University of Illinois at Urbana-Champaign

ECE408 – Applied Parallel Programming

Final Project – GPU Cryptohash Recovery

December 18, 2013

Gaurav Lahoti

Michael Chmiel

Uttaresh Mehta

Table of Contents

Introduction

For our project, we wanted to choose a project that would illustrate best the task of "programming massively parallel processors". Because of our interest in computer security and the exciting nature of the problem, we decided to look into cryptohash recovery. Computer systems need to store user passwords somewhere. However, storing passwords in plaintext is never recommended, as an attacker could simply find the password file and thus have access to all passwords stored on the machine! Therefore, passwords are first encrypted using strong cryptohash functions. Given a generated cryptohash, it is hard to crack the hash to generate the original plaintext. Therefore they are the natural solution to password storage. Though there are tools to recover passwords from cryptohashes, most of them are sequential and CPU based. As passwords get more complex, these CPU based solutions fail to recover passwords in a reasonable amount of time. Indeed it can take hours or days to recover a password as simple as 'john78' on some implementations. We decided to make such a password recovery tool for the MD5 hash using CUDA to see if we could do any better.

Problem Description

The reason cryptohashes work is that even though the hash generation functions are well known, it is not possible to reverse engineer an effective cryptohash; i.e. given a hashing function h and input x such that generated cryptohash c = h(x), even if the h and c are know, we cannot solve for x in a reasonable amount of time. Then the only way to obtain x would be to keep guessing what x is until our cryptohash function spits out the correct value for c. As with all hashing functions, there can of course be collisions. However, cryptohashes are built to avoid collisions as much as possible. For the purposes of this project, they are not a concern.

Given a cryptohashed value, if we wanted to recover the password, we would have to try every possible permutation of characters until the correct password was generated. For instance, if we wanted to crack a password that is 8 characters or less, there would be about 2.2x1014 possible plaintexts – assuming that the character set includes lowercase a-z, uppercase A-Z, and digits 0-9 for a total of 62 characters. To go through all of these permutations would take days on a CPU. Using CUDA, we can bring this time down to a few hours by moving the hash generation to the Tesla GPU provided on the GEM cluster. That's a big leap, but we think we can do better than this crude form of brute forcing.

Implementation

The goal of our code is, given a target MD5 hashed value, find its original source plaintext value, i.e. the password. Please remember that we have developed code for doing this both via CPU and GPU for performance testing. However, we are discussing the GPU version here.

MD5 Cryptohash Generation

As stated in the Problem Description, it is not possible to reverse engineer the original password given its MD5 hash. The only way to find the original password is to go through as many plaintexts as possible and see if any of them generate our target hash. Clearly to do this efficiently, we need a fast MD5 generation function. Since we have to check so many plaintexts in bulk, it makes sense to generate MD5 hashed values on the GPU – and this is exactly what we do.

Before we can send an input string to the MD5 generation function, we must preprocess it according to the input guidelines for the MD5 format. All inputs to the function must be 64 bytes long. For the purposes of this project, we assume that all passwords are at most 55 characters long – true for ~99.998% of passwords. The next bit right after the plaintext should then be set to 1. The last 8 bytes should include the length of the plaintext in 64-bit unsigned int. All remaining bits should be set to 0. We do this preprocessing inside the GPU kernel.

After this preprocessing is done, the function to generate MD5 hashes for all values is called and the resulting hashed values are compared with the target hash to see if there is a match. If there is a match, then we have found the password. If there is no match, we keep on trying for more input values until the solution is found or until all included permutations have been exhausted.

Dictionary Attack

A Simple Dictionary Attack

Since we are recovering passwords for humans, it is very unlikely that someone's password will be "x8Z1%cD". Most passwords make sense to its creator. Our grandparents' passwords range from "1945" to "mittens", while our more sophisticated parents use words like "Password1". Even the majority of (non-CS) kids from the age of the Internet hardly ever go beyond "you=n00b". There are already dictionaries of commonly used passwords out there that account for about 90% of passwords used today! Consequently, if we were to only try out words

from these dictionaries, we could guess 90% of passwords in O(*n*), where *n* is the number of passwords in the dictionary. Even for a dictionary containing a million words, this would not take more than a second or two on a modern GPU. So we decided to incorporate a dictionary attack in our implementation. In fact, before anything else, we simply generate MD5 hashes for each word in the provided dictionary in parallel and see if any of these matches our target hash. Our dictionary contains over 750K possible passwords, including the 10,000 most common passwords, passwords used by other password recovery tools, leaked passwords from well-known websites, and a dictionary of 300K words from the English language.

Advanced Dictionary Manipulation: Mutating the Dictionary

What happens if your password is in the 10% that isn't in the available dictionary? The fact is that even in that 10%, the vast majority of passwords will still be human-readable. For example, many websites require users to include uppercase letters or numbers in their passwords. Fortunately for us, most people still use predictable techniques to circumvent this hassle. So "patel" becomes "Patel" or "patel1", and "hajj" becomes "hajj2013" or "hajJ". We realized that we could generate these passwords by mutating the contents of the original password dictionaries using certain rules. For example, add a year to the end of the password or capitalize the first letter. So after the simple dictionary attack if we still haven't recovered the password, we start using this more advanced dictionary attack, which could still recover most remaining passwords in C*O(n), where C is the number of manipulation rules applied.

Brute Force Method

Of course, there is that 4-5% of passwords that really are difficult to find using a dictionary attack. Unfortunately there is no alternative here but brute force. As long as the password is under 7 characters and includes only alphanumeric characters, using the GPU, we can still recover the password in a matter of minutes. Please note that due to the 5-minute restriction on the GEM cluster, we have decided to only use lowercase letters. This can easily be changed to include more characters.

In our implementation, we try every password combination possible up to a given length. The basic algorithm involves launching a separate kernel for each possible password length. So, if we want to find a password of length 8, then we launch 8 kernels (1 letter passwords, 2 letter passwords, and so on up to 8 letter passwords). So, for instance, if we are trying to find a 4 letter password, then there would be 26^4 combinations. We divide up the work by the number of

threads used. If we are using 1024 threads, then there are $\frac{\lceil 264 \rceil}{1024}$ = 447 possible password plaintexts to check per thread. So thread 0 checks plaintexts 0 through 446, thread 1 checks plaintexts 447 through 893, etc. until all plaintexts have been checked.

Our brute force code has been optimized for performance. No memory access is required and there is minimal control divergence. Every thread knows how many plaintexts it needs to generate using its thread and block index. To reduce divergence, for loops in the code have the same conditions. One for loop goes over the number of words_per_thread, which is constant across all threads. Another goes over the length of the password to check – also constant across threads. There will however be one divergent warp because of the initial boundary check; it will be the warp that has some threads check the last possible combinations of passwords. Some thread indices will be out of bounds.

The worst situation is a password like "A8@mgZ_1>0x21" – incidentally one of our group member's older, now unused passwords. Unfortunately, there are some passwords like this out there (~1%) that really have no visible pattern attached to them at all. They are long, and use all usable ASCII or Unicode characters randomly. Our application does not claim to solve these, as it would take trillions of years to crack such a password on a modern GPU system.

Results

Overview

We were successfully able to crack all test passwords in a very short amount of time. Below is a table for password recovery times for given passwords. Both the CPU and GPU versions were developed by us.

Password	CPU Recovery Time (s)	GPU Recovery Time (s)
johnson59	17.016	1.085
Lyrics	0.458	0.445
recoil7	2.553	0.463
josh1993	48.854	0.233
john	0.211	0.786
bkso	60.189	1.321

As you can observe, the GPU implementation recovers passwords faster than the CPU implementation for most passwords. The reason the password "john" takes longer on the GPU is because it exists in the dictionary and has very quick recovery time. Most of the time is for the overhead involved in copying over the data to the GPU. The same can be said of "Lyrics".

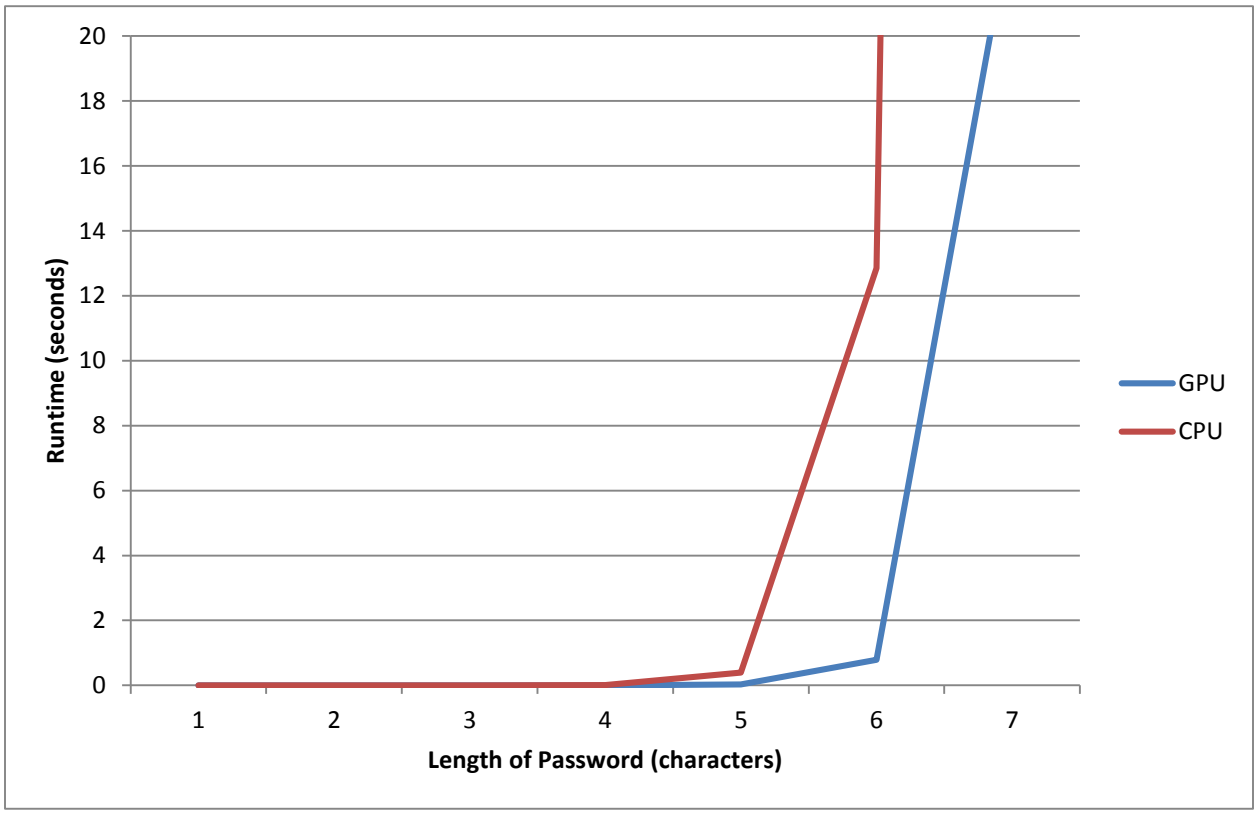
In addition, were able to achieve MD5 generation times on the GPU that are very close to the professional security industry standard. We were able to generate 1363M MD5 hashes per second, compared to the industry standard of 1392M/s on the Tesla M2050 – which is actually faster than what is provided on the GEM cluster.

Brute Force Performance

Comparing the results of the GPU code versus the same code written for use on a CPU, the GPU has the expected faster run time for words over 2 characters long. Here is a graph of the GPU runtimes vs. the CPU run times plotted against the number of characters in the found password.

	1 Letter	2 Letters	3 Letters	4 Letters	5 Letters	6 Letters	7 Letters
GPU	0.000055	0.000088	0.000154	0.000917	0.025313	0.787139	23.7312
CPU	0.000002	0.000009	0.000316	0.011767	0.393745	12.8542	257.992

**All times in seconds*



However, it's hard to show the increased performance in the GPU because I cannot run tests on passwords over 7 letters. These tests time out on the GEM cluster that has a maximum ~5 minute run time.

Challenges

MD5 Generation

The biggest challenge to MD5 hash generation was warp divergence during padding. Every thread would get a different string to hash during the pre-processing operation. If their sizes vary too much, there would be extreme warp divergence in the for loop that iterates to the end of the string. For example, if there are three original plaintexts of lengths 2, 8, and 30 respectively, all three will execute separately on the GPU. Now imagine if 32 consecutive threads all have different lengths. This would seriously undermine the maximum potential speed of the GPU. Therefore it was decided that the MD5 generation function would only be given input plaintexts that were as close to each other in length as possible. This was achieved by sorting the input dictionary by length, and by ensuring we only generate strings of the same length with the brute force technique. Please note that since we use the same dictionary every time the program is run, the dictionary is simply sorted once before compilation and reused.

Dictionary Manipulation

As with MD5 hash generation, since dictionary manipulation requires plenty of string manipulations, there would be plenty of warp divergence if the length of the strings was different. Moreover, since there are so many different rules to apply – 250 to be precise, and since some rules change the length of the string by a variable size, we would have even more warp divergence. For example, one rule might be to capitalize the first letter of the original plaintext, while another might be appending "0000" to the string. Therefore, in addition to using a sorted original dictionary as input, we decided to apply the same rule to all strings in all threads at a time. This has the added benefit that since some patterns occur more frequently than others, applying these rules sequentially lets us try out more probable password permutations first. For example, capitalizing the first letter is very common and should be one of the first rules applied to all values in the original dictionary. Therefore, by selecting the same rule in all threads executing in the warp and by making sure the lengths of all input plaintexts in the warp are close to each other, we significantly reduce divergence and increase performance.

Brute Force

For the brute force method, the main challenge was figuring out how to generate all possible passwords to try. We could create all possible passwords, but transferring to GPU memory and accessing them would just take too long. So, we had the idea to use numbers as indices into which passwords to test. The question then was how to get the nth password from the number n. After a little thought, we figured we could convert the n number into the base that is the number of characters that can be in our password. We convert this number to a readable ASCII string by treating it as a number with the base of the number of characters possible (base 26 for lowercase letters) and adding the ASCII value for 'a' to each character. For instance, $(446)_{10} \rightarrow (00H4)_{26} \rightarrow$ "aare". Using this, the password generation becomes like this: aaa, aab, aac…baa, bab, bac…zzy, zzx, zzz.

Future Work

There are a number of improvements we would like to make for this program if we were to actually release it to the public. Firstly, we only perform MD5 hash recovery at the moment. However, there are other cryptohashes out there. We would like to implement the solution for as many of these hashing algorithms as possible.

We would also like to increase the number of rules used in the advanced dictionary attacks. Currently, our implementation does not use letter-to-symbol substitutions. For example, batcat → b@tcat or Michael → M1chael. This is because a) not every letter has a commonly used symbol substitute (e.g. W), and some letters have more than one (e.g. i → ! or i → 1). Accounting for all this would create immense control divergence. It may just be easier to selectively do these on the CPU. Or maybe we can find a way to do this on the GPU and avoid control divergence altogether. Another nice feature would be to use combinations of known passwords and then apply rules. For example, *noobz* and *pwned* can be merged and manipulated to create the password *n00bzpwned*.

For the brute force technique, we would have liked to experiment with splitting up larger kernels into smaller kernels. Our current implementation requires 64-bit unsigned integers for generating all possible base-26 numbers. Unfortunately, there is a division involved in our kernel, which takes twice the time for 64-bits than for 32-bits. One possible way to reduce this time would be to go back to an unsigned 32-bit integer and only giving 2^{32} words per kernel. This may or may not improve overall runtime, since this require more kernel overhead and may

not be very scalable. The possible combinations of passwords grows very fast past 7 or 8 characters, but it would be interesting to see the actual outcome using working code.

Conclusion

Overall the project was a success. We were able to create an application that can recover most passwords from an MD5-hashed value with ease. Our hash generation rate is extremely close to the professional standard, and we can recover not only most passwords today, but also passwords that will be used in the future that follow well-known patterns. Even for completely random passwords, we were successfully able to come up with a brute force solution that can recover passwords up to a certain length in a reasonable amount of time. Our only regret is that we were not eligible for the iPad.

References

"MD5." *Wikipedia*. N.p., 12 June 2013. Web. <http://en.wikipedia.org/wiki/MD5>.

"Hashcat: Advanced Password Recovery." Hashcat. <http://hashcat.net/hashcat/>.

Golubev, Ivan. "GPU Speed Estimations for MD5/SHA1/Office 2007/WPA/WinZip/SL3."*Ivan Golubev's Lair*. <http://golubev.com/gpuest.htm>.