HW2 report

IMPORTANT, PLEASE READ

- you will need to update in the make file to the location of where openssl is located in the machine you're working with for the make file to be able to link stuff
  - the `CXX_FLAG` contains `-I/opt/homebrew/opt/openssl/include`
  - the linking step contains `-L/opt/homebrew/opt/openssl/lib -lcrypto`
- These are the required updates to make compilations work

Task 1

1. First, open and read each line of the CSV file, then store each column of that row into a `vector<vector<string>> data` using the function `read_csv_file`.
   - For example, if the input has `username1,password,salt`, the `data[i]` will contain `{username1, password, salt}`.
2. Then, open the output file `task1.csv` for writing the results.
3. Generate all valid strings of lengths 1-4 using a backtracking algorithm with recursion.
   - Generate these strings using `generate_string` and store all results in a `vector<string>`.
4. Create another `vector<string>` with the size of the input and an `unordered_map<string,vector<int>>` to store key pairs of the hashed passwords we're looking for and the index of their appearance from the input. Since the same hash may appear more than once, the value is a `vector<int>` and not `int`.
5. Iterate over the entire `data` vector and store all the hashed passwords and their indices (their position in `data`).
6. Iterate over all generated strings of lengths 1-4. For each string, compute its hash using the MD5 Algorithm (Evp_md5 1), then check if it exists in the map. If it does, iterate through the value (a vector of indices) and update the `index` of the result vector with the current string.

7. Iterate over the result vector. If the current index contains a password, write it to the output file along with its username. If not, write `FAIL` for that respective line.
8. Output the number of successfully fetched passwords (calculated while checking the hashes) and the elapsed time. Initialize the timer at the beginning of the function and calculate the difference at this point.

This method ensures only one pass over all the strings of lengths 1-4 without needing a rainbow table, which makes it an efficient brute force approach compared to attempting brute force for each password individually.

Task 2

The overall structure of this function is essentially the same as Task 1, except instead of generating strings of lengths 1-4 using recursion, we read `common_passwords.csv` and store all the passwords in a `vector<string>`.

1. First, open and read each line of the CSV file, storing each column of that row into a `vector<vector<string>> data` using the function `read_csv_file`.
    - For example, if the input has `username1,password,salt`, the `data[i]` will contain `{username1, password, salt}`.
2. Open the output file `task2.csv` for writing the results.
3. Read `common_passwords.csv` and store every password into a `vector<string>`.
4. Create another `vector<string>` with the size of the input and an `unordered_map<string,vector<int>>` to store the hashed passwords and their indices (their positions in `data`).
5. Iterate over the entire `data` vector and store all the hashed passwords and their indices.
6. Iterate over all the common passwords. For each password, compute its hash and check if it exists in the map. If it does, iterate through the value (a vector of indices) and update the `index` of the result vector with the current password.
7. Iterate over the result vector. If the current index contains a password, write it to the output file along with its username. If not, write `FAIL` for that respective

line.

8. Output the number of successfully fetched passwords and the elapsed time.

This method ensures only one pass over all the 10,000 common passwords, without needing a rainbow table, making it an efficient brute force method.

Task 3

1. First, open and read each line of the CSV file, storing each column of that row into a `vector<vector<string>> data` using the function `read_csv_file`.
    - For example, if the input has `username1,password,salt`, the `data[i]` will contain `{username1, password, salt}`.
2. Open the output file `task3.csv` for writing the results.
3. Read `common_passwords.csv` and store every password into a `vector<string>`.
4. Divide the 10,000 passwords into 4 chunks:
    - 0 to 2500, 2500 to 5000, 5000 to 7500, and 7500 to 10,000.
5. Create an `unordered_map<string,string>` where the hashed passwords are the keys and the plain-text passwords are the values (this forms the rainbow table).
6. Use threads to compute the hashes of their respective chunks of passwords and store them in the map.
    - Use a mutex to ensure safe writing to the map.
7. Wait for all threads to finish executing.
8. Loop over all the passwords. If the password exists in the map, output its plain-text value to the output file. If not, write `FAIL` for that line.
9. Output the number of successfully fetched passwords and the elapsed time.

Task 4

1. First, open and read each line of the CSV file, storing each column of that row into a `vector<vector<string>> data` using the function `read_csv_file`.

- For example, if the input has `username1,password,salt`, the `data[i]` will contain `{username1, password, salt}`.

2. Open the output file `task4.csv` for writing the results.
3. Read `common_passwords.csv` and store every password into a `vector<string>`.
4. Read all the salts from the `data` vector and store them in an `unordered_set<string>`.
   - Iterate through the entire `data` vector, reading `data[i][2]` (the salt).
5. Divide the 10,000 passwords into 4 chunks:
   - 0 to 2500, 2500 to 5000, 5000 to 7500, and 7500 to 10,000.
6. Create an `unordered_map<string,string>` where the hashed passwords are the keys and the plain-text passwords are the values (this forms the rainbow table).
7. Use threads to compute the hashes of their respective chunks of passwords, and for each password, compute the hash for each salt.
   - Use a mutex to ensure safe writing to the map.
8. Wait for all threads to finish executing.
9. Loop over all the passwords. If the password exists in the map, output its plain-text value to the output file. If not, write `FAIL` for that line.
10. Output the number of successfully fetched passwords and the elapsed time.

Note: I did not construct a rainbow table for all 10,000 common passwords combined with all salts, as this would result in 100 billion entries, requiring around 2.67-3 TB of storage or RAM usage if loaded into a map.

Task 5

1. First, open and read each line of the CSV file, storing each column of that row into a `vector<vector<string>> data` using the function `read_csv_file`.
   - For example, if the input has `username1,password,salt`, the `data[i]` will contain `{username1, password, salt}`.
2. Open the output file `task5.csv` for writing the results.

3. Read `common_passwords.csv` and store every password into a `vector<string>`.
4. Attempt to find all the valid passwords among those used in Task 4, as they should be the most common and will save computation time if we can crack all of them.
5. If there are still unmatched hashes, begin the longer computation.
6. Generate all possible length 1-4 digit combinations, storing them in an `unordered_set<string>`.
7. ~~Divide the 10,000 passwords into (amount of cores your CPU has) chunks:~~
    - I've found out that single thread will parsed the passwords way faster for some reason then having more threads in our current implementation.
    - In a single thread, it was able to compute all the stuff for "wrongpassword" in a mere 5 minutes but it took 50 minutes in a multithread environment with all cores being used.
        - I tried removing all the mutexes which didn't help despite nothing should be stopping the threads going full power.
        - I tried researching whether the underlying data structures have internal blocking, it doesn't
        - I tried to see if my MD5 Algorithm is thread safe (it is)
        - I am very clueless as to why that is as I would expect the performance to be greater but the single thread is working much faster so I will switch to that instead.
        - The only thing left I could suspect is issues regarding to cache hit / miss or locality issues, as many of the data structures are created dynamically in the thread. These memory are "shared" between all threads and not local to the thread like the Stack would. So the locality / cache ability is reduced comparing to a single thread which compute value that's already nearby.
    - Please continue this reading assuming that the multi thread no longer exist and it is just the 0→10,000 as 1 massive chunk.

8. Use threads to compute the hashes of their respective chunks, passing in the salts, 1-4 digit combinations, and the hashes we are trying to find.
    - For each password, generate all uppercase variants and store them in a set. Then, for each variant, generate character swap operations (e/E → 3, o/O → 0, t/T → 7)
        - these are all done with a recursion with backtrack idea, cannot be solve via DP as each cases are unique sub problem
    - Iterate over all strings in the set, append the 1-4 digit combinations to each string, and finally append each salt. Compute the hash.
    - If all passwords are found, terminate the threads early to save computation time.
    - Use a mutex to ensure safe writing to the map.
    - This process is heavily dependent on the amount of unique salts in the input file
    - every unique salts is a multiplication of time of using 1 singular salt value
        - For example if let say the password “abc123” took 17 seconds to calculate for all combination for 1 salt. Every unique salts * 17 seconds will be the amount of time spend on that 1 string with all its variations.
9. Wait for all threads to finish.
10. Iterate over the result vector. If the current index contains a password, write it to the output file along with its username. If not, write `FAIL` for that respective line.
11. Output the number of successfully fetched passwords and the elapsed time.

In summary, Task 5 builds on Task 4 by attempting to find a match using the rainbow table. If unsuccessful, it generates all possible combinations of each password, including character case and swaps, along with 1-4 digit combinations. Once all matches are found, threads are signaled to stop early, ensuring a single full pass of all combinations at most. Passwords with more more English alphabets take longer to compute.

At the current implementation, passwords toward the end of the chunk take longer to compute on average because all previous passwords need to be checked first.

Appendix

Evp_md5(3): EVP Digest Routines - Linux Man Page, linux.die.net/man/3/evp_md5. Accessed 1 Oct. 2024.