Algorithm Documentation for Search during the Cracking Process

In the context of computer security, the efficient analysis and search of information within large data volumes is crucial. This project focuses on evaluating the effectiveness of various search algorithms to identify NTLM hash matches in a dataset composed of previously filtered password hashes from cyber attacks.

The data used comprises two main files: original.txt and extended.txt. The file original.txt contains unique NTLM password hashes and the number of times each password has been compromised. On the other hand, the file extended.txt includes NTLM hashes derived from a cracking process using the John the Ripper tool, as well as additional hashes not present in the first file to evaluate the detection capability of the algorithms introduced randomly.

The objective of the algorithms is to efficiently compare the hashes from extended.txt with those from original.txt, ignoring frequencies and other additional information to focus solely on NTLM hash matches. To ensure comparison, the hash format of both files is normalized.

This project implements and evaluates several different search algorithms: linear search, binary search, binary search trees, Bloom filters, and hash tables.

Prerequisites

To ensure that test results are consistent and reproducible, it is essential to clearly specify the environment in which they were conducted. Below are the prerequisites and system characteristics used for testing the algorithms:

Virtual Machine Specifications:

- Operating System: Kali Linux
- RAM: 8 GB
- Storage: 100 GB
- Processor: Intel Core i7 13th generation

These specifications provide a controlled and sufficiently powerful environment to conduct performance and functionality tests of complex data structures and hash handling operations.

Software and Tools:

- **Python:** Version 3.11
    - **Reason:** Using the latest stable version of Python allows taking advantage of performance improvements and compatibility with the latest versions of external libraries.
    - **External Libraries:**
        - `cuckoofilter`: Used for the implementation of Cuckoo filters.
        - `binascii`: Used for conversion between binary and ASCII formats.
    - **Library Installation:** The necessary libraries should be installed using `pip`, Python's package manager, ensuring that all dependencies are correctly managed and updated.

The filtered password hashes were downloaded from the page https://haveibeenpwned.com for academic purposes. These hashes were introduced into the John the Ripper tool to crack them.

Linear Search

Linear search, also known as sequential search, is one of the simplest methods for finding an element within a list. It works by checking each element of the list sequentially until a match with the desired element is found.

To execute a linear search script in Python, external libraries are generally not required as basic Python syntax is used. We have used the `sys` library to measure the data structure's size and `time` to measure search time.

Implemented Functions in the Script:

- **`load_hashes_from_file(file_path):`**
  - **Purpose:** Load hashes from a given file, extracting only the hash part of each line, ignoring the frequency of occurrences.
  - **Parameters:** `file_path`: the path of the file from which the hashes are read.
  - **Functioning:**
    - Opens the file in read mode.
    - Reads each line, splits by the character `:` and takes the first part (the hash).
    - Stores all hashes in a list.
    - Prints the number of loaded hashes and returns the hash list.
- **`load_decrypted_hashes_from_file(file_path):`**
  - **Purpose:** Load decrypted hashes from a file, skipping the first line, converting them to uppercase, and removing the prefix `$NT$` to normalize them just like the cracked password.
  - **Parameters:** `file_path`: the path of the file from which the decrypted hashes are read.
  - **Functioning:**
    - Opens the file in read mode.
    - Skips the first line (usually used for headers or irrelevant information) and processes the rest of the lines.
    - For each valid line, extracts the hash, converts it to uppercase, and removes specific prefixes.
    - Stores the processed hashes in a list.
    - Prints the number of loaded decrypted hashes and returns the list.
- **`linear_search_hashes(hashes, decrypted_hashes):`**
  - **Purpose:** Performs a linear search of the hashes from original.txt in the decrypted hashes from extended.txt and generates statistics about the process.
  - **Parameters:**
    - `hashes`: list of original hashes.
    - `decrypted_hashes`: list of decrypted and modified hashes.
  - **Functioning:**
    - Starts a time counter.
    - Iterates over each original hash and checks if it is in the decrypted hash list.
    - Calculates the percentage of matches.
    - Records the end time and calculates the total elapsed time.
    - Prints the start and end times of the process, the total time, the match percentage, and additional statistics like false positives and false negatives.
    - Calculates the size in bytes of the data structure used to store the hashes and prints it.

Script Execution:

At the end of the script, the file paths are defined, and the functions are called in the appropriate order to perform the linear search. The hashes are loaded from original.txt and the decrypted hashes from extended.txt. Finally, the function `linear_search_hashes` is executed with the obtained hash lists to perform the comparison and analysis.

Binary Search

Binary search is an efficient search method used on sorted lists to find a specific element. This algorithm repeatedly divides the range of numbers in half until it reduces the possible locations to just one. For each iteration, it compares the middle element with the target: if they are equal, the search ends; if the target is greater, it continues with the upper half; if it is smaller, it proceeds with the lower half. It is much faster than linear search, especially for large volumes of data.

To execute a binary search script in Python, external libraries are generally not required as basic Python syntax is used. We have used the `sys` library to measure the data structure's size and `time` to measure search time.

Implemented Functions in the Script:

- **`load_hashes_from_file(file_path)`:**
  - **Purpose:** Loads and processes hashes from a text file, converting them to standard format for comparison as before.
  - **Functioning:** Reads each line of the file, extracts and normalizes the hashes (removes prefixes and converts to uppercase if necessary). Also informs about the number of processed hashes.
- **`create_binary_filter(hashes)`:**
  - **Purpose:** Prepares the hashes for binary search by sorting them.
  - **Functioning:** Sorts the hash list and notifies how many elements the sorted list contains.
- **`binary_search(hashes_clean, hashes_extended)`:**
  - **Purpose:** Performs binary search for each hash in extended.txt within the sorted list original.txt. This function sets up the binary search functionality by continually halving until the hash is found or not found. The search time starts when this function is called, thus not recording the time taken to sort the list.
  - **Functioning:** For each hash in extended.txt, performs binary search in original.txt. Records each found hash and calculates the match percentage.
- **`compare_original_hashes(found_hashes, original_hashes)`:**
  - **Purpose:** Evaluates the accuracy of the search by identifying false positives.
  - **Functioning:** Compares the found hashes with the originals to determine how many are false positives (should not have been found).
- **`main()`:**
  - **Purpose:** Coordinates the script execution from loading data to printing results and performance metrics.
  - **Functioning:** Loads the data, prepares the filter, executes binary search, measures execution time, and finally evaluates and reports results such as total search time, match percentage, and false positives/negatives.

Hash Tables

Hash tables are a data structure that uses a hash function to map data such as keys or identifiers to a position in an array. This method is especially efficient for searching and verifying elements as it allows nearly constant time accesses regardless of the table size. They are ideal for applications where fast insertion and search are critical.

To execute a hash table script in Python, external libraries are generally not required as basic Python syntax is used. We have used the `sys` library to measure the data structure's size and `time` to measure search time.

Implemented Functions in the Script:

- **`load_hashes_from_file(file_path)`:**
  - **Purpose:** Load hashes from a specified file.
  - **Functioning:** Reads each line of the file, extracts the hash, and prints them. Returns a list of these hashes.
- **`create_hash_table(hashes)`:**
  - **Purpose:** The main purpose of this function is to create a hash table that allows quick and efficient verification of the presence of elements. This structure is fundamental to improve the efficiency of search operations compared to a simple list or sequential search.
  - **Functioning:** Iterates over the hash list and adds them to a dictionary using the hash as the key and assigning it a value of True. This prepares the table for quick searches.
- **`check_hashes_in_table(hash_table, hashes_to_check)`:**
  - **Purpose:** Check if the hashes in a list are present in the hash table.
  - **Functioning:** Iterates over the hashes to check and looks for each one in the hash table.
    - **Initialization:** Creates an empty dictionary in Python that will act as the hash table.
    - **Iteration over the hashes:** The function iterates over each provided hash in the list.
    - **Insertion into the hash table:** Each hash is added to the dictionary. The hash itself acts as the key, and a boolean value True is assigned to each key. This value is arbitrary in this specific context since what matters is the presence of the key (the hash) in the dictionary, not the value it maintains.
    - **Efficiency:** By using a dictionary, the insertion and subsequent search for keys (hashes) is performed in average constant time (O(1)), making this method extremely efficient for large data sets.
- **`compare_original_hashes(found_hashes, original_hashes)`:**
  - **Purpose:** Identify false positives and negatives by comparing found hashes with original hashes.
  - **Functioning:** Iterates over the found and original hashes to determine which should not have been found and which were missed, respectively.
- **`get_data_structure_size(hash_table)`:**
  - **Purpose:** Obtain the memory size of the hash table.
  - **Functioning:** Uses `sys.getsizeof` to get and return the size in bytes of the hash table.
- **`main()`:**
  - **Purpose:** Orchestrates the entire script execution.
  - **Functioning:** Defines the file paths, loads the hashes, creates the hash table, verifies the hashes, compares the results, and prints the final metrics, including the hash table size.

Binary Search Trees

Binary search trees are data structures that organize elements hierarchically, allowing efficient searches, insertions, and deletions. Each tree node contains a value, a reference to the left node, and a reference to the right node. The elements in the left subtree are less than the node's value, and those in the right subtree are greater. This property ensures that search operations are fast, particularly when the tree is balanced, providing logarithmic time searches on average.

To execute this script in Python, we have used the `sys` library to measure the data structure's size and `time` to measure search time, as before, and the `binascii` library used for binary manipulations and conversions crucial for handling hashes in binary form, and the `string` library necessary to verify if a string contains only hexadecimal characters. An external library is still not needed as all these are included with Python.

Classes Created in the Script:

It is necessary to create the `TreeNode` and `BinarySearchTree` classes in the script to efficiently and organizedly implement and manage a binary search tree structure.

- **`TreeNode`:**
  - **Purpose:** Represents an individual node within the tree.
  - **Attributes:**
    - `value`: Stores the node's value (in this case, the hash converted to an integer).
    - `left`: Reference to the left child node.
    - `right`: Reference to the right child node.
  - **Functionality:** Stores the node's value and references to the left and right child nodes. The `TreeNode` class simplifies managing these links, making insertion and search operations more intuitive and less prone to manual node handling errors.
- **`BinarySearchTree`:**
  - **Purpose:** Implements the tree structure and provides methods to operate on it.
  - **Methods:**
    - `add_node(value)`: Inserts a new value into the tree in the correct position according to BST properties.
    - `search_node(value)`: Searches for a value in the tree and returns True if it is present.
    - `inorder_tree(current_node)`: Returns a list of all values in the tree sorted in order.
    - `get_depth()`: Calculates the tree's maximum depth, useful for evaluating its balance.
  - **Functionality:**
    - **Node Management:** Includes methods to add nodes and search for values within the tree. This centralizes the logic to handle the tree structure, ensuring that BST properties are maintained during insertions and searches.
    - **Tree Operations:** Methods like `inorder_tree` and `get_depth` handle traversals and depth calculations, respectively, which are fundamental for analyzing and optimizing the tree.

Implemented Functions in the Script:

- **`load_hashes_from_file(file_path)`:**
  - **Purpose:** Load hashes from a file and prepare them to be inserted into the tree.
  - **Functioning:** Reads each line of the file, extracts the hash, and prepares it for conversion and manipulation.
- **`get_tree_size(tree)`:**
  - **Purpose:** Calculate the memory size of the tree.
  - **Functioning:** Sums the size of all nodes and their stored values.
- **`create_binary_search_tree(hashes)`:**

    - **Purpose:** Create a binary search tree from a list of hashes, allowing efficient searches within the tree once all hashes have been inserted.
    - **Detailed Functioning:**
        - **Tree Initialization:** The function starts by creating an instance of `BinarySearchTree` which is initially empty (without a root).
        - **Hash Conversion:** Each hash, which is a hexadecimal string, needs to be converted into a numeric value for efficient handling within the tree. The conversion is done as follows:
            - Each hexadecimal hash is converted into bytes using `binascii.unhexlify(h)`, which transforms the hexadecimal string into a byte sequence.
            - Then `int.from_bytes(hash_val_bytes, byteorder='little')` converts this byte sequence into an integer. The `byteorder='little'` argument specifies that the byte order is little-endian, meaning the least significant byte is at the lowest position (at the beginning).
        - **Insertion into the Tree:**
            - For each integer value derived from a hash, the `add_node(value)` method of the tree is called.
            - Within `add_node`, it checks if the tree's root is defined:
                - If the root is `None` (the tree is empty), the new node with this value is set as the root.
                - Otherwise, the auxiliary method `_add_node(value, current_node)` is called to find the appropriate position for the new node.
        - **Method `_add_node(value, current_node)`:**
            - This method is recursive and is the core of how BST properties are maintained.
            - Compares the value with the `current_node`'s value:
                - If `value < current_node.value`, the method is called recursively with the left subtree (`current_node.left`). If no left subtree exists (i.e., `current_node.left` is `None`), a new node is created at this position.
                - If `value > current_node.value`, the process is similar but uses the right subtree (`current_node.right`).
                - If the value already exists in the tree (`value == current_node.value`), nothing is done as duplicates are typically not allowed in a BST.
- **`get_decrypted_passwords(tree, file_path)`:**
    - **Purpose:** Verify which decrypted hashes are present in the tree.
    - **Functioning:** Reads and converts the hashes from the decrypted file and searches them in the tree.
- **`compare_hashes(tree, original_hashes_path)`:**
    - **Purpose:** Evaluate the accuracy of the hashes stored in the binary search tree by comparing them with a set of original hashes. This comparison helps determine how well the tree has captured and can recognize the hashes that should be present according to the original set, as well as identify those incorrectly recorded.
    - **Functioning:** Identifies and counts false positives and false negatives.
        - For each hash in the original hash list, the function does the following:
            - Converts the hash from hexadecimal format to an integer (similar to the process used in `create_binary_search_tree`). This involves decomposing the hash into bytes and then converting these bytes into an integer using `int.from_bytes`.
            - Uses the `search_node` method of the tree to check if this integer value is present in the tree.

Cuckoo Filters

Cuckoo filters are probabilistic data structures that allow efficient insertion and search operations, ideal for large datasets where false positives are acceptable but should be kept minimal. They use multiple hash functions to resolve collisions through a relocation method reminiscent of the cuckoo bird's behavior, displacing other elements as necessary.

In Cuckoo filters, the configuration of certain parameters is crucial to optimize the performance and effectiveness of the structure. The parameters capacity, fingerprint_size, and bucket_size play important roles in how the filter behaves.

Capacity: Defines the maximum number of elements the filter is designed to store. A higher capacity allows storing more elements without increasing the false positive rate or requiring too many relocations, which can help maintain the efficiency of insertion and search operations.

If the capacity is too low relative to the number of inserted elements, the filter may suffer from many relocations or even fail to insert new elements if the capacity limit is reached.

Fingerprint Size: The size of the fingerprint (in bits) used for each element inserted into the filter. The fingerprint is a summary derived from the element used instead of the full element for insertion and search operations.

A larger fingerprint size reduces the likelihood of false positives because it allows greater differentiation between stored elements. Increasing the fingerprint size raises the memory usage of the filter but improves accuracy. A smaller size reduces memory usage at the cost of increasing the false positive rate.

Bucket Size: Defines the number of fingerprints that each "bucket" of the filter can store. Each bucket is a basic storage unit within the filter structure.

A larger bucket size can reduce the need for relocations by providing more space for each set of fingerprints, which can be useful in environments with high insertion density.

While a larger bucket size can help manage collisions, it can also lead to inefficient memory usage if many buckets are not fully utilized.

For the search script using Cuckoo Filters, we have used the `cuckoofilter` library, which provides an efficient implementation of Cuckoo filters in Python with all the functionalities required for their probabilistic behavior. It can be installed using the command `pip install cuckoofilter`. We have also used the libraries mentioned in previous algorithms, `sys` to measure the data structure's size, comparing it with the size returned by the `pympler` library, and `time` to measure the search time.

Implemented Functions in the Script:

- **`load_hashes_from_file(file_path):`**
  - **Purpose:** Load hashes from a text file. This function is fundamental to start the process, providing the initial data that will be inserted into the filter.
  - **Functioning:**
    - Opens the specified file and reads each line.
    - Splits each line by the character `:` and takes the first part, ensuring only the hash is used.
    - Converts each hash to uppercase to standardize the format before insertion into the filter.
    - Returns a list of standardized hashes.
- **`create_cuckoo_filter(hashes, capacity, fingerprint_size):`**

    - **Purpose:** Initialize a Cuckoo filter and fill it with the provided hashes. This function configures the filter with a specific capacity and fingerprint size.
    - **Functioning:**
        - Creates an instance of `CuckooFilter` with the specified capacity and fingerprint size.
        - Converts each hexadecimal hash to bytes.
        - Inserts each converted hash into the filter.
        - Displays the number of inserted elements and returns the filter.
- **`check_hashes_in_cuckoo_filter(filter, hashes_to_check)`:**
    - **Purpose:** Verify the presence of hashes in the Cuckoo filter and measure search performance.
    - **Functioning:**
        - Converts each hash to check from hexadecimal to bytes.
        - Starts a timer to measure the duration of the verification process.
        - Verifies each converted hash in the filter.
        - Counts and collects the found hashes.
        - Calculates and displays the match percentage and the total operation time.
        - Returns the list of found hashes.
- **`calculate_false_positives(original_hashes, found_hashes)`:**
    - **Purpose:** Evaluate the accuracy of the filter by determining the number of true and false positives.
    - **Functioning:**
        - Compares the found hashes with the original hashes.
        - Counts the true and false positives based on whether the found hashes were actually in the original list.
        - Displays the number of true positives, false positives, and the total found hashes.
- **`main()`:**
    - **Purpose:** Orchestrate the complete program flow, from data loading to result evaluation.
    - **Functioning:**
        - Loads original hashes and the hashes to verify.
        - Experiments with different Cuckoo filter configurations, varying the capacity and fingerprint size.
        - For each configuration, creates a filter, verifies the hashes, and evaluates the results.
        - Ends by displaying a summary of the execution.