**Documentación Algoritmos de Búsqueda durante el proceso de crackeo.**

En el contexto de la seguridad informática, el análisis y la búsqueda eficiente de información dentro de grandes volúmenes de datos es crucial. Este proyecto se enfoca en evaluar la eficacia de diversos algoritmos de búsqueda para identificar coincidencias de hashes NTLM en un conjunto de datos compuesto por hashes de contraseñas previamente filtradas en ataques cibernéticos.

Los datos utilizados comprenden dos archivos principales: limpio.txt y alargado.txt. El archivo limpio.txt contiene hashes NTLM únicos de contraseñas que han sido filtradas y el número de veces que cada contraseña ha sido comprometida. Por otro lado, el archivo alargado.txt incluye hashes NTLM derivados de un proceso de cracking utilizando la herramienta John the Ripper, además de hashes adicionales no presentes en el primer archivo para evaluar la capacidad de detección de los algoritmos introducidos de forma aleatoria.

El objetivo de los algoritmos es comparar eficientemente los hashes del archivo alargado.txt con los del archivo limpio.txt, ignorando las frecuencias y demás información adicional para centrarse únicamente en la coincidencia de los hashes NTLM. Para garantizar la comparación, se normaliza el formato de los hashes de ambos archivos.

Este proyecto implementa y evalúa seis algoritmos diferentes de búsqueda: búsqueda lineal, búsqueda binaria, árboles de búsqueda binaria, filtros Cuckoo y tablas hash.

1. **Prerrequisitos**

Para asegurar que los resultados de las pruebas sean consistentes y reproducibles, es fundamental especificar claramente el entorno en el que se han realizado. A continuación, se describen los prerrequisitos y las características del sistema utilizado para las pruebas de los algoritmos:

**Especificaciones de la Máquina Virtual**

* **Sistema Operativo**: Kali Linux
* **Memoria RAM**: 8 GB
* **Almacenamiento**: 100 GB
* **Procesador**: Intel Core i7 de 13ª generación

Estas especificaciones proporcionan un entorno controlado y suficientemente potente para realizar pruebas de rendimiento y funcionalidad de estructuras de datos complejas y operaciones de manejo de hashes.

**Software y Herramientas**

* **Python**: Versión 3.11
  + **Razón**: Utilizar la última versión estable de Python permite aprovechar mejoras en el rendimiento y compatibilidad con las últimas versiones de librerías externas.
* **Librerías Externas**:
  + **cuckoofilter**: Utilizada para la implementación de filtros Cuckoo.
  + **binascii**: Utilizada para la conversión entre formatos binarios y ASCII.
  + **Instalación de librerías**: Las librerías necesarias deben ser instaladas usando **pip**, el gestor de paquetes de Python, lo que garantiza que todas las dependencias están correctamente gestionadas y actualizadas.

Los hashes de contraseñas filtradas han sido descargados de la página <https://haveibeenpwned.com> con fines académicos. Estos hashes han sido introducidos en la herramienta John The Ripper con el objetivo de crackearlos.

1. **Búsqueda Lineal**

La búsqueda lineal, también conocida como búsqueda secuencial, es uno de los métodos más sencillos para encontrar un elemento dentro de una lista. Funciona revisando cada elemento de la lista de forma secuencial hasta encontrar una coincidencia con el elemento deseado.

Para ejecutar un script de búsqueda lineal en Python, generalmente no se requieren librerías externas, ya que se utiliza la sintaxis básica de Python. Sí que hemos usado la librería ***sys*** para medir el tamaño de la estructura de datos y ***time*** para medir el tiempo de búsqueda.

Funciones Implementadas en el script:

* **cargar\_hashes\_desde\_archivo(ruta\_archivo)**:
  + **Propósito**: Carga hashes desde un archivo dado, extrayendo únicamente la parte del hash de cada línea, ignorando la frecuencia de apariciones.
  + **Parámetros**: **ruta\_archivo**: la ruta del archivo de donde se leen los hashes.
  + **Funcionamiento**:
    - Abre el archivo en modo lectura.
    - Lee cada línea, separa por el carácter **:**, y toma la primera parte (el hash).
    - Almacena todos los hashes en una lista.
    - Imprime la cantidad de hashes cargados y devuelve la lista de hashes.
* **cargar\_hashes\_descifrados\_desde\_archivo(ruta\_archivo)**:
  + **Propósito**: Carga hashes descifrados desde un archivo, omitiendo la primera línea, convirtiéndolos a mayúsculas y eliminando el prefijo **$NT$** para normalizarlos al igual que la contraseña descifrada.
  + **Parámetros**: **ruta\_archivo**: la ruta del archivo de donde se leen los hashes descifrados.
  + **Funcionamiento**:
    - Abre el archivo en modo lectura.
    - Omitiendo la primera línea (generalmente usada para encabezados o información no relevante), procesa el resto de las líneas.
    - Para cada línea válida, extrae el hash, lo convierte a mayúsculas y elimina los prefijos específicos.
    - Almacena los hashes procesados en una lista.
    - Imprime la cantidad de hashes descifrados cargados y devuelve la lista.
* **buscar\_hashes\_lineal(hashes, hashes\_descifrados)**:
  + **Propósito**: Realiza una búsqueda lineal de los hashes del archivo **limpio.txt** en los hashes descifrados de **alargado.txt** y realiza estadísticas sobre el proceso.
  + **Parámetros**:
    - **hashes**: lista de hashes originales.
    - **hashes\_descifrados**: lista de hashes descifrados y modificados.
  + **Funcionamiento**:
    - Inicia un contador de tiempo.
    - Itera sobre cada hash original y verifica si este se encuentra en la lista de hashes descifrados.
    - Calcula el porcentaje de coincidencias.
    - Registra el tiempo de finalización y calcula el tiempo total transcurrido.
    - Imprime la hora de inicio y finalización del proceso, el tiempo total, el porcentaje de coincidencias, y estadísticas adicionales como falsos positivos y falsos negativos.
    - Calcula el tamaño en bytes de la estructura de datos usada para almacenar los hashes y lo imprime.

**Ejecución del Script:**

* Al final del script, se definen las rutas a los archivos y se llaman a las funciones en el orden adecuado para realizar la búsqueda lineal.
* Se cargan los hashes desde **limpio.txt** y los hashes descifrados desde **alargado.txt**.
* Finalmente, se ejecuta la función **buscar\_hashes\_lineal** con las listas de hashes obtenidas para realizar la comparación y análisis.

1. **Búsqueda Binaria**

La búsqueda binaria es un método de búsqueda eficiente que se utiliza en listas ordenadas para encontrar un elemento específico. Este algoritmo divide repetidamente a la mitad el rango de números hasta reducir las posibles ubicaciones a solo una. Para cada iteración, compara el elemento medio con el objetivo: si son iguales, la búsqueda termina; si el objetivo es mayor, continúa con la mitad superior; si es menor, procede con la mitad inferior. Es mucho más rápido que la búsqueda lineal, especialmente para grandes volúmenes de datos.

Para ejecutar un script de búsqueda Binaria en Python, generalmente no se requieren librerías externas, ya que se utiliza la sintaxis básica de Python. Sí que hemos usado la librería ***sys*** para medir el tamaño de la estructura de datos y ***time*** para medir el tiempo de búsqueda.

Funciones Implementadas en el script:

* **cargar\_hashes\_desde\_archivo(ruta\_archivo)**:
  + **Propósito**: Carga y procesa los hashes de un archivo de texto, convirtiéndolos a formato estándar para su comparación al igual que antes.
  + **Funcionamiento**: Lee cada línea del archivo, extrae y normaliza los hashes (elimina prefijos y convierte a mayúsculas si es necesario). También informa sobre la cantidad de hashes procesados.
* **crear\_filtro\_binario(hashes)**:
  + **Propósito**: Prepara los hashes para la búsqueda binaria ordenándolos.
  + **Funcionamiento**: Ordena la lista de hashes y notifica cuántos elementos contiene la lista ordenada.
* **busqueda\_binaria(hashes\_limpio, hashes\_alargado)**:
  + **Propósito**: Realiza la búsqueda binaria de cada hash en **hashes\_alargado** dentro de la lista ordenada **hashes\_limpio**. Es en esta función donde se establece el funcionamiento de búsqueda binaria reduciendo continuamente a la mitad hasta que es encontrado el hash o no se encuentre. El tiempo de búsqueda se inicia cuando se llama a esta función no registrando así el tiempo que se tarda en ordenar la lista.
  + **Funcionamiento**: Para cada hash en **hashes\_alargado**, realiza la búsqueda binaria en **hashes\_limpio**. Registra cada hash encontrado y calcula el porcentaje de coincidencias.
* **comparar\_hashes\_originales(hashes\_encontrados, hashes\_originales)**:
  + **Propósito**: Evalúa la precisión de la búsqueda identificando falsos positivos.
  + **Funcionamiento**: Compara los hashes encontrados con los originales para determinar cuántos son falsos positivos (no deberían haberse encontrado).
* **main()**:
  + **Propósito**: Coordina la ejecución del script, desde cargar datos hasta imprimir resultados y métricas de rendimiento.
  + **Funcionamiento**: Carga los datos, prepara el filtro, ejecuta la búsqueda binaria, mide el tiempo de ejecución, y finalmente evalúa y reporta resultados como el tiempo total de búsqueda, porcentaje de coincidencias y falsos positivos/negativos.

1. **Tablas Hash**

Las tablas hash son una estructura de datos que utiliza una función hash para mapear datos, como claves o identificadores, a una posición en un arreglo. Este método es especialmente eficiente para la búsqueda y verificación de elementos, ya que permite accesos de tiempo casi constante, independientemente del tamaño de la tabla. Son ideales para aplicaciones donde la inserción rápida y la búsqueda son críticas.

Para ejecutar un script de Tablas Hash en Python, generalmente no se requieren librerías externas, ya que se utiliza la sintaxis básica de Python. Sí que hemos usado la librería ***sys*** para medir el tamaño de la estructura de datos y ***time*** para medir el tiempo de búsqueda.

Funciones Implementadas en el script:

* **cargar\_hashes\_desde\_archivo(ruta\_archivo)**:
  + **Propósito**: Cargar hashes desde un archivo especificado.
  + **Funcionamiento**: Lee cada línea del archivo, extrae el hash y los imprime. Retorna una lista de estos hashes.
* **crear\_tabla\_hash(hashes)**:
  + **Propósito**: El propósito principal de esta función es crear una tabla hash que permita una verificación rápida y eficiente de la presencia de elementos. Esta estructura es fundamental para mejorar la eficiencia de las operaciones de búsqueda comparadas con una lista simple o una búsqueda secuencial.
  + **Funcionamiento**: Itera sobre la lista de hashes y los agrega a un diccionario, utilizando el hash como clave y asignándole un valor de **True**. Esto prepara la tabla para búsquedas rápidas.
* **comprobar\_hashes\_en\_tabla(tabla\_hash, hashes\_a\_comprobar)**:
  + **Propósito**: Comprobar si los hashes de una lista están presentes en la tabla hash.
  + **Funcionamiento**: Itera sobre los hashes a comprobar y busca cada uno en la tabla hash:
    - **Inicialización**: Se crea un diccionario vacío en Python, que actuará como la tabla hash.
    - **Iteración** **sobre los hashes**: La función itera sobre cada hash proporcionado en la lista de hashes.
    - **Inserción en la tabla hash:** Cada hash se agrega al diccionario. El hash mismo actúa como la clave, y se asigna un valor booleano True a cada clave. Este valor es arbitrario en este contexto específico, ya que lo importante es la presencia de la clave (el hash) en el diccionario, no el valor que esta clave mantiene.
    - **Eficiencia:** Al utilizar un diccionario, la inserción y la posterior búsqueda de claves (hashes) se realiza en tiempo constante promedio (O(1)), lo que hace que este método sea extremadamente eficiente para conjuntos de datos grandes.
* **comparar\_hashes\_originales(hashes\_encontrados, hashes\_originales)**:
  + **Propósito**: Identificar falsos positivos y negativos al comparar los hashes encontrados con los hashes originales.
  + **Funcionamiento**: Itera sobre los hashes encontrados y los originales para determinar cuáles no deberían haber sido encontrados y cuáles faltaron, respectivamente.
* **obtener\_tamano\_estructura\_datos(tabla\_hash)**:
  + **Propósito**: Obtener el tamaño en memoria de la tabla hash.
  + **Funcionamiento**: Utiliza **sys.getsizeof** para obtener y retornar el tamaño en bytes de la tabla hash.
* **main()**:
  + **Propósito**: Orquestar la ejecución completa del script.
  + **Funcionamiento**: Define las rutas de los archivos, carga los hashes, crea la tabla hash, verifica los hashes, compara los resultados, e imprime las métricas finales incluyendo el tamaño de la tabla hash.

1. **Árboles de Búsqueda Binaria**

Los árboles de búsqueda binaria son estructuras de datos que organizan los elementos de manera jerárquica, permitiendo búsquedas, inserciones y eliminaciones eficientes. Cada nodo del árbol contiene un valor, una referencia al nodo izquierdo y una referencia al nodo derecho. Los elementos en el subárbol izquierdo son menores que el valor del nodo, y los del subárbol derecho son mayores. Esta propiedad garantiza que las operaciones de búsqueda sean rápidas, particularmente cuando el árbol está equilibrado, proporcionando búsquedas en tiempo logarítmico en el caso promedio.

Para ejecutar este script en Python, hemos usado la librería ***sys*** para medir el tamaño de la estructura de datos y ***time*** para medir el tiempo de búsqueda al igual que antes y las librerías ***binascii*** usada para manipulaciones binarias y conversiones, crucial para manejar los hashes en forma binaria y la librería ***string*** necesaria para verificar si una cadena contiene solo caracteres hexadecimales. Sigue sin ser necesaria una librería externa ya que todas están incluidas de serie en Python.

Clases creadas en el script:

Es necesaria la creación de las clases ***NodoArbol*** y ***ArbolBusquedaBinaria*** en el script para implementar y manejar la estructura de un árbol de búsqueda binaria de manera eficiente y organizada.

* **NodoArbol**:
  + **Propósito**: Representar un nodo individual dentro del árbol.
  + **Atributos**:
    - **valor**: Almacena el valor del nodo (en este caso, el hash convertido a un entero).
    - **izquierda**: Referencia al nodo hijo izquierdo.
    - **derecha**: Referencia al nodo hijo derecho.
  + **Funcionalidad:** Almacena el valor del nodo y referencias a los nodos hijo izquierdo y derecho. La clase ***NodoArbol*** simplifica la gestión de estos enlaces, haciendo que las operaciones de inserción y búsqueda sean más intuitivas y menos propensas a errores de manejo manual de nodos.
* **ArbolBusquedaBinaria**:
  + **Propósito**: Implementar la estructura del árbol y proporcionar métodos para operar con él.
  + **Métodos**:
    - **agregar\_nodo(valor)**: Inserta un nuevo valor en el árbol en la posición correcta según las propiedades del BST.
    - **buscar\_nodo(valor)**: Busca un valor en el árbol y devuelve **True** si está presente.
    - **arbol\_inorden(nodo\_actual)**: Devuelve una lista de todos los valores en el árbol ordenados inorden.
    - **obtener\_profundidad()**: Calcula la profundidad máxima del árbol, útil para evaluar su balance.
  + **Funcionalidad**:
    - **Gestión de nodos:** Incluye métodos para agregar nodos y buscar valores dentro del árbol. Esto centraliza la lógica para manejar la estructura del árbol, como asegurar que se mantengan las propiedades de un BST durante inserciones y búsquedas.
    - **Operaciones de árbol:** Métodos como ***arbol\_inorden*** y ***obtener\_profundidad*** se encargan de elegir recorridos y calcular la profundidad, respectivamente, operaciones que son fundamentales para el análisis y la optimización del árbol.

Funciones Implementadas en el script:

* **cargar\_hashes\_desde\_archivo(ruta\_archivo)**:
  + **Propósito**: Cargar hashes desde un archivo y prepararlos para ser insertados en el árbol.
  + **Funcionamiento**: Lee cada línea del archivo, extrae el hash y lo prepara para la conversión y manipulación.
* **obtener\_tamano\_arbol(arbol)**:
  + **Propósito**: Calcular el tamaño en memoria del árbol.
  + **Funcionamiento**: Suma el tamaño de todos los nodos y sus valores almacenados.
* **crear\_arbol\_busqueda\_binaria(hashes)**:
  + **Propósito**: Crear un árbol de búsqueda binaria a partir de una lista de hashes, lo cual permite realizar búsquedas eficientes dentro del árbol una vez que todos los hashes han sido insertados.
  + **Funcionamiento Detallado**:
    - **Inicialización del Árbol:** La función comienza creando una instancia de Arbol, que inicialmente está vacía (sin raíz).
    - **Conversión de Hashes:** Cada hash, que es una cadena hexadecimal, necesita ser convertido en un valor numérico para ser manejado más eficientemente dentro del árbol. La conversión se hace de la siguiente manera:
      * Cada hash hexadecimal se convierte en bytes usando binascii.unhexlify(h), que transforma la cadena hexadecimal en una secuencia de bytes.
      * Luego, int.from\_bytes(hash\_val\_bytes, byteorder='little') convierte esta secuencia de bytes en un número entero. El argumento byteorder='little' especifica que el orden de los bytes es little-endian, es decir, el byte menos significativo está en la posición más baja (al principio).
    - Inserción en el Árbol:
      * Para cada valor entero derivado de un hash, se llama al método agregar\_nodo(valor) del árbol.
      * Dentro de agregar\_nodo, se verifica si la raíz del árbol está definida:
        + Si la raíz es None (el árbol está vacío), el nuevo nodo con este valor se establece como la raíz.
        + Si no, se llama al método auxiliar \_agregar\_nodo(valor, nodo\_actual) para encontrar la posición adecuada para el nuevo nodo.
    - Método \_agregar\_nodo(valor, nodo\_actual):
      * Este método es recursivo y es el núcleo de cómo se mantienen las propiedades del BST.
      * Compara el valor con el valor del nodo\_actual:
        + Si valor < nodo\_actual.valor, el método se llama recursivamente con el subárbol izquierdo (nodo\_actual. Izquierda). Si no existe subárbol izquierdo (es decir, nodo\_actual.izquierda es None), se crea un nuevo nodo en esta posición.
        + Si valor > nodo\_actual.valor, el proceso es similar, pero utilizando el subárbol derecho (nodo\_actual.derecha).
        + Si el valor ya existe en el árbol (valor == nodo\_actual.valor), no se hace nada, pues en un BST típicamente no se permiten valores duplicados.
* **obtener\_contraseñas\_descifradas(arbol, ruta\_archivo)**:
  + **Propósito**: Verificar qué hashes descifrados están presentes en el árbol.
  + **Funcionamiento**: Lee y convierte los hashes del archivo descifrado y los busca en el árbol.
* **comparar\_hashes(arbol, ruta\_hashes\_originales)**:
  + **Propósito**: Evaluar la precisión de los hashes almacenados en el árbol de búsqueda binaria comparándolos con un conjunto de hashes originales. Esta comparación ayuda a determinar qué tan bien el árbol ha capturado y puede reconocer los hashes que deberían estar presentes según el conjunto original, además de identificar aquellos que se registran incorrectamente.
  + **Funcionamiento**: Identifica y cuenta los falsos positivos y falsos negativos.
    - Para cada hash en la lista de hashes originales, la función realiza lo siguiente:
      * Convierte el hash de formato hexadecimal a un número entero (similar al proceso usado en crear\_arbol\_busqueda\_binaria). Esto implica descomponer el hash en bytes y luego convertir estos bytes en un entero usando int.from\_bytes.
      * Usa el método buscar\_nodo del árbol para verificar si este valor entero está presente en el árbol.

1. **Filtros Cuckoo**

Los filtros Cuckoo son estructuras de datos probalísticas que permiten operaciones de inserción y búsqueda eficientes, ideales para conjuntos de datos grandes donde los falsos positivos son aceptables pero deben mantenerse mínimos. Utilizan múltiples funciones hash para resolver colisiones mediante un método de reubicación que recuerda al comportamiento del cuco, desplazando otros elementos según sea necesario.

En los filtros Cuckoo, es posible la configuración de ciertos parámetros lo que es crucial para optimizar el rendimiento y la eficacia de la estructura. Los parámetros **capacity**, **fingerprint\_size**, y **bucket\_size** juegan roles importantes en cómo se comporta el filtro.

* **Capacity:** define el número máximo de elementos que el filtro está diseñado para almacenar. Una mayor **capacity** permite almacenar más elementos sin aumentar la tasa de falsos positivos ni requerir demasiadas reubicaciones, lo que puede ayudar a mantener la eficiencia de las operaciones de inserción y búsqueda.

Sin embargo, la **capacity** es demasiado baja en comparación con el número de elementos insertados, el filtro puede sufrir de muchas reubicaciones o incluso fallar al insertar nuevos elementos si se alcanza el límite de capacidad.

* **Fingerprint Size** El tamaño de la huella digital (en bits) que se utiliza para cada elemento insertado en el filtro. La huella digital es un resumen derivado del elemento que se utiliza en lugar del elemento completo para las operaciones de inserción y búsqueda.

Un **fingerprint\_size** mayor reduce la probabilidad de falsos positivos porque permite una mayor diferenciación entre los elementos almacenados. Aumentar el tamaño de la huella digital incrementa el uso de memoria del filtro, pero mejora la precisión. Un tamaño menor reduce el uso de memoria, pero a costa de aumentar la tasa de falsos positivos.

* **Bucket Size:** Define el número de huellas digitales que cada "bucket" (contenedor) del filtro puede almacenar. Cada bucket es una unidad básica de almacenamiento dentro de la estructura del filtro.

Un **bucket\_size** mayor puede reducir la necesidad de reubicaciones al proporcionar más espacio para cada conjunto de huellas digitales, lo que puede ser útil en entornos con alta densidad de inserciones.

Aunque un tamaño de bucket más grande puede ayudar a manejar colisiones, también puede llevar a un uso ineficiente de la memoria si muchos buckets no se utilizan completamente.

Para el script de búsqueda mediante Filtros Cuckoo hemos usado la librería ***cuckoofilter*** que proporciona una implementación eficiente de los filtros Cuckoo en Python con todas las funcionalidades requeridas para su comportamiento probabilístico. Se puede instalar mediante el comando ***pip install cuckoofilter***. También hemos usado las librerías que hemos estado mencionando en algoritmos anteriores, la librería ***sys*** para medir el tamaño de la estructura de datos y ***time*** para medir el tiempo de búsqueda.

Funciones Implementadas en el script:

* **cargar\_hashes\_desde\_archivo(ruta\_archivo)**:
  + **Propósito**: Cargar hashes desde un archivo de texto. Esta función es fundamental para iniciar el proceso, ya que proporciona los datos iniciales que se insertarán en el filtro.
  + **Funcionamiento**:
    - Abre el archivo en la ruta especificada y lee cada línea.
    - Divide cada línea por el carácter **':'** y toma la primera parte, asegurando que se trabaja solo con el hash.
    - Convierte cada hash a mayúsculas para estandarizar el formato antes de su inserción en el filtro.
    - Retorna una lista de hashes en formato estandarizado.
* **crear\_filtro\_cuckoo(hashes, capacity, fingerprint\_size)**:
  + **Propósito**: Inicializar un filtro Cuckoo y llenarlo con los hashes proporcionados. Esta función configura el filtro con una capacidad y tamaño de huella digital específicos.
  + **Funcionamiento**:
    - Crea una instancia de **CuckooFilter** con la capacidad y el tamaño de huella digital especificados.
    - Convierte cada hash hexadecimal a bytes.
    - Inserta cada hash convertido en el filtro.
    - Muestra cuántos elementos se han insertado y devuelve el filtro.
* **comprobar\_hashes\_en\_filtro\_cuckoo(filtro, hashes\_a\_comprobar)**:
  + **Propósito**: Verificar la presencia de hashes en el filtro Cuckoo y medir el rendimiento de la búsqueda.
  + **Funcionamiento**:
    - Convierte cada hash a verificar de formato hexadecimal a bytes.
    - Inicia un contador de tiempo para medir la duración del proceso de verificación.
    - Verifica cada hash convertido en el filtro.
    - Contabiliza y recopila los hashes encontrados.
    - Calcula y muestra el porcentaje de coincidencias y el tiempo total de la operación.
    - Devuelve la lista de hashes encontrados.
* **calcular\_falsos\_positivos(hashes\_originales, hashes\_encontrados)**:
  + **Propósito**: Evaluar la precisión del filtro mediante la determinación de la cantidad de verdaderos y falsos positivos.
  + **Funcionamiento**:
    - Compara los hashes encontrados con los hashes originales.
    - Contabiliza los verdaderos y falsos positivos, basándose en si los hashes encontrados estaban realmente en la lista original.
    - Muestra la cantidad de verdaderos positivos, falsos positivos y el total de hashes encontrados.
* **main()**:
  + **Propósito**: Orquestar el flujo completo del programa, desde la carga de datos hasta la evaluación de los resultados.
  + **Funcionamiento**:
    - Carga hashes originales y los hashes a verificar.
    - Experimenta con diferentes configuraciones del filtro Cuckoo, variando la capacidad y el tamaño de la huella.
    - Para cada configuración, crea un filtro, verifica los hashes y evalúa los resultados.
    - Finaliza mostrando un resumen de la ejecución.