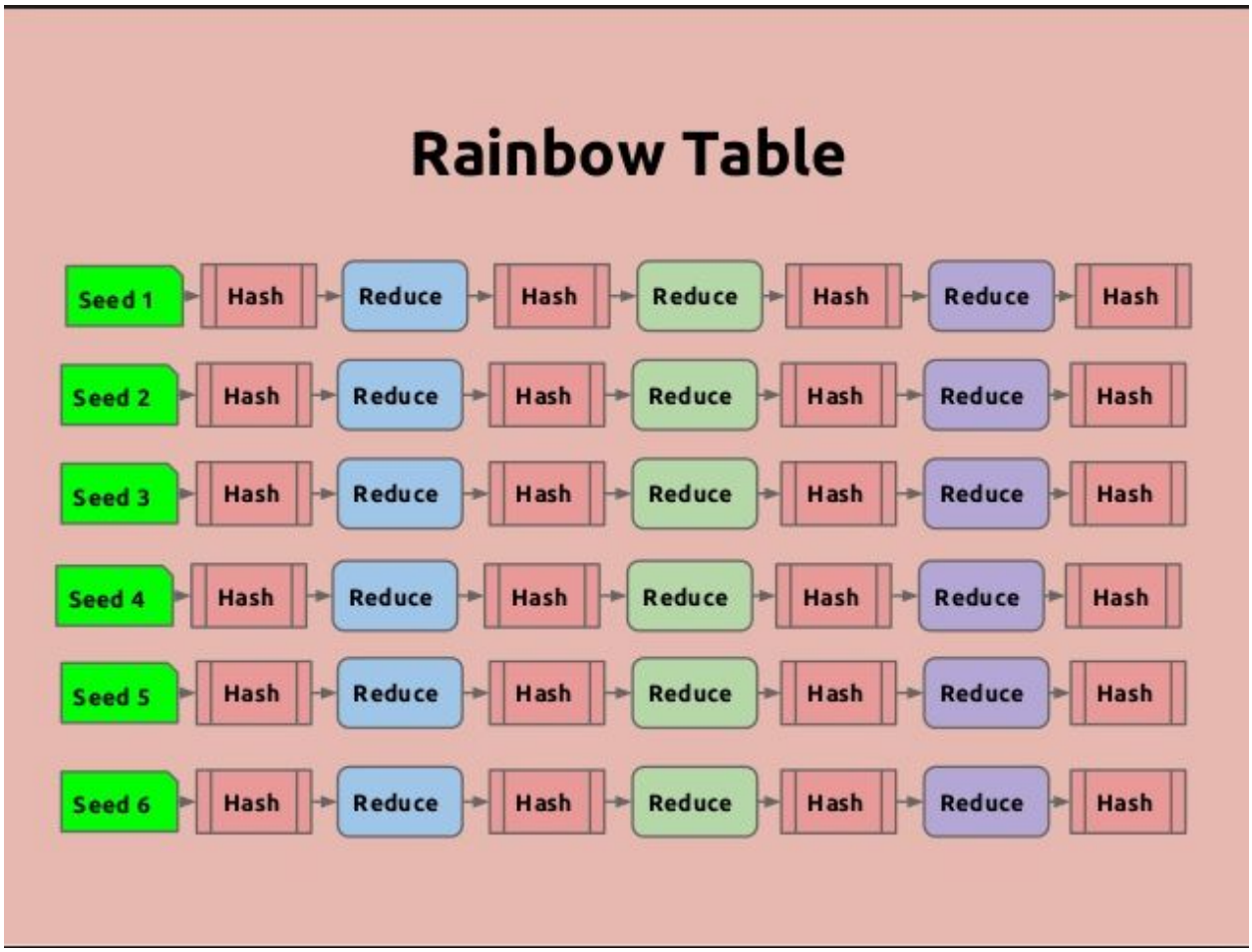
Rainbow Tables: By Matthew Fevold



(image source: http://www.slideshare.net/VahidSaffarian/what-is-a-rainbow-table?next_slideshow=25)

Matthew Fevold
CSCI 415
12/16/16

Rainbow Tables

Abstract:

Rainbow tables are an implementation for retrieving a plaintext password from given a hash. It takes a combined approach of using part of a lookup table (aka dictionary attack) and part of a brute force by doing most of the calculations before hand but still doing calculations when trying to crack a password hash. It is also called the time-memory tradeoff because of this approach. It is called this because brute force takes too much time and lookup tables take too much memory. The bottleneck for execution time for this approach is doing the calculations before hand. I chose to try and speed up this process by implementing multithreading (pthreads) on this portion of the program. After the program was written both serially and with multithreading I saw an increase of speed on 4 threads of 123% from the serial implementation.

Report:

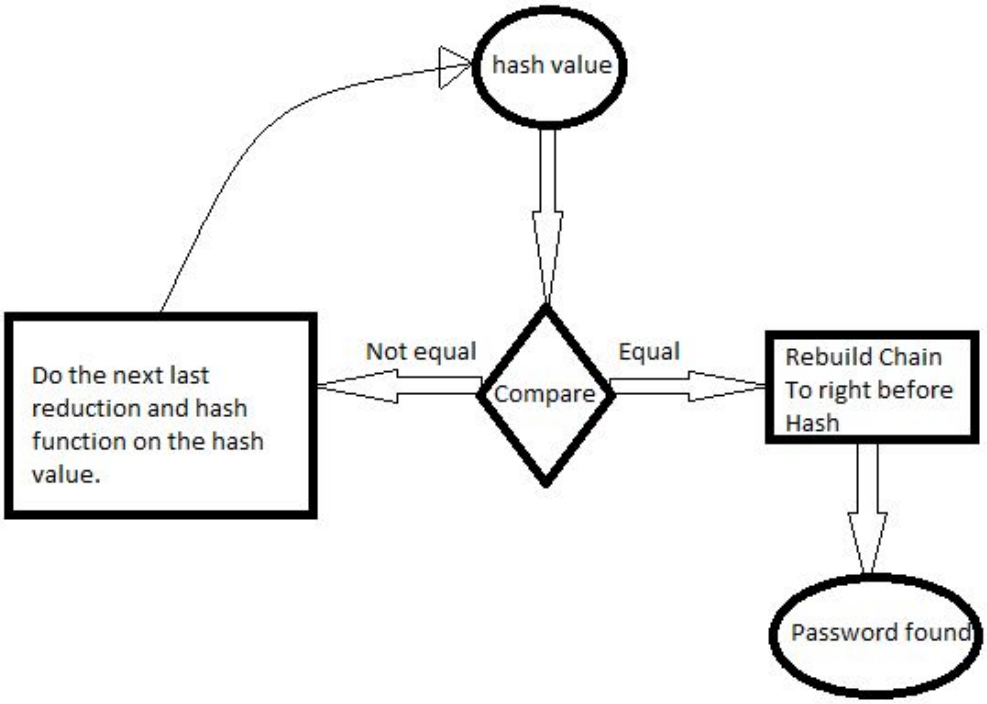
Background:

Understanding how rainbow tables work to crack password on a deep level is not necessary for this report, and you may read through my code which the comments should speak for themselves on a lot of the technical implementation of it. There are two programs with this project. The part that creates the rainbow table and the part that uses it. In the following two paragraphs I try to describe exactly how each part works as simply and without losing as much information as possible.

The high level of creating the rainbow table is that you precompute *rainbow chains* by taking an arbitrary plaintext (of the type that you want to solve for), hashing it (with the function you want to create a table for - MD5/SHA-256… I did NTLM), using a unique reduction function (***which does not reverse the hash but produces a new plaintext***) and repeating this process N times with the new plaintext each time. You store the first plaintext and the final hash for each *chain* (because the previous process described is for just one rainbow chain) you create and none of the other data. Because you know which functions were used to create the chain you can easily recreate this chain on demand which allows you to only store a fraction of the data. The beautiful part of this process is that once you have your *Rainbow table* computed it is not computationally intensive to use this table to break hashes.

The high level of using a rainbow table to crack a password is that given the value of a password hash and your Rainbow table you do the following:(look at graph 1) you compare your

hash to the final hashes in your rainbow table, if you get a match you rebuild the chain until the plaintext right before that hash - that is your plaintext and you are done, if you don't you apply the last reduction function and hash again and do that same comparison. Do this recursively until you find your password or you go through all the reduction functions in your table.



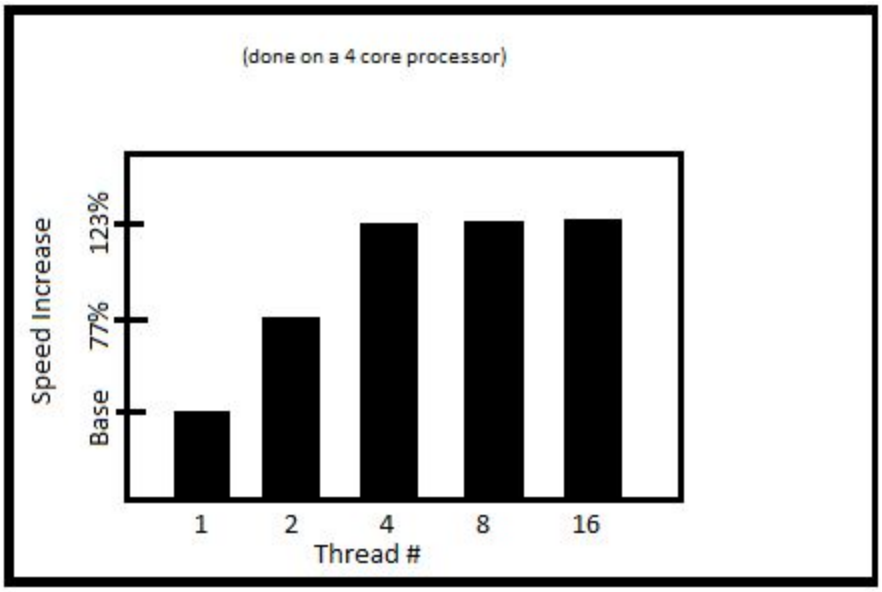
Graph 1

What I did:

In c++, from scratch I built a program that attempts to do what is described in the background section. I decided on trying to break windows NTLM hashes for two reasons, I found some code already written in c++ (see sources) that did it and that NTLM is actually used a lot which makes it slightly more practical. The rest of the program I did build myself but it is unwise to attempt to create your own hashing algorithms so I left that for the professionals. The next hurdle I faced was the reduction function. I decided early on that this should just be proof of concept so to keep things simple I just only tried to crack numerical passwords - this simplified my code tremendously. For my reduction function what I did was turn all letters in the hash into 0's and then based on which step it was chose a substring of the hash. Once the reduction function and the hash function were built I made 2 copies of the program, one to create the chains and one to use the chains. Creating a rainbow table is as simple as taking a random password, hashing it and reducing it N times and writing just the first plaintext and the final hash to a file which was read in by the second program. The second program uses these same hash and reduce functions in the reverse order, comparing the hash it's trying to break with all of the endpoints in the rainbow table. If the comparison matches the endpoints it rebuilds that chain until the point right before that hash and returns that plaintext. If it doesn't it applies the final

reduction function on its hash, hashes that new plaintext and then compares again. It just zips through the entire chain doing this process until it gets to the end or finds a plaintext.

When it came to deciding how to improve the performance of this program it was clear that creating the table was taking 99% of the time so this is where I focused my efforts. Each chain relies on each function feeding the next so that process was not a good candidate for parallelization. But each chain does not depend on one another. So I can have multiple threads each creating a lot of chains. The part that is a little tricky is writing to the file to be read in later, I didn't want any race conditions so I used a mutex to lock this file for writing for each thread. Once I completed the parallel implementation as you can see in the following graph (graph 2), my speed increased by about **123%** with 4+ threads over the serial implementation. This was speed was measured with 10000 length chains and doing 2000 of these chains on a 4 core processor (about 106 seconds baseline, 43 seconds 4+ threads).



Graph 2 (10,000,000 calculations - chain length 10000, 2000 of them)

Issues:

My program wasn't perfect, creating the rainbow tables works as expected multithreaded but when it comes to using these tables I am still at the proof of concept stage. If the chain length is 1 then I can find the password (as it is a simple lookup table at that point) but I am having issues with my logic when I increase the length at all. This is most likely a coding error but this wasn't the "main" focus of the assignment which was to parallelize a difficult function. My reduction function is also not the most sophisticated ever but it is as complex as it needs to be by reducing the chance for chains to merge (for two plaintexts to produce the same hash - which is bad)

Conclusion:

In conclusion I sunk considerable time into this project and am proud of it. I did not take an existing algorithm and make it parallel, I took an existing concept and made it parallel. By myself. This proved to be difficult but I enjoyed the challenge. I saw some decent speed increases which I feel met the merit of this project. I will continue to develop it and to a fully functional state that uses multiple different hashing algorithms and to include all 96 unique keyboard characters.

Sources:

https://www.codeproject.com/articles/328761/ntlm-hash-generator

http://www.slideshare.net/VahidSaffarian/what-is-a-rainbow-table?next_slideshow=25