Fall 2022 ME/CS/ECE759 Final Project Report
University of Wisconsin-Madison

MD-5 Hash Attack

Shlok Sabarwal

December 14, 2022

Abstract

Digital passwords are some of the most critical information people possess. Exposure of passwords may lead to breach of private information about the victim – their banking information, personal conversations, records etc. Therefore, it is crucial to have strong, and secure passwords. This project attempts to quantify the validity of various strengths of passwords and brute-force approaches by calculating the time taken to match randomly generated passwords' MD-5 hash digest to the MD-5 hash of the legitimate password.

This project is entirely implemented using Nvidia's CUDA library to take advantage of faster operation speeds and parallel computing capabilities in GPUs. With aid of documentation on the MD5 hash algorithm, an implementation of the algorithm was constructed. Multiple forms of brute-force attacks were also designed to compare time taken to breach password integrity. For the implementation of parallel brute-force attacks, each thread was tasked with checking 500 passwords with each block containing 512 threads, allowing faster execution of the brute-force attack with parallel execution.

The results of the experimentation were very conclusive, with the non-parallel approach of brute force attacks taking roughly 1024 times longer than its parallel counterpart. Furthermore, as password strength increased, there was a 4x increase in the total time needed to decrypt the hash given, which is very strong evidence in support of the password strength guidelines.

Drawing from the results of the experimentation, it was concluded that passwords that include lowercase and uppercase letters and numbers take considerably longer to break compared to passwords that may only include lower case letters or even the combination of lowercase and uppercase letters. It was also noted that the parallelization of the brute-force attacks presented a significantly faster execution time. From this project thus, the importance of generating strong passwords to hinder brute-force attempts at finding passwords has been validated.

Link to repository: *https://git.doit.wisc.edu/SSABARWAL/md5-hash-attack*

Contents

1. General information

1. Home department: **Letters & Science – Computer Science Department**

2. Current status: **Undergrad**

3. Code Licensing: **I release the ME759 Final Project code as open source and under a BSD-3 license for unfettered use of it by any interested party.**

2. Problem statement

There are many guidelines available online that dictate rules to formulate strong passwords difficult to breach. As it may happen often with guidelines, they are not often followed by the demographic. Therefore, the goal of this project is to implement a brute force attack implementation for the MD5 hash algorithm and to improve the speed of the implementation by generating multiple hashes in parallel using CUDA. This implementation will be used to compare the time taken to correctly assess passwords of varying strengths to help visualize the importance of generating highly-secure passwords. **<u>I chose to work on the MD-5 attack default project templates provided.</u>**

The motivation for this project comes from my primarily held position at the university as systems administrator at the Simulations Based Engineering Laboratory. One of the biggest responsibilities that comes with my position is ensuring the digital security of workstations and nodes in-use at the lab. Recent exposure of an integral password in-use for multiple sudo accounts meant that many systems had to be given new passwords immediately. This accident made me re-evaluate steps taken to ensure the integrity of systems and the importance of choosing strong passwords. I believe this project will be an exciting opportunity for me to apply the lessons taught from the course while developing a better understanding of how guidelines for password security make it more difficult to breach said passwords.

3. Solution description

My project can be broken down into 2 major problems – MD5 hash implementation and passwords iterations. Thus, I will be splitting my solution description into two parts explaining both concepts.

<u>MD5 Hash Implementation</u>

The MD5 Hash implementation can be described in 3 small processes. The first step is defining the prerequisite variables, followed by processing the input password into the required format and finishing with the conversion of the processed password into the hash digest. Following is a detailed description of the three steps.

Defining needed variables

The MD5 Hash implementation begins by defining the **buffer array**. The buffer array consists of 4 unsigned integers (For the rest of the implementation, they will be referred to as A, B, C and D respectively) that will be modified and conjoined with each other to return the hash digest.

The implementation thus begins with the definition of buffers and certain constant variable arrays. These variable arrays include 3 arrays that are used to store information about constant values that will be added to the buffers, and one array labeled **S** that represents shifting values.

Lastly, dynamic char arrays labeled **bit_array** and **hash** are defined. It is important to note that dynamic char arrays were used instead of strings because CUDA does not provide support for strings. Hash is defined with a defined length of 32 since all MD5 hash digests are of 32 char len. To define bit_array's length, we first pre calculate the total number of 512 bit "chunks" in the original password. This number is then multiplied by 64 to get 64 bytes per chunk and the received number is determined to be bit_array's length.

If implemented correctly, **hash should have 32 chars** and **bit_array's length must be a multiple of 512.** Now, we are ready to begin pre-processing bit_array for further calculations.

Preprocessing bit_array

Official MD5 hash guidelines dictate that the preprocessed string must consist of the original password followed by a separator character, followed by continuation of 0s until the total memory occupied is 64 bits short of the next multiple of 512.

Finally, the last 64 bits must be fitted with a 64-bit representation of the length of password. Let us see how this string is prepared.

The first step of adding the original password to bit_array is implemented by running a simple for loop and copying values by the index. After which, the separator char is added to the current final index which is followed by another linear for loop to insert an appropriate amount of 0s into bit_array.

The conversion of the 32 bit length to a 64 bit answer is a little tricky. To do so, length is stored into a union structure (union structures may have multiple data members, but at a certain time only 1 data member's worth of memory is allocated. Thus, accessing other data members automatically divides stored data into chunks if necessary. This is a very interesting and useful concept!).

The 2nd data member of the union data structure is a char array with 4 elements. Copying each of these 4 elements sequentially into the end of bit_array helps us get the pre-processed string!

Processing the string

The bit_array is processed sequentially in chunks of 512 bits, according to the MD5 guidelines. Since we pre-calculated the number of 512 bit chunks, we do not have to do so again. For each iteration, 4 copies of the A, B, C, and D buffers are made and the 512 bit chunk of bit_array is copied into the union structure.

Following this, each chunk is processed a total of 64 times! This is accomplished with 2 for loops, with the outer loop being used to set the values of the addition matrices **M** and **T.**

Now, for the inner for loop which runs a total of 16 times, a unique index G is defined which allows us to access 1 of 16, 32-bit chunks of the part of bit_array being processed. Let us assume this value to be **word_chunk.** Then, one of 4 operations is performed on the buffer copies. These predetermined bit operations are described as follows:

1. **(B AND C) OR ((NOT B) AND C)**
2. **(B AND D) OR (NOT D AND C)**
3. **B XOR C XOR D**
4. **C XOR (B AND (NOT D))**

These values are stored in an unsigned integer F, **F = F + A + K[**inner_index + (16*outer_index)**]** + **word_chunk.** Finally, bitwise left rotation is applied to F, shifting it by a certain number stored in the the shifting matrix **S**, the index of S being calculated by iteration indexes of the inner and outer loops.

The last step of this processing involves the right shift of buffer values. (**A = D, D = C, C = B** and **B = B + F**). This process continues for a total of 64 times! After the buffer copies have undergone the modifications, their values are added back into the original buffer values.

After multiple iterations of these operations, the final values stored in the buffers are the wanted digest values! These values are converted to hexadecimals and concatenated into **hash.** After this operation, **hash** now is equivalent to the hash digest of the given password!

Random Password Iterations

The 2nd biggest hurdle to overcome for the completion of this project was the determination of passwords given their index in the list of all possible password combinations! This task was made quite difficult by the fact that an index that has a password of length, say 5 will still have to account for passwords generated of lesser lengths. This made things complicated but nevertheless a robust solution was developed to tackle this problem in a memory-efficient manner!

This process is split into 2 tasks – Calculating length of password at index and re-adjusting index for all passwords of given length AND Extrapolation of password from re-adjusted index and known length.
Calculating length and adjusting index

Both these problems are addressed in one **__device__** function labeled **calcWordLength**. CalcWordLength accepts the index of the password as a parameter. Inside calcWordLength, a for loop is run till index is greater than $(N\_CHARS)^{\wedge length}$ where N_CHARS is the number of total possible characters.

For every iteration, the power value of $(N\_CHARS)^{\wedge length}$ is subtracted from index and the length variable is incremented by 1. After the while loop exits, length is returned, and index is re-adjusted from the subtraction operations.

Extrapolating password iteration from index

For the calculation of the word in the **__device__** function labeled **calculateWord,** the parameters of a char array labeled **placeholder** and the adjusted index are accepted. Before beginning the processing of the placeholder array, it is filled with the beginning most char for all cases: 'a'.

After placeholder has been processed for further processing, it is passed through a for loop which modifies each character of placeholder. The logic behind this process is simple: to understand how much increment for each index's character needs to occur, we simply need to calculate the total possible words able to be generated after the index. If this number is greater than the **index** variable, then we move to the next index character. If not, we increment the character and continue the process until **index** variable becomes smaller than the number of possibly generatable passwords after.

After this process is repeated for each character in **placeholder,** the function comes to an end.

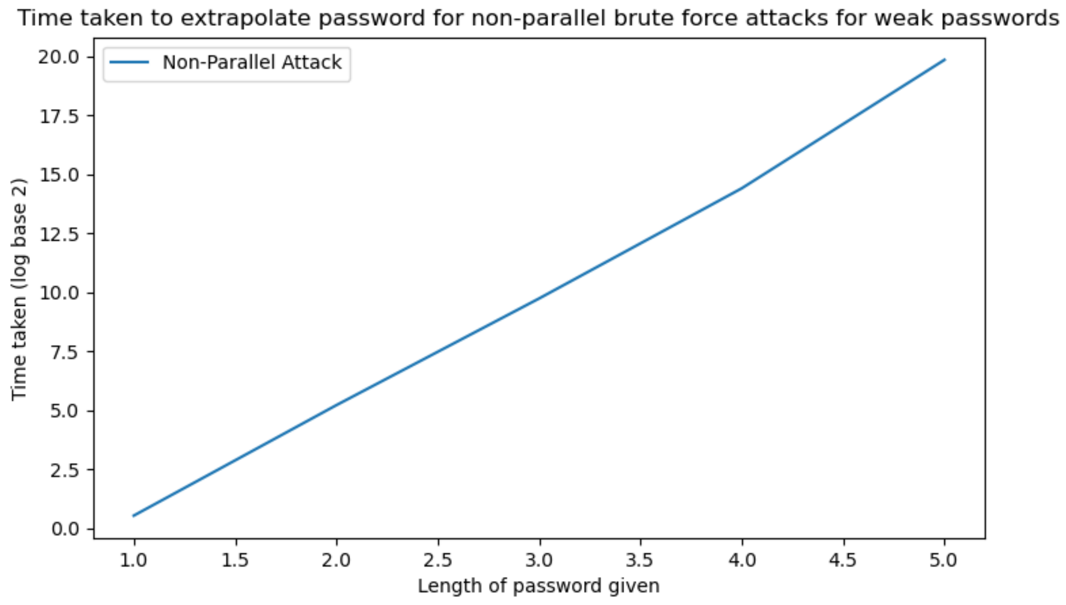
Finally, after having established these two concepts, further implementation is rudimentary. For each thread, we understand that 500 words must be processed. We can calculate the index value from the thread and block id values. From the index value calculated, we know that the beginning word is **calculateWord** (index).

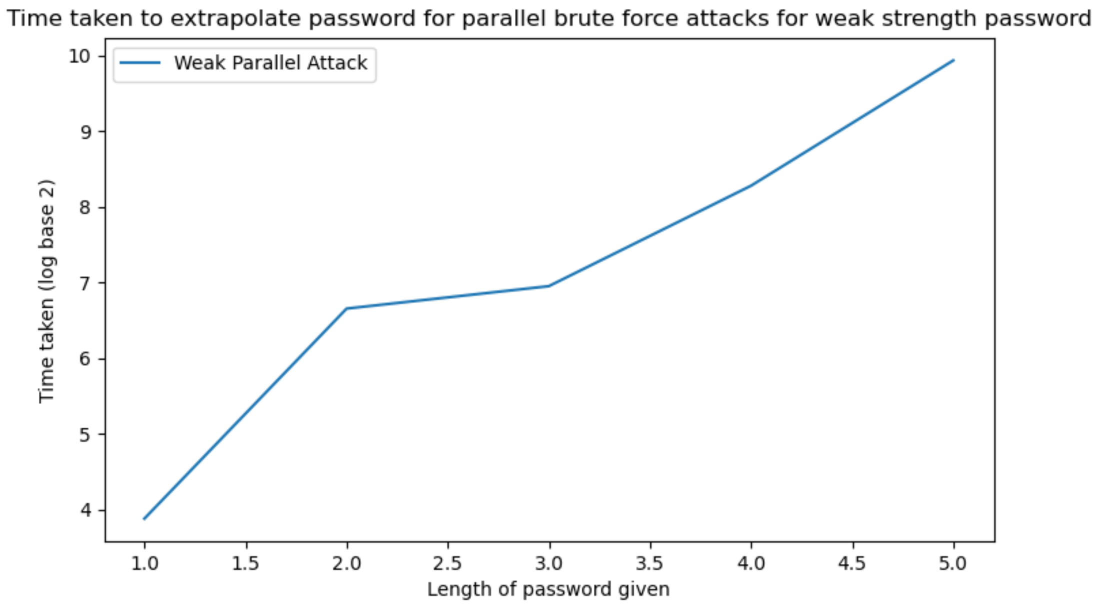
After this, we can simply run a for loop for 500 iterations, increasing to the next password iteration with every loop. Simply calling the hash generation function and comparing the hash generated to the original hash value, we can process all 500 words for each thread and find if one of them is the password!

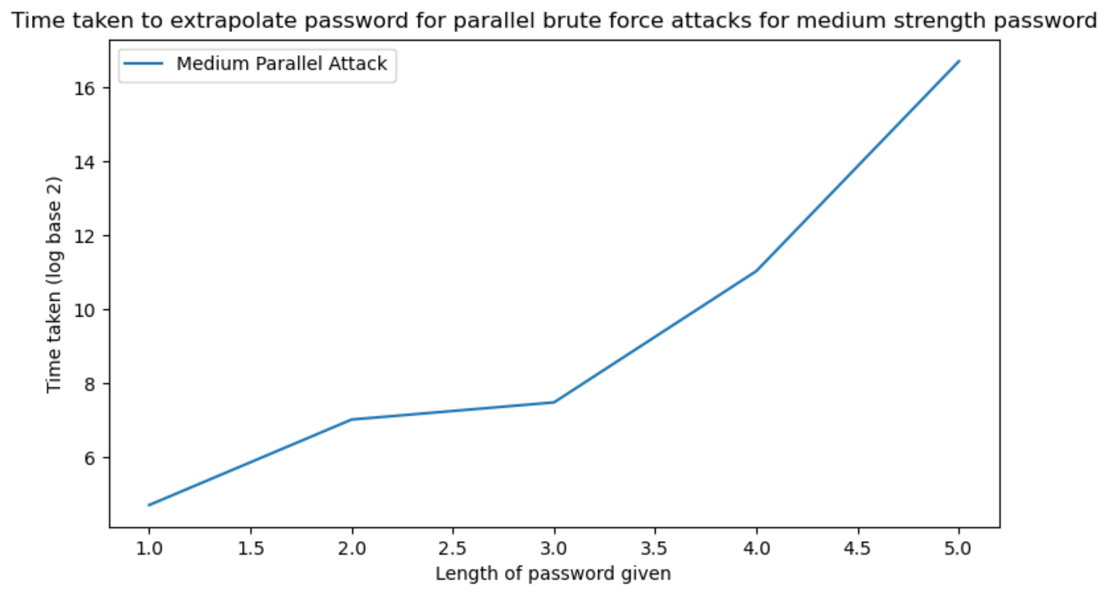
All the processes – in coordination with each other – running parallel across thousands of threads allow a robust parallel brute-force attack to help us retrieve the password from the hash value given!

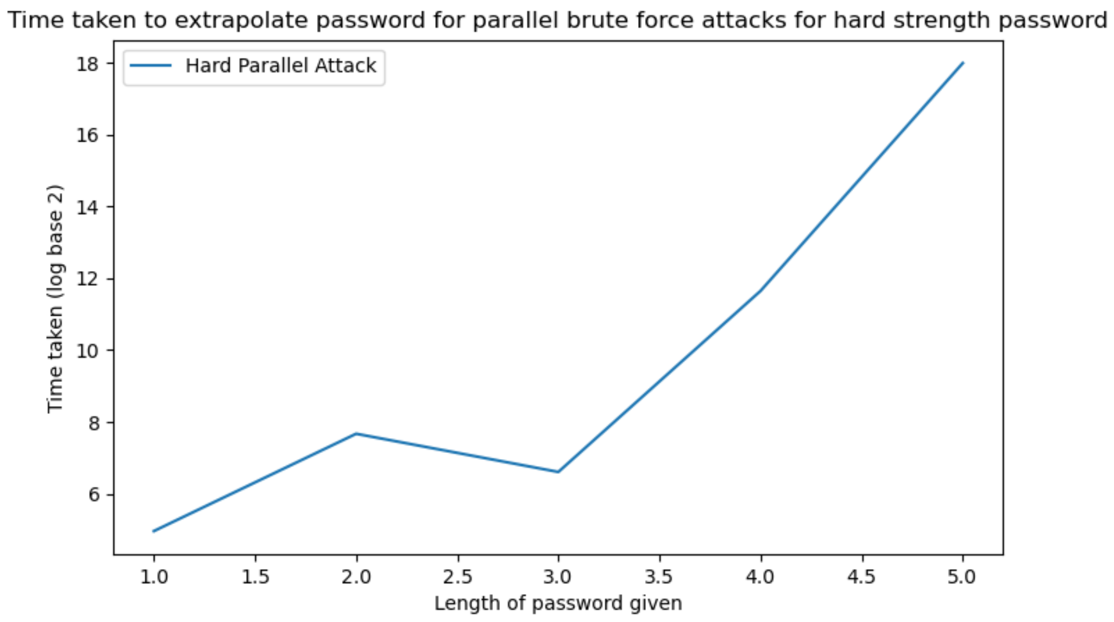
4. Overview of results. Demonstration of your project

From testing out the 4 different parallel brute force attack methods – non-parallel, weak, medium and hard – strong evidence was found in regards to how parallel computation helps brute force attacks run exponentially faster and on how much of a difference is generated between different password strengths and lengths quantified by the total time taken by the brute force attacks to find the password from the hash. Presented below are the plots generated from each attack for varying password lengths.









From the plots presented above, it is clearly visible that there are stark differences in the total time taken to calculate the passwords of varying strengths. This is very indicative of the fact that the guidelines that dictate password strength rules are very accurate and must be followed!

5. Deliverables:

Following are the list of official files that are available on the GitLab repository with descriptions of each file:

1. **main.cu:** Responsible for accepting custom password from user and calling custom brute-force implementations. After execution of the kernel, the total time taken by the program (in ms) and the original password is outputted to the display.
2. **md5.cu/md5.cuh:** Header and code files for base implementation of MD5 hash algorithm kernel. Accepts char arrays for original password and hash digest. After successful implementation, the generated hash digest is stored in the hash variable.
3. **md5-non-parallel.cu/md5-non-parallel.cuh:** Header and code files for non-parallel brute force attack. This implementation assumes that only 1 thread has been allotted to the kernel. Due to the long execution times for this approach, the only passwords checked for are passwords consisting of only lowercase letters.
4. **md5-weak-parallel.cu/md5-weak-parallel.cuh:** Header and code files for parallel brute force attack for weak-strength passwords. Weak passwords are all passwords that only include lowercase letters. This implementation permits each thread to check 500 passwords and relies on parallelization for faster execution than a non-parallel approach.
5. **md5-medium-parallel.cu/md5-medium-parallel.cuh:** Header and code files for parallel brute force attack for medium-strength passwords. Medium strength passwords are all passwords that include lowercase and uppercase letters. This implementation permits each thread to check 500 passwords and relies on parallelization for faster execution than a non-parallel approach.
6. **md5-hard-parallel.cu/md5-hard-parallel.cuh:** Header and code files for parallel brute force attack for strong-strength passwords. Strong strength passwords are all passwords that include lowercase and uppercase letters and digits. This implementation permits each thread to check 500 passwords and relies on parallelization for faster execution than a non-parallel approach.

Compilation Instructions

1. In main.cu, replace the **T_CHAR** variable with the number of possible password characters. (weak=26, medium=52, hard=62) depending on password strength to test.
2. Replace line 6 (**#include "md5-hard.cuh"**) to wanted brute-force implementation header file.
3. Run the following compilation command in terminal: **nvcc main.cu md5.cu <brute-force-implementation.cu> -Xcompiler -O3 -Xcompiler -Wall -Xptxas -O3 -std c++17 -o main**
4. Run the following command to execute brute force attack: **./main <password-to-find>**

6. Conclusions and Future Work

This was a very exhilarating project to have had the opportunity to work on! I would love to go back to the original purpose of this project, which was to understand the importance of having strong and secure passwords. Through personally writing brute-force implementations in attempts to crack passwords of varying strengths, I was given the unique opportunity to understand just how complicated it can be to extract a password from a hash! For the scope of this project, I chose to work only with passwords of maximum length of 5, but with each extra character the total possible combinations increase by $(62)^{\wedge(len+1)}$. From the presented plots, I do not need to dive into just how much longer it may take to break a password with each extra character! From this experience, I now have a truly greater appreciation for strong password guidelines and the people working on generating complex hashing algorithms!

Following is the list of highlights in my project and the lessons I learnt from them:

1. **Bit Manipulation** – This project involved work with individual bits for each character array for conversions. Previously, I was not quite so well-versed with bit operations but this project helped me understand the fundamentals of it and I will make sure to use this information in the future! It was quite difficult for me to fully understand the necessary bit operations and manipulations to get a successful hash generation, but I am glad to have gone through that!

2. **Permutations & Combinations** – Interestingly so, this project involved work with a lot of fundamentals regarding permutations when generating the list of all possible passwords. One previous approach I considered to generate passwords was to store previous passwords in a list and add new chars to the beginning of each of them. This approach was too memory-inefficient however, and this forced me to think outside of the box. Ultimately, after multiple rounds of trial-and-error, I was able to successfully extract the iteration of a password given its index in the list of all possible passwords. This was much more challenging than expected, and this has given me renewed interest in Mathematics and Probabilities!

3. **Memory Allocation and Deallocation** – When working with thousands of threads across multiple blocks, I had to be quite mindful about memory allocation and storing constant values in sections of memory with low latencies. My implementation of the brute-force attacks failed many times due to improper management of dynamically created arrays. Through this, I understood the importance of keeping track of dynamic variables and ensuring that all memory is deallocated after its work is complete. This is, I'm sure, a very useful ability for a programmer.

Finally, this project put ME 759's CUDA teachings to excellent use. For this project, I had to ensure that workload was being split evenly across threads. I also had to work extensively with kernel functions, device functions and the concepts of shared and constant memories. I would not have been able to complete this project without the CUDA lessons taught in ME 759.

References

1. *Wikipedia, Independent Writer. "**MD5.**" Wikipedia, Wikimedia Foundation, 6 Dec. 2022, https://en.wikipedia.org/wiki/MD5*

2. *M, Shruti. "**MD5 Hash Algorithm in Cryptography: Here's Everything You Should Know.**" Simplilearn.com, Simplilearn, 21 July 2022, https://www.simplilearn.com/tutorials/cyber-security-tutorial/md5-algorithm.*