de aplicar modificaciones menores de $m_1$.
  Obtener el valor *hash* de cada $m_1'$ y guardarlo junto con su mensaje originario, para poder obtener el mensaje cuando se busque un valor *hash*.
**end while**

**while** $h(m_1') \neq h(m_2')$ **do**
  Generar mensajes fraudulentos $m_2'$ mediante modificaciones menores de $m_2$, cada vez obteniendo su valor *hash* correspondiente y comprobando si coincide con alguno de los valores *hash* de cualquiera de los $m_1'$, deteniendo en proceso una vez se encuentre tal coincidencia.
**end while**

Relacionando este ataque con la paradoja del cumpleaños, tras realizar $\mathcal{O}(2^{\frac{n}{2}})$ operaciones la probabilidad de encontrar una colisión se eleva de manera significativa. Esto llegó a plantear un serio problema para algunos métodos de firma electrónica de la época, ya que permitía usurpar la identidad de cualquier usuario de un sistema de firma electrónica vulnerable.

## 2.2 *Time-memory trade-off*

En julio de 1980 era publicado por Martin E. Hellman el primer artículo introduciendo el concepto del intercambio tiempo-memoria [3](dicho intercambio será referido a partir de ahora como TMTO). En dicho artículo, Hellman indica como el TMTO se puede aplicar con el objetivo de averiguar la información necesaria para descifrar textos de forma más eficiente que anteriormente visto. Para

este fin emplea estructuras de datos en forma de tablas, que sirven como piedra angular para el desarrollo de este tipo de ataques criptográficos.

El TMTO es un concepto aplicable a problemas con una cierta estructura, por ejemplo en los problemas de la mochila [4] o del logaritmo discreto [5] en sus espacios de búsqueda respectivos. De tal forma, si existen un total de $N$ soluciones posibles en el espacio de búsqueda, el TMTO permite encontrar la solución deseada en $T$ pasos empleando $M$ palabras de memoria, siempre que se cumpla que $T \cdot M = N$. Esto resulta ser mucho más eficiente en cuanto al tiempo empleado que la búsqueda por fuerza bruta, la cual puede requerir $N$ pasos para encontrar una solución.

### 2.2.1. Parámetros a tener en cuenta

Para ejemplificar la construcción de una tabla de Hellman, se puede emplear el criptosistema *Data Encryption Standard* (DES), el cual toma textos de 64 bits y los transforma en cifrados de 64 bits, empleando una clave de 56 bits, analizado más a fondo en [6]. Para el correcto desarrollo del método ideado por Hellman, son necesarios los siguientes parámetros:

- $P$: conjunto de posibles textos sin cifrar.
- $C$: conjunto de posibles textos cifrados.
- $K$: conjunto de claves empleadas para cifrar los textos de $P$, transformándolos en textos de $C$.
- $R$: función de reducción que transforma textos cifrados de $C$ en claves de $K$.
- $m$: número de filas de la tabla.
- $t$: número de columnas de la tabla.

El uso de los parámetros $P, C$ y $K$ es sencillo. Un texto sin cifrar $p \in P$ pasa a ser un texto cifrado $c \in C$ mediante el uso de una clave $k \in K$. De esta manera, si se pretende averiguar el texto original $p$ habiendo obtenido el texto cifrado, es necesario conocer la clave $k$ para descifrar $c$. La operación de cifrado se puede representar de la siguiente forma:

$$c = k(p). \tag{1}$$

Para transformar $c$ en una posible clave de $K$ mediante el uso de la función $R$, se define la siguiente operación:

$$f(k) = R[c]. \tag{2}$$

### 2.2.2. Generación de una 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 $1, \ldots, m$. Se comenzará tomando un punto de partida, el cual será una clave aleatoria del conjunto de claves $K$. A este punto inicial se le denominará $p_{i_0}$, al cual le será aplicado la función $f$. Una vez hecho esto, se habrá obtenido un nuevo punto en la tabla $p_{i_1}$, diferente a la anterior por requerimientos de la función de reducción $R$. Este proceso será repetido un total de $t$ veces en cada fila, generando por último un punto final $p_{i_t}$, llegando a obtener al final la tabla indicada de $m$ filas y $t$ columnas. Una representación visual de la generación de la tabla podría ser la siguiente:

$$
\begin{array}{ccccccccc}
p_{1_1} & \xrightarrow{f} & p_{1_2} & \xrightarrow{f} & p_{1_3} & \xrightarrow{f} & \cdots & \xrightarrow{f} & p_{1_t} \\
p_{2_1} & \xrightarrow{f} & p_{2_2} & \xrightarrow{f} & p_{2_3} & \xrightarrow{f} & \cdots & \xrightarrow{f} & p_{2_t} \\
p_{3_1} & \xrightarrow{f} & p_{3_2} & \xrightarrow{f} & p_{3_3} & \xrightarrow{f} & \cdots & \xrightarrow{f} & p_{3_t} \\
 & & \vdots & & & & \vdots & & \\
p_{m_1} & \xrightarrow{f} & p_{m_2} & \xrightarrow{f} & p_{m_3} & \xrightarrow{f} & \cdots & \xrightarrow{f} & p_{m_t}
\end{array}
$$

**Figura 2.1:** 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 serán almacenadas la primera y última entradas de cada fila, formando una tupla. De tal forma, tomando el ejemplo anterior, se almacenarían las tuplas $\{p_{1_0}, p_{1_t}\}$, $\{p_{2_0}, p_{2_t}\}$, $\{p_{3_0}, p_{3_t}\}$, $\ldots$, $\{p_{m_0}, p_{m_t}\}$. Este almacenamiento se hace con el objetivo de ahorrar espacio en memoria. Visto esto, el algoritmo 2.2 corresponde a la generación de las tablas de Hellman para su TMTO.

**Algorithm 2.2** Algoritmo de generación de tablas de Hellman en su TMTO

**Entrada:** tabla de Hellman vacía *tabla*, número de filas $m$, número de columnas $t$, operación $f$ de transformación de claves
**Salida:** tabla de Hellman *tabla* con dimensiones $m \times t$

$i, j = 1$
**while** $i \leq m$ **do**
  $p_{i_j} \leftarrow$ Clave inicial de la fila
  **while** $j \leq t$ **do**
    $p_{i_{j+1}} \leftarrow f(p_{i_j})$
    $j \leftarrow j + 1$
  **end while**
  *tabla*.append($\{p_{i_1}, p_{i_j}\}$)
  $i \leftarrow i + 1$
  $j = 1$
**end while**

### 2.2.3. Empleando tablas para descifrar textos

La tabla generada es empleada durante el ataque para averiguar la clave empleada en el cifrado. Durante este proceso de búsqueda, cada vez que se encuentre una posible clave, se tratará de descifrar el texto cifrado para obtener el texto original. El primer paso es probar con el punto final de la primera fila, $p_{1_t}$ en el caso anterior. Si esta resulta ser la clave deseada el ataque concluye con éxito. De lo contrario, se comprueba el punto final de la siguiente fila, repitiendo este proceso hasta encontrar la clave. En caso de no averiguarla la clave deseada en ninguna de las últimas columnas de la tabla, se le aplica la función $f$ para obtener una nueva clave. Dicha nueva clave se busca de nuevo en las entradas correspondientes a la última columna de la tabla. Este proceso se repite hasta encontrar la clave, en cuyo caso el ataque finaliza con éxito. Si tras aplicar la función $f$ un total de $t$ veces no se ha conseguido encontrar la clave buscada, se concluye la búsqueda de dicha clave sin éxito. Este proceso está representado en el algoritmo 2.3.

**Algorithm 2.3** Algoritmo de búsqueda de claves de Hellman en su TMTO

**Entrada:** tabla de Hellman *tabla* con dimensiones $m \times t$, número de filas $m$, número de columnas $t$, operación $f$ de transformación de claves

$i, j = 1$
**while** $i \leq m$ **do**
  **if** $p_{i_t}$ es la clave buscada **then**
    búsqueda detenida con éxito
  **else**
    $i \leftarrow i + 1$
  **end if**
**end while**
$i = 1$
**while** $i \leq m$ **do**
  $p = p_{i_t}$
  **while** $j \leq t$ **do**
    $p' = f(p)$
    **if** $p'$ es la clave buscada **then**
      búsqueda detenida con éxito
    **else**
      $j \leftarrow j + 1$
      $p = p'$
    **end if**
  **end while**
  $i \leftarrow i + 1$
**end while**

# CAPÍTULO 3
# El ataque del arco iris

## 3.1 Relación con trabajos anteriores

Hasta ahora se han visto diferentes aproximaciones que forman una base fundamental para la búsqueda de colisiones en funciones resumen. Aunque dichas aproximaciones no son empleadas hoy en día, sirvieron como base para el desarrollo del ataque del arco iris, el cual consiguió mejorar las prestaciones de todo lo visto anteriormente, mediante el uso de una estructura de datos que hace uso de los mismos principios que aquellos vistos en la Figura 2.1, expandiendo sobre ellos y resultando en las conocidas como tablas del arco iris.

En 2003, Philippe Oechslin publicó el primer artículo en el cual aparecía el concepto de la tabla del arco iris [7]. Dicha tabla sigue la estructura mostrada en la Figura 2.1, introduciendo algunos cambios, como por ejemplo un aumento del tamaño de la tabla.

## 3.2 Características de las tablas del arco iris

La clave de este ataque reside en el uso de sus tablas, las cuales son capaces de representar mucha más información que las tablas que emplea Hellman. De hecho, el ataque llevado a cabo por Hellman requiere hacer uso de varias de sus tablas, mientras que si se pretende obtener los mismos resultados mediante tablas del arco iris únicamente haría falta emplear una de estas tablas. Las dimensiones de las tablas del arco iris dependerán, en principio, del número total de colisiones diferentes que puedan ser averiguadas, lo cual dependerá de las características de dichos textos. Diferentes factores como el número de caracteres permitidos o el uso o no de diferentes dominios de caracteres (letras minúsculas y/o mayúsculas, números, símbolos . . . ) juegan un papel fundamental para determinar las dimensiones de las tablas del arco iris.

### 3.2.1. Funciones de reconstrucción

Las funciones de reconstrucción representan un aspecto fundamental en cualquier implementación del ataque del arco iris. En resumidas cuentas, estas fun-

ciones son capaces de transformar valores *hash* en palabras o mensajes válidos dentro del contexto de la implementación. El funcionamiento básico de estas funciones requiere conocer dos aspectos principales, los cuales son la función resumen para la cual se buscan las colisiones y el conjunto de textos válidos empleados en el contexto del ataque. Este último aspecto variará dependiendo del ámbito en el que se aplique el ataque. Por ejemplo, un ataque a contraseñas permite únicamente palabras individuales, mientras que un ataque a un sistema de firma electrónica estará tratando con mensajes enteros. Para ejemplificar el comportamiento de una función de reconstrucción $R$, si se pretende atacar una función resumen $h$ que recibe textos $p$ sería como sigue:

$$p_1 \xrightarrow{h} h(p_1) \xrightarrow{R} p_2. \tag{3}$$

La característica más distinguida de las tablas del arco iris, y por la cual reciben su nombre, reside en el uso que son capaces de hacer de la función de reconstrucción, siendo de esta manera el foco de mayor importancia en cuanto a la construcción de tablas del arco iris se refiere la elección apropiada de una o varias funciones de reconstrucción.

### 3.2.2. Generación de tablas del arco iris

Antes de construir una tabla del arco iris, es necesario conocer la función resumen $h$ a atacar y el conjunto de textos válidos $P$ que pueden ser pasados a dicha función resumen. Una vez sabido esto, pretendiendo atacar una función resumen $h$ en búsqueda de colisiones se construirá una tabla del arco iris de $m$ filas y $t$ columnas. Para la construcción de dicha tabla el mismo procedimiento tendrá lugar para cada fila $i$ en el rango $1, \ldots, m$. Comenzando por un punto inicial $p_{i_0} \in P$, su valor *hash* será calculado, resultando en $h(p_{i_0})$. Tras obtener este valor, la función de reconstrucción lo tomará como entrada, produciendo un texto $p_{i_1} \in P$, el cual será distinto al texto anterior por requerimientos de la función de reconstrucción. Para cada fila $i$, este proceso se repetirá $t$ veces, generando por último el valor *hash* correspondiente al último texto $p_{i_t}$. Si se repite este procedimiento para $m$ filas, el resultado es una tabla del arco iris de dimensiones $m \times t$. Si se denomina $f$ al proceso de aplicar a un texto una función resumen seguida de una función de reconstrucción, una representación visual de la generación de la tabla sería la siguiente:

$$
\begin{array}{ccccccccccc}
p_{1_1} & \xrightarrow{h} & h_{1_1} & \xrightarrow{r} & p_{1_2} & \xrightarrow{h} & \dots & \xrightarrow{r} & p_{1_t} & \xrightarrow{h} & h(p_{1_t}) \\
p_{2_1} & \xrightarrow{h} & h_{2_1} & \xrightarrow{r} & p_{2_2} & \xrightarrow{h} & \dots & \xrightarrow{r} & p_{2_t} & \xrightarrow{h} & h(p_{2_t}) \\
p_{3_1} & \xrightarrow{h} & h_{3_1} & \xrightarrow{r} & p_{3_2} & \xrightarrow{h} & \dots & \xrightarrow{r} & p_{3_t} & \xrightarrow{h} & h(p_{3_t}) \\
 & & \vdots & & & & & & \vdots & & \\
p_{m_1} & \xrightarrow{h} & h_{m_1} & \xrightarrow{r} & p_{m_2} & \xrightarrow{h} & \dots & \xrightarrow{r} & p_{m_t} & \xrightarrow{h} & h(p_{m_t})
\end{array}
$$

**Figura 3.1:** Generación esquematizada de una tabla del arco iris

La forma de almacenar estas tablas es la misma que la que empleaba Hellman en su TMTO, ya que se almacenará una tupla con el primer y el último elemento de cada fila, el lugar de la totalidad de la misma. De esta manera, las tuplas almacenadas en el ejemplo de la Figura 3.1 serían $\{p_{1_1},\ h(p_{1_t})\}$, $\{p_{2_1},\ h(p_{2_t})\}$, $\{p_{3_1},\ h(p_{3_t})\}$, $\dots$, $\{p_{m_1},\ h(p_{m_t})\}$.

Puede ocurrir también que durante el proceso de generación de la tabla se generen colisiones internas 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. Estas colisiones pueden también aparecer en las columnas intermedias de la tabla, aunque en este trabajo tan sólo se tendrán en cuenta las halladas en las últimas columnas.

Dicho TMTO ocurre en este caso dependiendo de los valores de $m$ y $t$ que se elijan para generar la tabla. Un mayor número de filas respecto al número de columnas supondrá un menor tiempo de computación para la generación de la tabla, pero requerirá a su vez 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 un mayor número de veces. Además de esto, aumentar en número de columnas de las tablas siempre tendrá como consecuencia elevar la probabilidad de encontrar colisiones internas en las tablas.

### 3.2.3. Búsqueda de colisiones mediante tablas del arco iris

Encontrar el valor *hash* que se está buscando en una entrada de una tabla del arco iris significa que la entrada anterior contiene un texto que genera dicho *hash*. Teniendo disponible una tabla del arco iris, si se desea encontrar colisiones para una función resumen $h$ se sigue un proceso similar al visto en las tablas de Hellman. Mirando primero al segundo valor de la primera tupla, si esta valor *hash* coincide con el que se busca, se sabe entonces que la entrada anterior en la

tabla contiene el texto que lo genera. De lo contrario, para cada fila, se comienza desde la última entrada de la misma, la cual contiene un valor resumen, y se le aplica la reconstrucción para a continuación obtener el *hash* correspondiente, comprobando si es una colisión del valor *hash* deseado. Esto se repite un máximo de $t$ veces para cada fila, tal y como se muestra en el algoritmo 3.1.

**Algorithm 3.1** Algoritmo de búsqueda de colisiones en tablas del arco iris

**Entrada:** tabla del arco iris $rt$ con dimensiones $m \times t$, número de filas $m$, número de columnas $t$, función resumen $H$, función de reconstrucción $R$, valor *hash* del cual se busca la colisión $h$

$i, j = 1$
**while** $i \leq m$ **do**
  **if** $rt[i,t] == h$ **then**
    búsqueda finalizada con éxito
  **else**
    $i \leftarrow i + 1$
  **end if**
**end while**
$i = 1$
**while** $i \leq m$ **do**
  $p = rt[i,t]$
  **while** $j \leq t$ **do**
    $p' = R(p)$
    $p'' = H(p')$
    **if** $p'' == h$ **then**
      búsqueda finalizada con éxito
    **else**
      $j \leftarrow j + 1$
      $p = p''$
    **end if**
  **end while**
  $i \leftarrow i + 1$
**end while**

### 3.2.4. Escalabilidad

Otra de las ventajas del ataque del arco iris respecto a otras implementaciones es su escalabilidad, la cual es observable mediante diferentes despliegues del ataque. En concreto, en la búsqueda de colisiones en funciones *hash*, si se consigue implementar el ataque contra una función resumen que produce valores *hash* de $n$ bits, obteniendo unos resultados calificados como buenos, al cambiar la función resumen por una que genere valores *hash* de $2n$ bits, la calidad de los resultados apenas varía respecto a los anteriores. El único aspecto que se ve afectado será el tiempo empleado en llevar a cabo el ataque, el cual aumentará en contextos más complejos.

## 3.3 Puntos distinguidos

Una variante habitual del ataque del arco iris es el método de los puntos de control o puntos distinguidos, ideado por Rivest [8]. Esta aproximación busca reducir el número de colisiones internas de las tablas de forma significativa, para de esta manera aumentar la información disponible en la misma.

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, siendo habitual buscar un cierto patrón en los valores *hash* que se van obteniendo en la generación de las filas de la tabla (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, lo cual puede darse en una entrada cercana a la inicial o puede tardar más de lo previsto en ocurrir. 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.

## 3.4 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, teniendo lugar mayormente cuando la función resumen la cual se pretende atacar hace uso de los siguientes métodos:

- ***Salting***: la técnica conocida como *salting* consiste en añadir una cadena aleatoria conocida como *salt* a un mensaje antes de ser resumido. Dicha cadena *salt* es privada y de igual longitud que el mensaje. Por ejemplo, si se partiera con una contraseña $p$, el *salt* $s$ y la operación de disyunción exclusiva $\oplus$, sería posible obtener como valores *hash* tal que $h = HASH(p \oplus s)$. 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.

- ***Peppering***: puede considerarse similar al *salting*, aunque existen diferencias entre ambas técnicas. En el caso del *peppering*, se concatena una cadena de longitud pequeña (generalmente 4 bits) al mensaje que va a ser resumido, la cual se conoce como *pepper*. Esta cadena se elige generalmente de forma aleatoria al resumir el mensaje. De esta manera las probabilidades de

encontrar una colisión para una función *hash* se ven reducidas de manera significativa.

- **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 es fácilmente observable puede resultar en la inviabilidad del ataque.

Cabe destacar que el *salting* y el estiramiento ocurren por lo general en sistemas de almacenamiento de contraseñas protegidas por funciones resumen. En este contexto, la aparición de estas técnicas consiguió dejar prácticamente inútil al ataque del arco iris en este contexto, aunque incluso después de dicha aparición, los sistemas que no hacen uso del *salting*, el *peppering* o el estiramiento siguen siendo vulnerables a este ataque. Un ejemplo muy notorio pueden ser algunas versiones del sistema operativo *Windows* de la empresa Microsoft, los cuales hasta el sistema *Windows* 7 (incluido) han sido atacados con éxito por tablas del arco iris. De hecho, en su artículo original, Philippe Oechslin consigue con éxito romper el 100 % de las contraseñas almacenadas en los sistemas *Windows* contra los que lanza su ataque [7]. Esto puede servir como muestra de la relevancia que llegó a alcanzar el ataque del arco iris.

# CAPÍTULO 4
# Estudio experimental

Habiendo expuesto el funcionamiento del ataque del arco iris y sus características más destacadas a tener en cuenta para su implementación, es el momento de detallar el desarrollo llevado a cabo en este proyecto. Dicho desarrollo se centra en la búsqueda de colisiones para funciones resumen, con el objetivo de obtener una implementación capaz de encontrar dichas colisiones de una manera eficiente. Para representar la búsqueda de colisiones de una forma clara, se simulará un ataque a las contraseñas de acceso a un sistema operativo, siendo el objetivo tratar de averiguar el máximo número de contraseñas posibles.

En el uso de contraseñas de acceso a un sistema operativo, las contraseñas como tal no son almacenadas, sino sus valores *hash*, generados por la función resumen elegida para protegerlas. De tal forma, cuando un usuario introduce su contraseña para acceder al sistema, se calcula su valor *hash* y, en caso de coincidir con el valor almacenado, se garantiza el acceso. De esta manera, cualquier sistema no resistente a colisiones, como se ha visto en el apartado 1.2.2, será vulnerable frente al ataque del arco iris. Esto es debido a que si se encuentra una colisión para el valor *hash* almacenado en el sistema, es posible acceder a él de forma no autorizada. Este es un ejemplo de la gran variedad de contextos que existen para los cuales resulta de interés encontrar colisiones para funciones resumen.

En primer lugar, se han de determinar dos aspectos fundamentales para el correcto funcionamiento de la experimentación: el dominio de contraseñas a atacar y la función *hash* que tratará de protegerlas. Aunque en un primer momento parezcan elecciones débiles, existe un motivo de peso para que así sea. Este desarrollo servirá para representar una implementación base, la cual es perfectamente escalable a dominios de contraseñas mayores y funciones *hash* más complejas. En el caso de una correcta implementación, el único aspecto que se verá afectado de forma más significativa sería el tiempo requerido para almacenar la información necesaria para llevar a cabo el ataque, así como la memoria necesaria para almacenar las tablas del arco iris, aunque la calidad de los resultados debería mantenerse idealmente similar. De esta manera, debido a que será necesario experimentar con diversos tamaños de tabla y configuraciones de las mismas, los tiempos empleados y la memoria ocupada en disco en dicha experimentación serán mínimos, agilizando todo el proceso.

## 4.1 Especificaciones

### 4.1.1. Dominio de contraseñas atacado

Un dominio de contraseñas puede ser descrito como todo el rango de posibles contraseñas válidas. En este caso, serán aquellas contraseñas que pueden emplearse para acceder al sistema operativo. Los factores que determinan un dominio son, fundamentalmente, la longitud de las contraseñas y los caracteres permitidos. En la implementación aquí desarrollada, las contraseñas tendrán una longitud exacta de seis caracteres, los cuales únicamente podrán ser números. Como resultado, el número total de contraseñas es de un millón, las cuales se encuentran en el rango $'000000', \ldots, '999999'$.

### 4.1.2. Función resumen atacada

Para este proyecto se ha escogido atacar valores *hash* 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. Contra una función resumen que genere valores *hash* de mayor longitud o que emplee más caracteres además de números, la calidad de los resultados debería ser similar, debido a la escalabilidad del ataque del arco iris.

## 4.2 Aproximación original

### 4.2.1. Dimensiones de las 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 2.2 al introducir el intercambio tiempo-memoria (TMTO), 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 en este caso dependerá de el número total de contraseñas. Recuérdese que el tamaño de una tabla en el ataque 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 TMTO de Hellman es que así como en el TMTO 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:

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

Teniendo esto en cuenta, se procederá a generar diferentes tablas de tamaños diversos, la mayoría cumpliendo con la relación establecida en la Fórmula 1, aunque algunas de las tablas tendrán tamaños por encima de $N$, para explorar más posibilidades y observar la eficiencia de las tablas con diferentes tamaños.

Aunque inicialmente existen muchos tamaños de tabla, los cuales cumplen con la Fórmula Fórmula 1, el objetivo será determinar a partir de qué tamaños se obtienen resultados calificados como buenos, para así poner el foco sobre aquellos tamaños para los cuales no se haya podido romper tantas contraseñas como se deseaba en un principio. De esta forma, se pretende averiguar los tamaños de tabla más eficientes que consigan encontrar el máximo número de colisiones para la función CRC-32. En este caso la eficiencia de una tabla viene medida en términos de memoria necesaria para su almacenamiento y tiempo requerido para su generación y búsqueda de contraseñas.

### 4.2.2. Función de reconstrucción

Queda todavía por especificar la función de reconstrucción a emplear. Cabe recordar que una función de reconstrucción toma como entrada una valor *hash*, y produce como salida un texto válido del dominio empleado. En este caso, la función de reconstrucción producirá como salida una contraseña dentro del dominio $N$. Inicialmente la función de reconstrucción elegida, la cual pasará a denominarse **R1** desde este instante, actuaba de manera muy eficiente. Su comportamiento se ve a continuación, haciendo uso un valor *hash* $h$ y su correspondiente reconstrucción $r$:

- tomando $h$ en forma numérica decimal, esta función le aplicará la operación módulo un millón, con el objetivo de obtener los seis últimos números de $h$. Partiendo desde una contraseña $p = '112233'$, y su correspondiente valor *hash* $h = '3570655599'$, la función de reconstrucción **R1** sigue el siguiente proceso:

  1. Una vez obtenido el valor *hash* $h = '3570655599'$, se le aplicará la operación módulo un millón, con el objetivo de obtener los últimos seis dígitos del *hash*.
  2. La reconstrucción resultante será $r = '655599'$, la cual pertenece al dominio de contraseñas $N$.

### 4.2.3. Generación y uso de tablas del arco iris

Una vez se ha especificado el dominio de contraseñas, la función *hash* a atacar y la función de reconstrucción a emplear, es momento de establecer el método de generación de las tablas del arco iris. Para ello, inicialmente será necesario elegir las dimensiones $m \times t$ de la tabla a generar, y posteriormente, como se ha visto en

la Figura 3.1, una vez determinados los parámetros $m$ (filas) y $t$ (columnas), los pasos a seguir en este caso serán los mostrados en el algoritmo 4.1.

**Algorithm 4.1** Algoritmo de generación de tablas del arco iris

**Entrada:** tabla del arco iris vacía $rt$, número de filas $m$, número de columnas $t$, función resumen $H$, función de reconstrucción $R$
**Salida:** tabla del arco iris $rt$ con dimensiones $m \times t$

```
i, j = 0
while i < m do
    p ← Contraseña inicial de la fila
    h ← H(p)
    while j < t do
        r ← R(h)
        h ← H(r)
        j ← j + 1
    end while
    i ← i + 1
    rt.append({p, h})
end while
```

Como muestra la construcción de la tabla, y de acuerdo con la propuesta de Oechslin, la tabla del arco iris resultante será almacenada en memoria, aunque no con todas las entradas generadas en su construcción. 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 la primera y la última columna de cada fila obtenidas 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 en cada fila para volver a generarla en su totalidad siguiendo el proceso descrito en el algoritmo 4.1

Cabe destacar que las contraseñas iniciales para cada fila se obtendrán del dominio en orden ascendente. De esta manera, la contraseña inicial para la primera fila de la tabla siempre será $p_{1_0}$ ='000000', mientras que la contraseña inicial para la segunda fila de la tabla será $p_{2_0}$ ='000001', y así sucesivamente.

Recuérdese que se está tratando de averiguar colisiones para la función resumen atacada. Al elegir representar este ataque mediante un sistema del almacenamiento de contraseñas en forma de sus valores *hash*, el éxito del ataque dependerá de la cantidad de contraseñas cuyos valores resumen son averiguados. A la hora de buscar colisiones 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 se puede concluir que es posible obtener su contraseña correspondiente, la cual se encontrará en la entrada anterior. En caso de no encontrar el valor *hash* en una de las últimas entradas de la tabla, se realizará el proceso de búsqueda tal y como ha sido mostrado en el algoritmo 3.1. Si en algún momento durante esta búsqueda se da con el valor *hash* del cual se pretende encontrar

una colisión, se concluye la búsqueda con éxito. El caso opuesto significará que la tabla no ha sido capaz de dar con la colisión deseada.

En cuanto a el tamaño de la tabla, puede intuirse que cuantas más filas posea, mayor probabilidad existirá de romper una contraseña sin necesidad de buscar más allá de las últimas entradas de la tabla, 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 buscando en dichas entradas, aunque aumentará a su vez la probabilidad de generar colisiones internas en la tabla. 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 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.

### 4.2.4. La estructura de datos para representar la tabla

Antes de proceder a las pruebas de este primer lote de tablas del arco iris, queda detallar el aspecto final de la implementación del ataque del arco iris. En cuanto han sido determinados todos los parámetros que configurarán las tablas del arco iris, únicamente falta elegir la estructura de datos que será empleada para representarlas, la cual queda a elección del atacante. A la hora de optar por una estructura de datos u otra, se ha de tener en cuenta, como ya ha sido mencionado anteriormente, 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 2.1 como ejemplo, la lista de tuplas que equivaldría a esta tabla del arco iris correspondería con la siguiente representación: $[(p_{1_0}, h_{1_t}), (p_{2_0}, h_{2_t}), (p_{3_0}, h_{3_t}), \ \ldots, (p_{m_0}, h_{m_t})]$.

- **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 2.1, el diccionario resultante que representaría a la tabla del arco iris correspondiente sería el siguiente: $\{p_{1_0} : h_{1_t}, \ p_{2_0} : h_{2_t}, \ p_{3_0} : h_{3_t}, \ \ldots, \ p_{m_0} : h_{m_t}\}$.

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 su generación y almacenamiento en disco, para así tenerlas preparadas para llevar a cabo el ataque contra las contraseñas de acceso al sistema, las cuales hay que generar también.

### 4.2.5. Generación de corpus de test

Para simular las contraseñas almacenadas en un sistema operativo en forma de valores *hash*, se generarán un total de 1.000 valores *hash* aleatorios. Para su obtención se escogerá de forma aleatoria 1.000 contraseñas contenidas en el dominio establecido ($N$), generando el valor *hash* correspondiente de cada una y almacenándolo en una lista. Dicha lista será almacenada en disco 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.

### 4.2.6. Primeras pruebas

Una vez generadas todas las tablas del arco iris y las contraseñas a atacar, es el momento realizar la experimentación con dichas tablas. A cada tabla empleada en el ataque del arco iris a las contraseñas se le asigna un porcentaje de éxito tras experimentar con ella. Dicho porcentaje de éxito para cada tabla se mide dividiendo el número de colisiones que ha sido capaz de encontrar entre el número

total de colisiones a averiguar, 1.000 en este caso debido al dominio $N$. Una vez concluida la experimentación con una tabla del arco iris, le será asignada su porcentaje de éxito correspondiente, el cual a su vez estará asociado a un color. Para todos los resultados se empleará el mismo código de color indicando la calidad de los mismos:

- 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 encontrar las colisiones deseadas con una probabilidad de éxito elevada. Para esta experimentación en particular, encontrar dichas colisiones corresponderá con adivinar las contraseñas almacenadas en el sistema. Si bien esto es posible en este caso para el dominio elegido, si se expandiese dicho dominio resultaría más complicado, por lo cual si una tabla consigue encontrar al menos 950 colisiones de las 1.000 generadas aleatoriamente será clasificada como idónea. Así mismo, una tabla que no llegue a descubrir tantas colisiones pero consiga al menos encontrar 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 averiguar 700 colisiones puede considerarse que se han quedado cerca de ser aceptables, mientras que una tabla que ni siquiera consiga hacerse con 700 colisiones debería de ser descartada de inmediato, ya que tendría una probabilidad de encontrar una colisión similar a sacar cara o cruz tirando una moneda al aire.

Antes de pasar a los resultados obtenidos en esta primera tanda de pruebas, cabe recordar que las dimensiones de las tablas que se van a ver servirán para determinar en qué tamaños de tabla habrá que centrarse en la búsqueda de las tablas del arco iris más eficientes. La eficiencia en este caso interesa desde el punto de vista del tiempo empleado en la generación de la tabla y la búsqueda de colisiones, así como la memoria necesaria para almacenar dicha tabla en el sistema.

Los resultados de la primera prueba pasan a verse a continuación, con las tablas que emplean la función de reconstrucción **R1**:

| $t$ | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | | | | | | | | | | | | | 100 % |
| | 2 | | | | | | | | | | | | 91.9 % | |
| | 4 | | | | | | | | | | | 90.6 % | | |
| | 5 | | | | | | | | | | 90.6 % | | | |
| | 10 | | | | | | | | | 95.2 % | | | | |
| | 20 | | | | | | | | 96.1 % | | | | | |
| | 40 | | | | | | | 97.1 % | | | | | | |
| | 50 | | | | | | 97.4 % | | 99.3 % | 99.6 % | | 99.6 % | 100 % | 100 % |
| | 100 | | | | | 96.4 % | 98.0 % | 98.4 % | 99.0 % | 99.8 % | | 100 % | 100 % | 100 % |
| | 200 | | | | 95.2 % | | | | | | | | | |
| | 400 | | | 91.8 % | | | | | | | | | | |
| | 500 | | 84.1 % | | | | | | | | | | | |
| | 1000 | 63.5 % | | | | | | | | | | | | |
| | | 1000 | 2000 | 2500 | 5000 | 10000 | 20000 | 25000 | 50000 | 100000 | 200000 | 250000 | 500000 | 1000000 |
| | | $m$ | | | | | | | | | | | | |

**Tabla 4.1:** Porcentajes de éxito para las tablas empleando **R1**

Puede observarse que las tablas que poseen un número de filas igual o superior a 2.500 obtienen buenos resultados. Tan sólo una de las tablas sería descartada, aquella con dimensiones $1.000 \times 1.000$, mientras que después de esa solamente hay otra que no llega a ser aceptable, aunque se queda realmente cerca de conseguirlo. Se ha llegado 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 $250.000 \times 100 = 25.000.000 > 1.000.000 = N$. Estos tamaños de tabla superiores al dominio se escogieron para tratar de llegar al límite del tamaño de la misma, ya que cuanto más se exceda el tamaño mayores serán las probabilidades de generar colisiones internas en la tabla. 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 $200.000 \times 5$ tiene menor porcentaje de éxito que la tabla con dimensiones $20.000 \times 50$, teniendo un 10 % de las filas, pero 10 veces la profundidad, lo cual resulta clave para su mejor resultado.

Los resultados obtenidos pueden servir para confirmar que el desarrollo llevado a cabe de la implementación del ataque del arco iris ha sido correcto, ya que la mayoría de las tablas generadas brindan excelentes resultados. Aún así, resultaría interesante experimentar con una función de reconstrucción diferente, con el objetivo de mejorar los resultados.

## 4.3 Una nueva función de reconstrucción

Así como la función de reconstrucción empleada en la sección anterior (**R1**) actúa sobre el valor numérico decimal del valor *hash*generado por la función CRC-32, esta nueva función de reconstrucción, la cual pasará a denominarse **R2**, actua-

rá sobre el valor en hexadecimal del *hash*, aunque también hace uso de su valor decimal. Al igual que ya se hizo con **R1**, a continuación se mostrará el comportamiento de la nueva función de reconstrucción **R2**, haciendo uso de un valor *hash* $h$ y su correspondiente reconstrucción $r$:

- tomando $h$ en forma hexadecimal, esta función inicialmente, mientras el valor resumen $h$ tenga un número de caracteres mayor que seis, irá eliminando el primero de estos caracteres, hasta llegar a un valor $h$ correspondiente con 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 decimal, comenzando desde el final, y tras cada sustitución se tomará el número anterior. Partiendo desde una contraseña $p = $ '112233', y su correspondiente valor *hash* hexadecimal $h = $ 'D4D3E16F', la función de reconstrucción **R2** sigue el siguiente proceso:

  1. Obtener primero los valores *hash* en forma numérica decimal $h_{num} =$ '3570655599' y en forma hexadecimal $h_{hex} =$ 'D4D3E16F'

  2. Mientras la longitud de $h_{hex}$ sea mayor que seis, eliminar su primer carácter, resultando finalmente en $h_{hex} =$ 'D3E16F'

  3. 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 especificado el comportamiento de esta nueva función de reconstrucción, de nuevo es momento de generar tablas del arco iris que hagan uso de ella, para tratar de atacar las mismas contraseñas que las tablas anteriores.

### 4.3.1. Nuevos resultados

En esta ocasión, el tamaño de las tablas generadas será el mismo que en el apartado 4.2.6, para así poder comprobar de manera sencilla la diferencia entre ambas implementaciones. Los resultados obtenidos con la función de reconstrucción **R2** son los siguientes:

| $t$ | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | | | | | | | | | | | | | 100 % |
| | 2 | | | | | | | | | | | | 91.1 % | |
| | 4 | | | | | | | | | | | 89.4 % | | |
| | 5 | | | | | | | | | | 88.8 % | | | |
| | 10 | | | | | | | | | 93.0 % | | | | |
| | 20 | | | | | | | | 95.5 % | | | | | |
| | 40 | | | | | | | 97.1 % | | | | | | |
| | 50 | | | | | | 97.1 % | | 98.8 % | 99.7 % | | 100 % | 100 % | 100 % |
| | 100 | | | | | 97.1 % | 98.8 % | 99.2 % | 99.6 % | 99.9 % | | 99.9 % | 99.9 % | 100 % |
| | 200 | | | | 94.3 % | | | | | | | | | |
| | 400 | | | 82.1 % | | | | | | | | | | |
| | 500 | | 74.9 % | | | | | | | | | | | |
| | 1000 | 57.1 % | | | | | | | | | | | | |
| | | 1000 | 2000 | 2500 | 5000 | 10000 | 20000 | 25000 | 50000 | 100000 | 200000 | 250000 | 500000 | 1000000 |
| | | $m$ | | | | | | | | | | | | |

**Tabla 4.2:** Porcentajes de éxito para las tablas empleando **R2**

Observando en un primer vistazo los datos que aparecen en la Tabla 4.2, se puede apreciar que los resultados aquí obtenidos son peores que los anteriores en la tabla 4.1 con la función de reconstrucción **R1**. La tabla del arco iris que ya se esperaba que fuera a ser descartada con $m = 1.000$ 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 buenos, ya que hay múltiples tablas con un porcentaje de éxito mayor al 95 %.

Hasta ahora tan sólo se ha hecho uso de una función de reconstrucción tanto para la generación como para la búsqueda en las tablas del arco iris. Una vez visto esto, se va a estudiar otros posibles usos de estas funciones de reconstrucción con el objetivo de obtener mejores resultados, explotando esta característica fundamental de las tablas del arco iris.

## 4.4 Combinando funciones de reconstrucción

### 4.4.1. Alternancia de funciones de reconstrucción

Habiendo ya empleado ambas funciones de reconstrucción, **R1** y **R2**, la forma más intuitiva de combinarlas en una misma tabla del arco iris consiste en alternar su uso. De esta manera, para cada fila de la tabla, para la primera entrada o columna la utilizará la función de reconstrucción **R1**, mientras que para la entrada siguiente se hará uso de la función **R2**, y así sucesivamente. Tomando como referencia la Figura 2.1, la construcción de cada fila de las tablas del arco iris configuradas con la alternancia de funciones de reconstrucción corresponderá con el siguiente esquema, para una columna $c$:

$$p_{1_c} \xrightarrow{CRC-32} h_{1_c} \xrightarrow{\mathbf{R1}} r_{1_c} \xrightarrow{CRC-32} h_{2_c} \xrightarrow{\mathbf{R2}} r_{2_c} \xrightarrow{CRC-32} h_{3_c} \ldots$$

**Figura 4.1:** Generación de una fila alternando funciones de reconstrucción

Siguiendo este método para la construcción de filas, el algoritmo 4.2 indica el proceso al completo de la generación de una tabla del arco iris que hace uso de la alternancia de las funciones de reconstrucción.

**Algorithm 4.2** Algoritmo de generación de tablas del arco iris que alternan las funciones de reconstrucción

**Entrada:** tabla del arco iris vacía $rt$, número de filas $m$, número de columnas $t$, función resumen $H$, función de reconstrucción $R1$, función de reconstrucción $R2$
**Salida:** tabla del arco iris $rt$ con dimensiones $m \times t$

```
i, j = 1
while i ≤ m do
    p ← Contraseña inicial de la fila
    h ← H(p)
    while j ≤ t do
        if j es impar then
            r ← R1(h)
        else
            r ← R2(h)
        end if
        h ← H(r)
        j ← j + 1
    end while
    i ← i + 1
    rt.append({p, h})
end while
```

Este nuevo uso de las funciones de reconstrucción también afecta a la búsqueda de colisiones. En esta ocasión, al igual que anteriormente, si tras comparar el valor *hash* almacenado con las últimas columnas de la tabla no se ha conseguido encontrar una colisión, se deberá de proceder al proceso de búsqueda, el cual será algo diferente en este caso, tal y como se aprecia en el algoritmo 4.3.

**Algorithm 4.3** Algoritmo de búsqueda de colisiones en tablas del arco iris que alternan las funciones de reconstrucción

**Entrada:** tabla del arco iris $rt$ con dimensiones $m \times t$, número de filas $m$, número de columnas $t$, función resumen $H$, valor *hash* del cual se busca la colisión $h$, función de reconstrucción $R1$, función de reconstrucción $R2$

```
if ∃ una entrada {ini, h} then
  búsqueda finalizada con éxito, devolver la colisión a partir de ini
end if
i = 1
while i ≤ longitud de la secuencia R1-R2 do
  r ← i-ésima función de reconstrucción en R1-R2
  for j = 1 hasta t do
    h ← H(r(h))
    if ∃ una entrada {ini, h} then
      búsqueda finalizada con éxito, devolver la colisión a partir de ini
    else
      r ← siguiente función de reconstrucción en la secuencia R1-R2
    end if
  end for
  i ← i + 1
end while
Devolver error en la búsqueda
```

El motivo por el cual puede llegar a iniciarse la búsqueda de nuevo para la misma fila es debido a la combinación de ambas funciones de reconstrucción, ya que al tratar de encontrar una colisión determinada, se desconoce qué función de reconstrucción ha sido empleada previo a su obtención, en el caso de que exista en la tabla. De esta manera, es importante realizar la búsqueda de ambas formas, primero empleando la alternancia de funciones de reconstrucción **R1-R2**, y si tras finalizar esta búsqueda no se ha encontrado la colisión del valor resumen se repite con la alternancia **R2-R1**.

Tras ver cómo estas tablas realizarán el proceso de búsqueda de colisiones para la función resumen, es momento de experimentar con ellas como se ha hecho en los casos anteriores. El rendimiento de estas tablas en la búsqueda de colisiones es el siguiente:

| $t$ | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | | | | | | | | | | | | | 100 % |
| | 2 | | | | | | | | | | | | 97.0 % | |
| | 4 | | | | | | | | | | | 96.2 % | | |
| | 5 | | | | | | | | | | 97.0 % | | | |
| | 10 | | | | | | | | | 98.3 % | | | | |
| | 20 | | | | | | | | 99.5 % | | | | | |
| | 40 | | | | | | | 99.7 % | | | | | | |
| | 50 | | | | | | 99.6 % | | 100 % | 100 % | | 100 % | 100 % | 100 % |
| | 100 | | | | | 99.9 % | 99.9 % | 99.9 % | 100 % | 100 % | | 100 % | 100 % | 100 % |
| | 200 | | | | 99.9 % | | | | | | | | | |
| | 400 | | | 98.1 % | | | | | | | | | | |
| | 500 | | 95.4 % | | | | | | | | | | | |
| | 1000 | 63.1 % | | | | | | | | | | | | |
| | | 1000 | 2000 | 2500 | 5000 | 10000 | 20000 | 25000 | 50000 | 100000 | 200000 | 250000 | 500000 | 1000000 |
| | | $m$ | | | | | | | | | | | | |

**Tabla 4.3:** Porcentajes de éxito para las tablas empleando la alternancia de las funciones de reconstrucción

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 encontrar todas las colisiones. Entrando en detalle en las tablas perfectas, además de las tablas más grandes de las cuales ya se esperaban estos resultados, se puede observar que la tabla con dimensiones $50.000 \times 50$ 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.

De esta manera ha quedado reflejada la utilidad de poder combinar diferentes funciones de reconstrucción en una misma tabla del arco iris. Teniendo en cuenta que esta combinación de ambas funciones de reconstrucción es la más simple que se puede emplear, resultaría interesante saber si una combinación más intrincada proporcionaría mejores resultados.

### 4.4.2. Combinando las funciones de reconstrucción en un patrón

Siguiendo el mismo comportamiento que la combinación anterior, surge una nueva con un ligero cambio. En esta ocasión, en lugar de alternar ambas funciones, se establecerá un patrón cíclico, el cual seguirá el siguiente comportamiento: 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.

Esta aproximación de combinar las funciones de reconstrucción en un patrón consiste en una aportación original al ataque del arco iris, ya que las anteriores configuraciones de funciones de reconstrucción ya habían sido empleadas anteriormente, tanto en el artículo original de Oechslin [7] como en otras aproximaciones posteriores a dicho artículo. De esta manera se pretende explorar aproximaciones no vistas anteriormente.

Tomando el esquema de la Figura 4.1, haciendo uso de nuevo de una columna $c$, el método de generación de una fila para las tablas del arco iris que sigan esta configuración será el siguiente:

$$p_{1_c} \xrightarrow{CRC-32} h_{1_c} \xrightarrow{\mathbf{R1}} r_{1_c} \xrightarrow{CRC-32} h_{2_c} \xrightarrow{\mathbf{R2}} r_{2_c} \xrightarrow{CRC-32} h_{3_c} \xrightarrow{\mathbf{R2}} r_{3_c} \ldots$$

**Figura 4.2:** Generación de una fila combinando funciones de reconstrucción mediante un patrón reducido

Empleando el método visto en la Figura 4.2, la construcción de tablas del arco iris con esta configuración de funciones de reconstrucción se muestra en el algoritmo 4.4.

**Algorithm 4.4** Algoritmo de generación de tablas del arco iris que emplean el patrón reducido de funciones de reconstrucción

**Entrada:** tabla del arco iris vacía $rt$, número de filas $m$, número de columnas $t$, función resumen $H$, función de reconstrucción $R1$, función de reconstrucción $R2$
**Salida:** tabla del arco iris $rt$ con dimensiones $m \times t$

```
i, j, ptr = 1
while i ≤ m do
  p ← Contraseña inicial de la fila
  h ← H(p)
  while j ≤ t do
    if ptr == 1 then
      r ← R1(h)
    else
      r ← R2(h)
    end if
    h ← H(r)
    j ← j + 1
    if ptr == 3 then
      ptr = 1
    else
      ptr ← ptr + 1
    end if
  end while
  i ← i + 1
  rt.append({p, h})
end while
```

Al igual que ocurre con las tablas que alternan ambas funciones de reconstrucción, las tablas que hagan uso de este patrón de funciones de reconstrucción también deberán variar su comportamiento en la búsqueda de colisiones. En esta ocasión, la primera vez que se busque en una fila una colisión determinada, se empleará el patrón al igual que en la Figura 4.2. Si se alcanza la profundidad $t$ sin encontrar la colisión deseada, se comenzará la búsqueda de nuevo con un desplazamiento en el patrón. En lugar de hacer uso del patrón **R1, R2, R2**, en esta segunda iteración se comenzará empleando la función de reconstrucción **R2** para las dos primeras reconstrucciones, siendo la función **R1** empleada para la tercera, repitiendo este patrón un total de $t$ veces mientras no se encuentre la colisión. Si de nuevo no se ha encontrado dicha colisión buscada, se realizará el proceso de búsqueda por vez final, esta vez empleando el patrón **R2, R1, R2**. El motivo por el cual la búsqueda se desarrolla de esta manera es el mismo que en el caso de las tablas que alternan ambas funciones, para compensar el hecho de que se desconoce qué función de reconstrucción se utilizó en el paso anterior a obtener la colisión, en el caso de que esta exista en la tabla. El algoritmo 4.5 muestra el proceso de búsqueda de colisiones para las tablas que hacen uso de este patrón reducido

**Algorithm 4.5** Algoritmo de búsqueda de colisiones en tablas del arco iris que emplean el patrón reducido de funciones de reconstrucción

**Entrada:** tabla del arco iris $rt$ con dimensiones $m \times t$, número de filas $m$, número de columnas $t$, función resumen $H$, valor *hash* del cual se busca la colisión $h$, función de reconstrucción $R1$, función de reconstrucción $R2$

```
if ∃ una entrada {ini, h} then
  búsqueda finalizada con éxito, devolver la colisión a partir de ini
end if
i = 1
while i ≤ longitud de la secuencia R1-R2-R2 do
  r ← i-ésima función de reconstrucción en R1-R2-R2
  for j = 1 hasta t do
    h ← H(r(h))
    if ∃ una entrada {ini, h} then
      búsqueda finalizada con éxito, devolver la colisión a partir de ini
    else
      r ← siguiente función de reconstrucción en la secuencia R1-R2-R2
    end if
  end for
  i ← i + 1
end while
Devolver error en la búsqueda
```

El ataque del arco iris llevado a cabo con las tablas que hacen uso de este patrón de reconstrucción obtienen los siguientes resultados:

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | | | | | | | | | | | | | 100 % |
| | 2 | | | | | | | | | | | | 98.1 % | |
| | 4 | | | | | | | | | | | 98.9 % | | |
| | 5 | | | | | | | | | | 99.4 % | | | |
| | 10 | | | | | | | | | 99.8 % | | | | |
| | 20 | | | | | | | | 100 % | | | | | |
| $t$ | 40 | | | | | | | 100 % | | | | | | |
| | 50 | | | | | | 100 % | | 100 % | 100 % | | 100 % | 100 % | 100 % |
| | 100 | | | | | 100 % | 100 % | 100 % | 100 % | 100 % | | 100 % | 100 % | 100 % |
| | 200 | | | | 100 % | | | | | | | | | |
| | 400 | | | 99.5 % | | | | | | | | | | |
| | 500 | | 99.3 % | | | | | | | | | | | |
| | 1000 | 80.5 % | | | | | | | | | | | | |
| | | 1000 | 2000 | 2500 | 5000 | 10000 | 20000 | 25000 | 50000 | 100000 | 200000 | 250000 | 500000 | 1000000 |
| | | $m$ | | | | | | | | | | | | |

**Tabla 4.4:** Porcentajes de éxito para las tablas empleando el patrón reducido

Los resultados obtenidos han conseguido superar a los de la alternancia de funciones de reconstrucción, en la Tabla 4.3. 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 significativamente cerca de ser considerada aceptable. La mejora de los resultados es debida a la mayor complejidad de este patrón con respecto a la alternancia de funciones. Existen muchas más tablas que han conseguido encontrar todas las colisiones, y si se comparan todas las tablas respecto a los resultados anteriores, se puede apreciar que cada una de ellas rinde mejor en este caso. En definitiva, el nuevo patrón ha proporcionado la mejora que se esperaba.

Tras observar estos resultados, cabe la posibilidad de mejora, la cual podría darse tratando de seguir el mismo esquema que el patrón empleado, aunque, al igual que se ha hecho en este patrón respecto a la alternancia de funciones de reconstrucción, será necesario un aumento en la complejidad del mismo.

### 4.4.3. Un patrón más extenso

En esta ocasión se va a emplear un método similar al anteriormente expuesto, aunque se va a complicar un poco más. En lugar de emplear un patrón que se repite cada tres pasos, la longitud del mismo casi se triplicará. Al igual que en el anterior patrón, se comenzará empleado **R1** para la primera entrada, mientras que para las siguientes dos entradas se hará uso de la función de reconstrucción **R2**. Tras estos tres primeros pasos, los dos siguientes emplearán la función de reconstrucción **R1**. Por último, para los tres pasos finales se utiliza de nuevo **R2**. De nuevo, para una columna $c$ de la tabla, el método de generación de una fila corresponderá con el siguiente esquema:

$$p_{1_c} \xrightarrow{CRC-32} h_{1_c} \xrightarrow{\mathbf{R1}} r_{1_c} \xrightarrow{CRC-32} h_{2_c} \xrightarrow{\mathbf{R2}} r_{2_c} \xrightarrow{CRC-32} h_{3_c} \xrightarrow{\mathbf{R2}} r_{3_c} \dots$$

$$\dots \xrightarrow{CRC-32} h_{4_c} \xrightarrow{\mathbf{R1}} r_{4_c} \xrightarrow{CRC-32} h_{5_c} \xrightarrow{\mathbf{R1}} r_{5_c} \xrightarrow{CRC-32} h_{6_c} \xrightarrow{\mathbf{R2}} r_{6_c} \dots$$

$$\dots \xrightarrow{CRC-32} h_{7_c} \xrightarrow{\mathbf{R2}} r_{7_c} \xrightarrow{CRC-32} h_{8_c} \xrightarrow{\mathbf{R2}} r_{8_c} \xrightarrow{CRC-32} h_{9_c} \dots$$

**Figura 4.3:** Generación de una fila combinando funciones de reconstrucción mediante un patrón extenso

Haciendo uso del método de generación de filas de la tabla del arco iris visto en la Figura 4.3, el proceso a seguir para la construcción de una tabla del arco iris que emplea esta configuración de funciones de reconstrucción se muestra en el algoritmo 4.6.

**Algorithm 4.6** Algoritmo de generación de tablas del arco iris que emplean el patrón extenso de funciones de reconstrucción

**Entrada:** tabla del arco iris vacía $rt$, número de filas $m$, número de columnas $t$, función resumen $H$, función de reconstrucción $R1$, función de reconstrucción $R2$
**Salida:** tabla del arco iris $rt$ con dimensiones $m \times t$

```
i, j, ptr = 1
while i ≤ m do
  p ← Contraseña inicial de la fila
  h ← H(p)
  while j ≤ t do
    if (ptr == 1)or(ptr == 4)or(ptr == 5) then
      r ← R1(h)
    else
      r ← R2(h)
    end if
    h ← H(r)
    j ← j + 1
    if ptr == 8 then
      ptr = 1
    else
      ptr ← ptr + 1
    end if
  end while
  i ← i + 1
  rt.append({p, h})
end while
```

De nuevo, la búsqueda de colisiones empleando las tablas del arco iris que hacen uso de este patrón de funciones de reconstrucción seguirá el mismo comportamiento que las tablas que emplean el patrón reducido del apartado anterior. En esta ocasión, como el patrón a utilizar tiene un ciclo de ocho pasos, la búsqueda

será realizada empleando el patrón inicial, seguido de los siete desplazamientos posibles en el caso de no dar con la colisión buscada.El algoritmo 4.7 muestra la forma en que se llevará a cabo dicha búsqueda de colisiones para la función resumen.

**Algorithm 4.7** Algoritmo de búsqueda de colisiones en tablas del arco iris que emplean el patrón extenso de funciones de reconstrucción

**Entrada:** tabla del arco iris $rt$ con dimensiones $m \times t$, número de filas $m$, número de columnas $t$, función resumen $H$, valor *hash* del cual se busca la colisión $h$, función de reconstrucción $R1$, función de reconstrucción $R2$

```
if ∃ una entrada{ini, h} then
  búsqueda finalizada con éxito, devolver la colisión a partir de ini
end if
i = 1
while i ≤ longitud de la secuencia R1-R2-R2-R1-R1-R2-R2-R2 do
  r ← i-ésima función de reconstrucción en R1-R2-R2-R1-R1-R2-R2-R2
  for j = 1 hasta t do
    h ← H(r(h))
    if ∃ una entrada {ini, h} then
      búsqueda finalizada con éxito, devolver la colisión a partir de ini
    else
      r ← siguiente función de reconstrucción en la secuencia R1-R2-R2-R1-
      R1-R2-R2-R2
    end if
  end for
  i ← i + 1
end while
Devolver error en la búsqueda
```

Los resultados tras aplicar el ataque del arco iris con las tablas que hacen uso de esta configuración son los siguientes:

| $t$ \ $m$ | 1000 | 2000 | 2500 | 5000 | 10000 | 20000 | 25000 | 50000 | 100000 | 200000 | 250000 | 500000 | 1000000 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | | | | | | | | | | | | 100 % |
| 2 | | | | | | | | | | | | 98.5 % | |
| 4 | | | | | | | | | | | 99.9 % | | |
| 5 | | | | | | | | | | 100 % | | | |
| 10 | | | | | | | | | 100 % | | | | |
| 20 | | | | | | | | 100 % | | | | | |
| 40 | | | | | | | 100 % | | | | | | |
| 50 | | | | | | 100 % | | 100 % | 100 % | | 100 % | 100 % | 100 % |
| 100 | | | | | 100 % | 100 % | 100 % | 100 % | 100 % | | 100 % | 100 % | 100 % |
| 200 | | | | 100 % | | | | | | | | | |
| 400 | | | 100 % | | | | | | | | | | |
| 500 | | 100 % | | | | | | | | | | | |
| 1000 | 63.0 % | | | | | | | | | | | | |

**Tabla 4.5:** Porcentajes de éxito para las tablas empleando el patrón extenso

Estos resultados han conseguido mejorar respecto a todos los vistos con anterioridad. El único empeoramiento comparando con los resultados obtenidos por las tablas que emplean el patrón reducido ocurre en la tabla de dimensiones 1.000 $\times$ 1.000. Todas las tablas restantes rinden mejor. De hecho, tan sólo se han generado tres tablas que no consiguen encontrar todas las colisiones buscadas. Para la primera de ellas, esto se debe al reducido número de filas que posee, ya que no consigue compensar aún teniendo la mayor profundidad de todas las tablas. Para las otras dos tablas que no han sido capaces de encontrar todas las colisiones hace falta observar si número de columnas. Esto justifica su rendimiento, ya que al poseer una profundidad tan reducida el patrón extenso no puede ser ejecutado en su totalidad. Aún con todo, estas tablas consiguen hacerse con una buena cantidad de colisiones.

## 4.5 En busca de las tablas más eficientes

Tras la 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 encontrar la mayoría de las colisiones buscadas, las cuales están representadas por los valores *hash* de las contraseñas almacenadas en el sistema. Si bien la intención del ataque del arco iris es dar con el mayor número de colisiones posible, resultaría interesante indagar en tablas de menor tamaño, para así poder determinar si realmente es necesario crear tablas que contengan tantas entradas, 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.

### 4.5.1. Nuevos tamaños

Si bien en las pruebas anteriores la tabla con el menor número de filas poseía $m = 1.000$ 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 tablas de tamaños obtenidos mediante diferentes combinaciones de los siguientes conjuntos de $m$ y $t$:

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

### 4.5.2. Colisiones internas de las nuevas tablas

En total se generarán tablas de tamaños muy diversos para cada combinación de funciones de reconstrucción. Tras generar todas estas tablas, pero antes de obtener su rendimiento, resulta de interés observar las colisiones internas que aparecen en dichas tablas para las diferentes combinaciones de funciones de reconstrucción. Cabe recordar que estas colisiones corresponderán únicamente con las encontradas en las últimas entradas de la tabla, por lo cual no se buscarán en las entradas intermedias. Para encontrar estas colisiones se obtendrá su porcentaje para cada tabla, obteniendo primero el número de colisiones internas recorriendo sus últimas columnas, determinando cuántos de sus valores no se repiten y asignando esa cantidad a un valor $c$, para seguidamente restarlo al número total de filas de la tabla ($m$), siendo el resultado de esta operación el número de colisiones internas que ocurren en la tabla. Acto seguido dicho número de colisiones internas $c$ será dividido entre el número de filas de la tabla $m$, dando como resultado el porcentaje de colisiones internas de la tabla respecto a su número de filas. Para dichos porcentajes, al igual que se ha hecho anteriormente 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 %.

Siempre que se hable de colisiones dentro de una tabla del arco iris se estará haciendo referencia a sus colisiones internas en sus últimas entradas, mientras que cuando se esté hablando de las colisiones encontradas por una tabla siempre será relacionado con el proceso de búsqueda de colisiones de la función resumen atacada.

Interpretando el código de color anterior, podría decirse que una tabla que tenga colisiones internas en un 70 por ciento de sus filas o más no serían las más ideales para ser empleadas en la búsqueda de colisiones para una función *hash*, ya que tantas colisiones internas se traducen en una reducción muy significante de la información cubiertas por esa tabla. Si este porcentaje se reduce un poco, mientras una tabla cubra cerca de la mitad de las colisiones de valores *hash* que se le presuponen, serán sus dimensiones las que determinen su utilidad, ya que si la tabla es lo suficientemente grande será capaz de compensar la pérdida de colisiones cubiertas con su tamaño. Cualquier tabla que cubra más de la mitad de las colisiones supuestas por su tamaño se considerará como 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 colisiones que debería 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:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $t$ | 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 % | 32.75 % |
| | 200 | 3.60 % | 4.60 % | 6.80 % | 8.90 % | 13.00 % | 16.00 % | 17.92 % | 30.60 % | 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 |
| | | $m$ | | | | | | | | |

**Tabla 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 internas menores respecto a las tablas de mayor tamaño. Las únicas tablas con un porcentaje de colisiones internas muy restrictivo ocurren con unas dimensiones superiores al dominio $N$, lo cual no es sorprendente en tales casos. Se puede concluir que es uso exclusivo de **R1**, en cuanto a colisiones internas se refiere, parece prometer resultados de buena calidad a la hora de atacar las contraseñas.

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

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $t$ | 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 |
| | | $m$ | | | | | | | | |

**Tabla 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 internas en las últimas entradas, 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 internas, la más grande de ellas superando el 90 por ciento de las mismas, 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 alternan ambas funciones de reconstrucción se obtiene:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $t$ | 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 |
| | | $m$ | | | | | | | | |

**Tabla 4.8:** Porcentajes de colisiones para las tablas empleando la alternancia de las funciones de reconstrucción

Aunque se ha mejorado respecto a las tablas que emplean únicamente **R2**, existen más colisiones internas en estas tablas que en comparación a las primeras que hacen uso tan sólo de **R1**, aunque en general estos resultados son comparables. Este pequeño empeoramiento resulta sorprendente, ya que se esperaba que al combinar ambas funciones de reconstrucción el porcentaje de colisiones internas 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 buenos en este contexto.

Para las tablas que emplean el denominado patrón reducido (**R1 + R2 + R2**) se generan los siguientes resultados:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $t$ | 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 |
| | | $m$ | | | | | | | | |

**Tabla 4.9:** Porcentajes de colisiones para las tablas empleando el patrón reducido

Los resultados mostrados en la Tabla 4.9 son los mejores en cuanto al porcentaje de colisiones internas se refiere. Así mismo, estos reducidos porcentajes de colisiones internas pueden justificar los buenos resultados obtenidos anteriormente en la Tabla 4.4. Incluso observando las tablas de mayores dimensiones no existe ninguna con un porcentaje de colisiones internas por encima del 70 por ciento, a la vez que únicamente nueve tablas de las 54 totales superan el 50 por ciento de colisiones internas. La mayor ventaja de tener estos porcentajes tan reducidos reside en el elevado número de colisiones de la función resumen que estarán cubiertas respecto a las tablas que emplean una configuración de funciones de reconstrucción diferente. De esta manera, las tablas que hacen uso del patrón reducido de funciones de reconstrucción prometen generar muy buenos resultados.

Por último, en cuanto a las tablas que emplean el denominado patrón extenso (**R1 + R2 + R2 + R1 + R1 + R2 + R2 + R2**) se observan los siguientes resultados:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $t$ | 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 |
| | | $m$ | | | | | | | | |

**Tabla 4.10:** Porcentajes de colisiones para las tablas empleando el patrón extenso

Puede observarse como las tablas con esta configuración poseen menor número de colisiones internas que las anteriores en la Tabla 4.6, lo cual era de esperar,

ya que el objetivo de establecer este patrón era forzar una reducción en el número de colisiones internas en las tablas. Lo sorprendente en esta caso es que hay unos porcentajes de colisiones internas muy similares a los de la Tabla 4.8 (**R1 + R2**), aunque algo mejores. Esto resulta bastante inesperado, ya que la alternancia de funciones de reconstrucción conforma un patrón significativamente más simple, dando a entender que produciría una mayor cantidad de colisiones internas, empeorando los resultados. Otro hecho sorprendente en este caso es que las tablas que emplean el patrón reducido resulten mejores en cuanto al porcentaje de colisiones internas respecto al patrón extenso, siendo esta mejora más que notable.

## 4.6 Estudio del éxito de las nuevas tablas en la recuperación de colisiones

Habiendo obtenido los porcentajes de colisiones internas 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 colisiones encontradas para la función resumen, 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 internas 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 reducido (**R1 + R2 + R2**), ya que presentan una diferencia significante en sus porcentajes de colisiones, como se ha visto en la Tabla 4.9.

Como ya se ha hecho anteriormente en el apartado 4.2.6, cada una de las tablas que se pretende analizar tendrá la tarea de atacar los mismos valores *hash* de las contraseñas aleatorias generadas en el apartado 4.2.5, obteniendo el porcentaje de contraseñas rotas respecto al total de contraseñas a romper, indicando la cantidad de colisiones encontradas para la función resumen CRC-32.

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:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $t$ | 50 | 14.6 % | 27.9 % | 39.0 % | 46.6 % | 59.4 % | 67.8 % | 73.8 % | 87.4 % | 94.7 % |
| | 100 | 22.1 % | 41.9 % | 55.8 % | 65.8 % | 77.2 % | 81.6 % | 85.9 % | 92.3 % | 96.4 % |
| | 200 | 24.0 % | 46.2 % | 61.2 % | 71.3 % | 81.6 % | 87.7 % | 91.0 % | 95.2 % | 97.7 % |
| | 400 | 23.7 % | 45.0 % | 60.5 % | 69.9 % | 82.1 % | 88.5 % | 91.8 % | 96.0 % | 98.3 % |
| | 500 | 23.5 % | 44.3 % | 58.2 % | 67.0 % | 78.3 % | 84.1 % | 87.9 % | 94.6 % | 97.9 % |
| | 1000 | 22.7 % | 42.4 % | 55.1 % | 63.5 % | 72.7 % | 77.6 % | 80.7 % | 86.2 % | 89.3 % |
| | | 250 | 500 | 750 | 1000 | 1500 | 2000 | 2500 | 5000 | 10000 |
| | | $m$ | | | | | | | | |

**Tabla 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 internas 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:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $t$ | 50 | 15.9 % | 29.7 % | 40.2 % | 48.8 % | 61.2 % | 69.7 % | 76.0 % | 88.6 % | 94.4 % |
| | 100 | 22.2 % | 41.7 % | 56.2 % | 66.2 % | 76.9 % | 82.8 % | 87.0 % | 93.4 % | 97.1 % |
| | 200 | 24.1 % | 44.4 % | 61.2 % | 70.3 % | 82.1 % | 87.3 % | 90.2 % | 94.3 % | 97.4 % |
| | 400 | 22.1 % | 41.0 % | 57.0 % | 64.7 % | 74.6 % | 78.9 % | 82.1 % | 90.1 % | 93.8 % |
| | 500 | 21.1 % | 38.2 % | 53.1 % | 60.8 % | 69.3 % | 74.9 % | 77.6 % | 81.9 % | 85.3 % |
| | 1000 | 20.5 % | 35.6 % | 48.0 % | 57.1 % | 69.4 % | 73.8 % | 76.1 % | 79.3 % | 79.6 % |
| | | 250 | 500 | 750 | 1000 | 1500 | 2000 | 2500 | 5000 | 10000 |
| | | $m$ | | | | | | | | |

**Tabla 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 internas de estas tablas (Tabla 4.7) respecto a las anteriores (Tabla 4.6). Otra diferencia que se puede apreciar al comparar estos resultados con los obtenidos en la Tabla 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 internas.

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:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $t$ | 50 | 15.5 % | 29.4 % | 42.5 % | 51.5 % | 65.8 % | 75.9 % | 82.4 % | 96.1 % | 98.8 % |
| | 100 | 23.8 % | 45.4 % | 63.3 % | 74.5 % | 88.3 % | 93.8 % | 96.2 % | 99.4 % | 99.9 % |
| | 200 | 24.4 % | 47.3 % | 68.2 % | 83.6 % | 94.4 % | 97.2 % | 98.1 % | 99.9 % | 100 % |
| | 400 | 24.4 % | 45.3 % | 64.1 % | 77.3 % | 92.6 % | 96.8 % | 98.1 % | 99.8 % | 100 % |
| | 500 | 24.2 % | 44.5 % | 62.30 % | 74.4 % | 89.9 % | 95.4 % | 97.7 % | 99.6 % | 100 % |
| | 1000 | 22.6 % | 39.3 % | 53.1 % | 63.1 % | 77.3 % | 85.9 % | 89.9 % | 95.4 % | 97.9 % |
| | | 250 | 500 | 750 | 1000 | 1500 | 2000 | 2500 | 5000 | 10000 |
| | | $m$ | | | | | | | | |

**Tabla 4.13:** Porcentajes de éxito para las tablas que emplean la alternancia de funciones de reconstrucción

Como ya ocurriera en las primeras pruebas realizadas en el apartado 4.2.6, las tablas que emplean la secuencia de funciones de reconstrucción **R1 + R2** consiguen encontrar un mayor número de colisiones para la función *hash* que aquellas tablas que tan sólo utilizan una función de reconstrucción. El número de tablas con un rendimiento pobre se ha reducido, mientras que ha aumentado de forma significativa la cantidad de tablas idóneas, algunas de ellas llegando incluso a ser capaces de encontrar colisiones para todas las contraseñas. Al igual que ha ocurrido con las tablas del arco iris anteriores (Tabla 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, con $m$ =1.500. 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 para encontrar todas las colisiones almacenadas, lo cual permite reducir los requerimientos de memoria significativamente.

Realizando el ataque con las tablas que emplean el denominado patrón reducido (**R1 + R2 + R2**) pueden obtenerse los siguientes resultados:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $t$ | 50 | 17.0 % | 32.0 % | 45.6 % | 55.5 % | 71.0 % | 82.0 % | 88.2 % | 98.0 % | 99.8 % |
| | 100 | 22.9 % | 44.7 % | 63.6 % | 76.7 % | 91.8 % | 96.5 % | 98.5 % | 100 % | 100 % |
| | 200 | 24.4 % | 48.0 % | 69.4 % | 85.8 % | 97.3 % | 99.3 % | 99.7 % | 100 % | 100 % |
| | 400 | 24.5 % | 47.8 % | 70.9 % | 88.0 % | 96.0 % | 99.0 % | 99.5 % | 99.9 % | 100 % |
| | 500 | 24.6 % | 48.0 % | 70.5 % | 88.0 % | 97.2 % | 99.3 % | 99.7 % | 100 % | 100 % |
| | 1000 | 23.6 % | 43.9 % | 62.7 % | 80.5 % | 95.6 % | 99.1 % | 99.8 % | 99.9 % | 100 % |
| | | 250 | 500 | 750 | 1000 | 1500 | 2000 | 2500 | 5000 | 10000 |
| | | $m$ | | | | | | | | |

**Tabla 4.14:** Porcentajes de éxito para las tablas que emplean patrón reducido

Estos resultados son mejores que los obtenidos con la alternancia de las funciones de reconstrucción. Las tablas generadas mediante esta configuración requieren un tamaño mínimo de tabla para obtener buenos resultados de 1.500 filas. En esta ocasión también se han obtenido tablas capaces de encontrar todas las colisiones buscadas, requiriendo la mitad de filas para que así sea respecto a las tablas de la configuración anterior, en la Tabla 4.13. De estas tablas que rompen todas las contraseñas, la de menor tamaño corresponde a unas dimensiones de $5.000 \times 100$, lo cual está por debajo del dominio $N$ establecido, ya que en principio esta tabla cubre únicamente la mitad de contraseñas del dominio. Hay otro 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 $m$.

Por último, observando los resultados obtenidos con tablas que emplean el denominado patrón extenso (**R1 + R2 + R2 + R1 + R1 + R2 + R2 + R2**) se obtiene:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $t$ | 50 | 18.9 % | 36.2 % | 50.9 % | 64.7 % | 81.8 % | 90.5 % | 94.8 % | 99.9 % | 100 % |
| | 100 | 23.6 % | 47.0 % | 67.0 % | 83.1 % | 96.9 % | 99.9 % | 100 % | 100 % | 100 % |
| | 200 | 24.4 % | 47.7 % | 68.9 % | 87.8 % | 99.4 % | 100 % | 100 % | 100 % | 100 % |
| | 400 | 24.0 % | 45.9 % | 65.3 % | 83.1 % | 99.9 % | 100 % | 100 % | 100 % | 100 % |
| | 500 | 24.1 % | 47.7 % | 69.1 % | 88.8 % | 99.8 % | 100 % | 100 % | 100 % | 100 % |
| | 1000 | 22.4 % | 39.0 % | 51.8 % | 63.0 % | 81.8 % | 95.8 % | 99.7 % | 100 % | 100 % |
| | | 250 | 500 | 750 | 1000 | 1500 | 2000 | 2500 | 5000 | 10000 |
| | | $m$ | | | | | | | | |

**Tabla 4.15:** Porcentajes de éxito para las tablas que emplean patrón extenso

Los resultados obtenidos en este caso pueden clasificarse como los mejores de entre todas las configuraciones distintas, al igual que ya ocurriera en la primera tanda de pruebas en el apartado 4.2.6. Se han obtenido una gran cantidad de tablas capaces de encontrar todas las colisiones buscadas, un total de 18 de las 54 totales, exactamente un tercio de todas las tablas con esta configuración. De todas ellas, la que tiene una menor dimensión es aquella con $m$ =2.000 filas y $t$ =200 columnas, la cual en principio debería cubrir tan sólo un 40 por ciento de las contraseñas del dominio $N$. 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.

Estos resultados muestran la importancia de averiguar una buena configuración de funciones de reconstrucción, ya que la calidad de los mismos aumentará de manera significativa si se obtiene una buena configuración.

## 4.7 Comparación y discusión

Tras implementar el ataque del arco iris con las nuevas tablas haciendo uso de tamaños de tabla diferentes y representar los resultados obtenidos, las configuraciones establecidas podrían clasificarse de la siguiente manera empleando como criterio de clasificación la cantidad de colisiones encontradas para la función resumen CRC-32:

1. Patrón extenso
2. Patrón reducido
3. Concatenación de funciones de reconstrucción
4. Uso exclusivo de **R1**
5. Uso exclusivo de **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, los mejores resultados se obtienen al combinar las funciones de reconstrucción. 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 internas de las tablas uno de los aspectos más importantes. Esto queda reflejado si se comparan los resultados obtenidos con los dos patrones establecidos respecto a las demás configuraciones, ya que la diferencia en la calidad de los resultados respecto a los demás resulta muy notoria.

- **Colisiones internas de las tablas**: Aunque este aspecto por sí sólo no es el más importante, es obvio que tiene un impacto significativo a la hora de encontrar colisiones para una función resumen. Un buen ejemplo para reflejar la importancia de mantener un número reducido de colisiones internas se encuentra en las tablas que hacen uso exclusivamente de la función de reconstrucción **R2** (Tabla 4.7), ya que dichas tablas son las que poseen un mayor número de colisiones internas y a la vez presentan los peores resultados (Tabla 4.12). A la vez, los resultados obtenidos por las tablas que emplean los patrones establecidos (reducido y extenso) deben parte de su calidad a la reducción de sus colisiones internas con respecto a las tablas de las demás configuraciones, ya que esto significa que cubren una mayor cantidad de colisiones de la función resumen. La clave del éxito de una tabla en cuanto a su porcentaje de colisiones internar se refiere reside en ser capaz de cubrir un rango suficiente de colisiones de la función *hash*.

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 internas de las tablas se verían reducidos, ya que con valores *hash* más largos disminuye su probabilidad. 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 internas.

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, mientras la calidad de los resultados debería mantenerse en un rango cercano a los aquí obtenidos. 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 textos válidos. En el caso de no tener conocimiento de dichas longitudes se correría el riesgo de producir textos no válidos 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 en el dominio de textos válidos**: 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 reconstrucción 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. Esto puede considerarse como consecuencia de lo visto en [9]. 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 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 cuanto a la estructura de datos utilizada. En el caso de realizar el ataque contra un dominio 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.

# CAPÍTULO 5
# Conclusiones

Tras observar la idea general de la primera implementación de Hellman [3] y tener en cuenta pero finalmente no aplicar la mejora propuesta por Rivest [8], 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. De tal forma, se planteó el estudio del uso de las tablas del arco iris como técnica de búsqueda de colisiones para funciones resumen.

Tras establecer el funcionamiento general en cuanto a la generación de las tablas se refiere, el paso siguiente corresponde a determinar una o varias funciones de reconstrucción a emplear durante dicha generación. Tras varias pruebas con diversas funciones se optó por emplear inicialmente la función **R1**, la cual proporcionó resultados razonablemente buenos. Tras ello, surgió la duda de si esta función, por sí misma, no era la más apropiada para este ataque, por lo cual se pasó a emplear la segunda función de reconstrucción **R2**, la cual no consiguió mejorar los resultados obtenidos anteriormente por la función **R1**. Esto sirvió para reflejar la importancia de concebir una función de reconstrucción apropiada, la cual variará según la implementación que se esté llevando a cabo. 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 internas de las tablas, permitiendo aumentar la cantidad de colisiones de la función resumen 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 internas de las mismas mediante el uso de patrones de mayor longitud que esta alternancia de funciones. Este uso de las funciones de reconstrucción en forma de patrones representa una aportación original de este trabajo al ataque del arco iris, siendo su objetivo principal reducir la cantidad de colisiones internas de las tablas para así aumentar el porcentaje de éxito en la búsqueda de colisiones para la función resumen. El primero de estos añadía únicamente un paso más (**R1 + R2 + R2**), mientras que el segundo resultó ser de mayor longitud (**R1 + R2 + R2 + R1 + R1 + R2 + R2 + R2**). De esta manera es establecieron los denominados patrones reducido y extenso, respectivamente, los cuales consi-

guieron con éxito reducir la cantidad de colisiones internas en sus tablas respecto a las demás configuraciones.

En una primera instancia se optó por generar tablas del arco iris 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 internas en dichas tablas. Tras realizar estos primeros ataques empleando los tamaños de tabla iniciales con las cinco configuraciones distintas y obtener sus resultados, 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 el patrón extenso, visto en la Figura 4.3, como puede observarse en las Tablas 4.5 y 4.15. 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 obtener la mayoría de las colisiones de la función *hash*. 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 encontrar todas las colisiones para 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 colisiones, así como la memoria necesaria para almacenar la tabla. Por último, es fácilmente observable que las tablas que hacen uso exclusivamente de la función de reconstrucción **R2** brindan los peores resultados, aunque puedan ser considerados buenos 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 internas, lo cual no ocurrió finalmente como se puede observar en la Tabla 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 dimensionalidad 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 no llegaban a cubrir el dominio $N$ en su totalidad. Aún así, generarlas y obtener sus resultados ha resultado ser necesario para poder establecer una frontera con claridad en las búsqueda por la tabla más eficiente de construir con mejores resultados. Habiendo alcanzado este punto no se estimó necesario establecer una nueva configuración de funciones de reconstrucción para las tablas,

por lo cual el proceso de obtención de resultados fue idéntico al empleado en el apartado 4.2.6.

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 su número de colisiones internas. 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 internas de las tablas se refiere, se aprecia que la función de reconstrucción **R2** es la peor en este aspecto de manera considerable (Tabla 4.7), mientras que el patrón reducido (Tabla 4.9) genera las tablas con el menor número de colisiones internas, lo cual en principio podría emplearse como argumento para predecir que estas tablas conseguirían encontrar el mayor número de colisiones de la función resumen atacada.

Finalmente, una vez implementado el ataque del arco iris empleando las tablas con los nuevos tamaños establecidos en el apartado 4.5.1 y habiendo representado los resultados generados, de nuevo las configuraciones establecidas podrían clasificarse de la siguiente manera empleando como criterio de clasificación la cantidad de colisiones encontradas para la función *hash* CRC-32:

1. Patrón extenso
2. Patrón reducido
3. Alternancia de funciones de reconstrucción
4. Uso exclusivo de **R1**
5. Uso exclusivo de **R2**

En la sección 4.7 ya han sido establecidos los motivos por los cuales se da lugar dicha clasificación en la implementación desarrollada para este proyecto. Con todo, los resultados de todas las configuraciones distintas han sido buenos en general, a la vez que han servido para reflejar la importancia de los diferentes factores de las tablas del arco iris a tener en cuenta, especialmente las dimensiones, el porcentaje de colisiones internas de las mismas y el buen uso de funciones de reconstrucción apropiadas. Los resultados que hacen uso de los patrones de las funciones de reconstrucción también han mostrado que es posible utilizar tablas del arco iris de tres ordenes de magnitud menores que el dominio de colisiones a encontrar, y obtener buenos resultados.

Como trabajo futuro queda pendiente, además de la incesante búsqueda de la función de reconstrucción ideal, realizar la implementación propuesta por Rivest [8] y sus puntos de control, como se ha visto en la sección 3.3, con el objetivo de comprobar si, en este caso, consigue mejorar los resultados obtenidos. También resultaría interesante lanzar este ataque contra una función resumen que ofrezca mayor seguridad que la función CRC-32, así como empleando un dominio de contraseñas de mayor tamaño, viendo de esta forma si la escalabilidad del ataque del arco iris permite obtener resultados suficientemente similares a los aquí vistos.

Tras todo esto, se espera que la efectividad del ataque del arco iris haya quedado suficientemente clara. Siempre que se realice este ataque con el objetivo de encontrar colisiones de una función *hash* que no utilice técnicas como el *salting*, *peppering* o el estiramiento (explicados en la sección 3.4), las tablas del arco iris serán capaces de hacerse con la gran mayoría de las colisiones deseadas. 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 los ataque 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 aquellos empleadas en el almacenamiento de contraseñas en todos los sistemas operativos de Microsoft *Windows* hasta llegar a *Windows* 7, incluído. También 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 de seguridad de ambas funciones frente a los intentos de obtención de sus colisiones. Esto puede ser útil en aquellos casos en los cuales por especificaciones de diseño u otros motivos no sea posible hacer uso del *salting*, el *peppering* 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] Jonathan Katz, Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

[2] Mario Cortina Borja and John Haigh. The birthday problem. *Significance*, 4(3):124–127, 2007.

[3] Martin Hellman. A cryptanalytic time-memory trade-off. *IEEE transactions on Information Theory*, 26(4):401–406, 1980.

[4] Ralph Merkle and Martin Hellman. Hiding information and signatures in trapdoor knapsacks. *IEEE transactions on Information Theory*, 24(5):525–530, 1978.

[5] Stephen Pohlig and Martin Hellman. An improved algorithm for computing logarithms over gf (p) and its cryptographic significance (corresp.). *IEEE Transactions on information Theory*, 24(1):106–110, 1978.

[6] Whitfield Diffie and Martin E Hellman. Special feature exhaustive cryptanalysis of the nbs data encryption standard. *Computer*, 10(6):74–84, 1977.

[7] Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. In *Annual International Cryptology Conference*, pages 617–630. Springer, 2003.

[8] Dorothy Elizabeth Robling Denning. *Cryptography and data security*. page 100, Addison-Wesley Longman Publishing Co., Inc., 1982.

[9] David H Wolpert, William G Macready, et al. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.