MD2 Hash Function with Brute Force Password Breaking on the GPU

Samuel Brooks
Department of Comupter Science
Virginia Commonwealth University
Richmond, USA
brookss9@vcu.edu

***Abstract*—Hash Functions and Brute Force Search algorithms have been around just as long as computer science itself. Their core ideas are the fundamentals of security that we have today in the growing fields of cybersecurity and cryptography. The growing complexity of cracking hash functions and sheer computational time needed to Brute Force password combinations is a clear use case of GPU computing in today's landscape. With the GPU's sheer computational growth year over year we have seen the need for more entropy in passwords than ever before. In this paper I will cover the prospects of shifting Brute Force algorithms to the GPU and what that can entail in terms of run times, speedups, and scalability. Implementing the MD2 hash function along with a Brute Force Search algorithm in Cuda to go over the pros and cons of this approach.**

***Index Terms*—Brute Force, Hash Function, MD2, GPU**

I. Introduction

Over the years as we have seen the explosion in internet usage, we have seen the rise and fall of many malicious attacks on users and networks. From the evolution of Denial of Service attacks to more severe types with the rise of trojans with auto encryption built in. The one attack that has stood the test of time is the Brute Force attack when trying to crack passwords for any given site. The goal of password security in the beginning was to outpace the computational capabilities of computers at the time as we have seen the recommended password length grow as years go by. This idea of password entropy also known as the amount of attempts needed to guess the password, is the core idea of its security against brute force attacks. E = log2 (number of combinations) being the formula for this calculation. True random passwords in this sense are the best for any password requirements as they are going to be the highest net entropy as with using actual words and combinations of words you can fall victim to dictionary attacks that reduce the net entropy of a brute force problem.

```
c ← first(P)
while c ≠ Λ do
    if valid(P,c) then
        output(P, c)
    c ← next(P, c)
end while
```

Fig. 1. Pseudocode for a general brute force search (Wikipedia).

The runtime of this problem on the CPU is the reason that the sheer entropy matters as it is run sequentially with one password combination being output at once. So as we push into the minimum password length in today's world of normally 10-12 for most websites we see a combination total of 3.7e+18 to 1.94e+22. The exponential growth aspect as you add just a single character creates the massive runtimes you will see below. This leads us into the rise of GPU computing and the massive benefits it can have when simplistic calculations are being done. The GPU allows for the parallelization of problems that when run sequentially equate to massive runtimes. We have seen its usage in many fields including cybersecurity, cryptography, and machine learning. It being the main reason we are able to even run the LLM's we see today. In the case of this paper and its main idea GPU's and their sheer computations per tick are the reason your passwords have had to grow in length in recent years. Gone are the days of 7 digit passwords that you came up with in middle school. The GPU has trivialized the brute forcing of any password under 8 characters that consist of the lower case alphabet, upper case alphabet, 0-9, and the allowed special cases. The mapping of a single password combination to a thread in the GPU in theory would allow for the instant computation of millions of computations per tick. Leading to the drastic decreases in overall computation time we see today in brute force attacks. Where this computation time is increased is when we start to find the hash value of one of these computed combinations. In this implementation you will see that even without the theoretical 1:1 mapping and only the parallelization of the major for loop used in the recursion we see massive decreases in overall run time.

```
{ 0x29, 0x2E, 0x43, 0xC9, 0xA2, 0xD8, 0x7C, 0x01, 0x3D, 0x36, 0x54, 0xA1, 0xEC, 0xF0, 0x06, 0x13,
  0x62, 0xA7, 0x05, 0xF3, 0xC0, 0xC7, 0x73, 0x8C, 0x98, 0x93, 0x2B, 0xD9, 0xBC, 0x4C, 0x82, 0xCA,
  0x1E, 0x9B, 0x57, 0x3C, 0xFD, 0xD4, 0xE0, 0x16, 0x67, 0x42, 0x6F, 0x18, 0x8A, 0x17, 0xE5, 0x12,
  0xBE, 0x4E, 0xC4, 0xD6, 0xDA, 0x9E, 0xDE, 0x49, 0xA0, 0xFB, 0xF5, 0x8E, 0xBB, 0x2F, 0xEE, 0x7A,
  0xA9, 0x68, 0x79, 0x91, 0x15, 0xB2, 0x07, 0x3F, 0x94, 0xC2, 0x10, 0x89, 0x0B, 0x22, 0x5F, 0x21,
  0x80, 0x7F, 0x5D, 0x9A, 0x5A, 0x90, 0x32, 0x27, 0x35, 0x3E, 0xCC, 0xE7, 0xBF, 0xF7, 0x97, 0x03,
  0xFF, 0x19, 0x30, 0xB3, 0x48, 0xA5, 0xB5, 0xD1, 0xD7, 0x5E, 0x92, 0x2A, 0xAC, 0x56, 0xAA, 0xC6,
  0x4F, 0xB8, 0x38, 0xD2, 0x96, 0xA4, 0x7D, 0xB6, 0x76, 0xFC, 0x6B, 0xE2, 0x9C, 0x74, 0x04, 0xF1,
  0x45, 0x9D, 0x70, 0x59, 0x64, 0x71, 0x87, 0x20, 0x86, 0x5B, 0xCF, 0x65, 0xE6, 0x2D, 0xA8, 0x02,
  0x1B, 0x60, 0x25, 0xAD, 0xAE, 0xB0, 0xB9, 0xF6, 0x1C, 0x46, 0x61, 0x69, 0x34, 0x40, 0x7E, 0x0F,
  0x55, 0x47, 0xA3, 0x23, 0xDD, 0x51, 0xAF, 0x3A, 0xC3, 0x5C, 0xF9, 0xCE, 0xBA, 0xC5, 0xEA, 0x26,
  0x2C, 0x53, 0x0D, 0x6E, 0x85, 0x28, 0x84, 0x09, 0xD3, 0xDF, 0xCD, 0xF4, 0x41, 0x81, 0x4D, 0x52,
  0x0A, 0xDC, 0x37, 0xC8, 0x6C, 0xC1, 0xAB, 0xFA, 0x24, 0xE1, 0x7B, 0x08, 0x0C, 0xBD, 0xB1, 0x4A,
  0x78, 0x88, 0x95, 0x8B, 0xE3, 0x63, 0xE8, 0x6D, 0xE9, 0xCB, 0xD5, 0xFE, 0x3B, 0x00, 0x1D, 0x39,
  0xF2, 0xEF, 0xB7, 0x0E, 0x66, 0x58, 0xD0, 0xE4, 0xA6, 0x77, 0x72, 0xF8, 0xEB, 0x75, 0x4B, 0x0A,
  0x31, 0x44, 0x50, 0xB4, 0x8F, 0xED, 0x1F, 0x1A, 0xDB, 0x99, 0x8D, 0x33, 0x9F, 0x11, 0x83, 0x14 }
```

Fig. 2. The S-table that is used in the MD2 hash function (Wikipedia).

This would be the end of the conversation in terms of finding a password but that is not the case as every password from any credible website or application is going to be run through a hash function before being stored. As storing a password

Identify applicable funding agency here. If none, delete this.

in plaintext in a database is a core fundamental mistake that no company should make in today's world. The goal of most hashes fall into two categories. One being that it is innately fast and that it is completely collision free. Meaning that no two strings map to the same output hash in a set length of input. The hash values themselves or often times called digests are the actual output of the hash and no matter the input length are always the same size. For example the MD2 is collision less but the later implementations are not like the MD5. So in today's environment most brute force attacks are used to search for a specific hash that has been posted from a given breach. Meaning you are exhaustive searching in a given password length range until your hash output matches the given hash from the breach. In the practical sense an attacker will usually run this hash against a rainbow table that in itself is a list of compromised passwords and their resulting hash from a given hash function. In the case of this paper I will be implementing and using the MD2 hash function as a proof of concept for the overarching idea. The MD2 was created in 1989 by Ronald Rivest for use in securing emails at the time, with the MD4 and MD5 coming out after. The MD2 was finally decommissioned in 2011 as at that point it had become inefficient with multiple successful crypto attacks against it noted. I will go over the fundamental idea of the MD2 here to promote a better understanding of what is happening in the code in later sections. The general idea of the MD2 is that it is going produce a 128-bit hash value through various bitwise calculations done to the input data while then padding it to a length of 16 bytes as laid out in the paper written by B. Kaliski in 1992. In this case it is a XOR calculation that is done on the given S-table to produce values for our block variable. Finally it then appends a 16-byte checksum that has been actively calculated while running to the end of the data. The actual math comes out to be a 48 byte block and a 256-byte S-table being used. The transformation math involved is to permute each byte in the block 18 times for every 16 input bytes that are read in. Finally the actual hash output comes from the first partial block of the block variable of size 48 we created to start. The code implementation of this will be explained in the methodology section below.

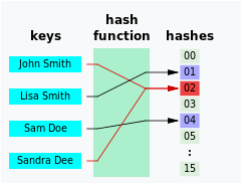

Fig. 3. Showing what it means for hash digests to be collision free (Wikipedia).

II. Methodology

In this section I will go over the specific implementations of the CPU and GPU code while covering any issues related to their scalabilities as well. A final note before the shifting into the explanations of the implementations is to say that all of the parallelization in this program is on the brute force search itself and not the hash function. Trying to parallelize the mass amount of password combinations that grow at an exponential rate will offer even at the non-efficient side far better speedup returns then trying to squeeze out any savings in the three has method calls. As you move to other hash functions as well the more secure they are usually means they are slower as their output bit length is longer. This may not seem substantial in terms of a single run of input but when we are running that e+21 times because our input length is 12 we will see this change in run time. This all to say that this is an inadvertent security added to having a more secure hash function when thinking of run time cases for brute forcing passwords.

A. CPU Implementation

To being on the CPU the program will begin by reading in a command line argument that is going to be the password length we will be brute forcing all combinations of. In the case of scalability here we can use a sh script to run all previous lengths if needed. Once the length is read in we will call our bruteForce method that will take in arguments for an output char that will store our current password combinations, the starting index of 0, and the input password length integer.

From this we move to the brute force method where we will enter the main for loop of the program that will iterate the length of the set alphabet size. In this case it is 62 consisting of the lower case alphabet, uppercase alphabet, and the digits 0-9. This is one part of scalability that can be easily shifted as the size is a definition along with the input characters if you were to want to remove characters or add in special characters. In the for loop out output will be set to the input character at an index value that equals our current iteration in the for loop.

Then we see the major if condition that handles the recursion of the loops. You will only enter the for loop if your index value is equal to the length of the password. This is the case because we are going to recursively iterate until at the last index where we will enter the if statement and only here will be start the process of calling the MD2 methods. To start we will the output as our current password and fill in the last index with a null value to eliminate any weird read in's or outputs.

Once done it is time to initialize our MD2 struct that is going to hold all of the values needed for its calculations and output. The first call of the MD2 is to the initialize method which is going to take in our current context as the argument and zero out all of the needed variables we created.

Next is the update call that is going to take in the context again, our current password, and the length of that password which is just our read in value. This method is going to iterate through the password length and store each current password char into a corresponding char data that our context holds at an index value length that always starts at 0. In this method we have one if condition that you enter if the current length is equal to 16 as the max size is 16 bytes for the char data. Once inside we will call our transform with our data as the argument to run all of the bitwise math on top of the input.

	Runtime						
Password Length	**1**	**2**	**3**	**4**	**5**	**6**	**7**
CPU	0.7ms	42ms	2.5s	157s	161.05m	166.4h	429.8d
GPU	7ms	17ms	0.684s	42s	43.7m	45.2h	116.75d

Fig. 4. The runtimes of the CPU and GPU code.

```
/* Process each 16-word block. */
For i = 0 to N'/16-1 do

   /* Copy block i into X. */
   For j = 0 to 15 do
      Set X[16+j] to M[i*16+j].
      Set X[32+j] to (X[16+j] xor X[j]).
    end /* of loop on j */

   Set t to 0.

   /* Do 18 rounds. */
   For j = 0 to 17 do

      /* Round j. */
      For k = 0 to 47 do
         Set t and X[k] to (X[k] xor S[t]).
      end /* of loop on k */

      Set t to (t+j) modulo 256.
   end /* of loop on j */

end /* of loop on i */
```

Fig. 5. The transform method in Pseudocode as seen in the original MD2 paper (9).

When done we will zero out our length and continue reading in data if required.

To note the transform method is going to read in a an input from its method call that can be either our checksum or buffer that holds our initial password input. Once inside it is going to iterate 16 times and fill in our block variable with the corresponding input while also filling in another index inside of the block variable with a XOR calculation of two block values. When out of this loop this is where we will do the 18 iterations of the full size 48 auxiliary block (our block data). The main math is to set the value of the block in the iteration of 48 to the XOR calculation found from comparing to the S-table. The value we are looking at in the S-table will then be updated as we iterate the outer for loop of 18 so that our block value is compared to a different value each time.

Finally our last method call is to final which is the corresponding final call of the MD2 that is going to take in a buffer that will be used to store our output hash in. This method is used to calculate the needed padding for our input which will at least be 1 but is based off of the calculated length of our password. When done we go to a for loop that is going to iterate over this length while less than 16 and append the padding to our data. Finally we will do two final transformations, one being on our data, then on our checksum to append it to the end of the input. This results in our hash output which we scrape from the current block which as stated above.

This results in a hash that needs to be converted to be readable but this then finishes the for loop. This will iterate the password length – 1 index as we until we get every combination output. The only time an index will iterate its for loop is if the index after it has finished all its alphabet size iterations. The final option part of this code would be implementing a read in for a hash that you would compare to the current hash of your code so that the code will stop running at spit out the current password combination. This didn't seem like the correct implementation as it would have cut the program short and I was focused on getting the total run times instead of what this would be used for in practicality of getting a specific password from a matching hash.

B. GPU Implementation

The GPU implementation starts off the same as the CPU code with a read in of the password length to brute force. This is where the shift happens as we then put this into a our BFS method call that is going to read in the password length only. Once inside this is where we are going to declare and initialize all of the variables needed to run this on the kernel. When done we will call our kernel that is going to be the main driver of the brute force algorithm.

The parallelization of the program is done by mapping a single thread to a instance in the main for loop. This loop being the one that iterates the size of the alphabet size. Resulting in our case 62 threads that each handle every password combination starting with that given letter. The implementation in this case is nested for loops that match the password length we are given by user input. Currently this is hard coded as the initial implementation of recursion is not suited for the GPU.

This is because when recursion happens on the GPU the beginning memory usage is known but you will run out of memory on the thread after a set amount of iteration in the loop. So to work around that the nested for loop was used to get outputs to see run time comparisons. The MD2 implementation is the exact same on the GPU as it is on the CPU with the same method calls and initializations. The singular difference between the two is that there is no sprint call on the GPU so to get the hash output there needed to be an atrocious printf statement. To note before moving on in the code there is a commented out implementation of the recursive algorithm on the GPU as a proof of concept of how it handles memory per thread.

III. Results

In this section I will detail the runtimes and speedups of the above mentioned implementations. Shown with the graph above we can see the clear advantages of the GPU implementation when running all lengths of input. The estimated run time can then be calculated with a reasonable degree from

Number of Characters	Numbers Only	Lowercase Letters	Upper and Lowercase Letters	Numbers, Upper and Lowercase Letters	Numbers, Upper and Lowercase Letters, Symbols
4	Instantly	Instantly	Instantly	Instantly	Instantly
5	Instantly	Instantly	Instantly	Instantly	Instantly
6	Instantly	Instantly	Instantly	Instantly	1 secs
7	Instantly	Instantly	6 secs	21 secs	50 secs
8	Instantly	1 secs	5 mins	22 mins	59 mins
9	Instantly	33 secs	5 hours	23 hours	3 days
10	Instantly	14 mins	1 weeks	2 months	7 months
11	1 secs	6 hours	1 years	10 years	38 years
12	6 secs	7 days	76 years	623 years	2k years
13	1 mins	6 months	3k years	38k years	187k years
14	10 mins	12 years	204k years	2m years	13m years
15	2 hours	324 years	10m years	148m years	917m years
16	17 hours	8k years	552m years	9bn years	64bn years
17	1 weeks	219k years	28bn years	571bn years	4tn years
18	2 months	5m years	1tn years	35tn years	314tn years

Number of Characters	Numbers Only	Lowercase Letters	Upper and Lowercase Letters	Numbers, Upper and Lowercase Letters	Numbers, Upper and Lowercase Letters, Symbols
4	Instantly	Instantly	Instantly	Instantly	Instantly
5	Instantly	Instantly	Instantly	Instantly	Instantly
6	Instantly	Instantly	Instantly	Instantly	Instantly
7	Instantly	Instantly	2 secs	7 secs	16 secs
8	Instantly	Instantly	2 mins	7 mins	19 mins
9	Instantly	10 secs	1 hours	7 hours	22 hours
10	Instantly	5 mins	3 days	3 weeks	2 months
11	Instantly	2 hours	6 months	3 years	12 years
12	2 secs	2 days	24 years	198 years	848 years
13	19 secs	2 months	1k years	12k years	59k years
14	3 mins	4 years	64k years	759k years	4m years
15	32 mins	103 years	3m years	47m years	290m years
16	5 hours	2k years	175m years	2bn years	20bn years
17	2 days	69k years	9bn years	181bn years	1tn years
18	3 weeks	1m years	473bn years	11tn years	99tn years

Fig. 6. The left are run times of hashcat on a 4900 and on the right are on a cluster of 8 A100's hosted on AWS (Hive Systems).

```
/* Clear checksum. */
For i = 0 to 15 do:
   Set C[i] to 0.
end /* of loop on i */

Set L to 0.

/* Process each 16-word block. */
For i = 0 to N/16-1 do

   /* Checksum block i. */
   For j = 0 to 15 do
      Set c to M[i*16+j].
      Set C[j] to S[c xor L].
      Set L to C[j].
    end /* of loop on j */
 end /* of loop on i */

The 16-byte checksum C[0 ... 15] is appended to the message. Let M[0
with checksum), where N' = N + 16.
```

Fig. 7. The check sum updating that will be inside of the transform method in Pseudocode from the original paper (9).

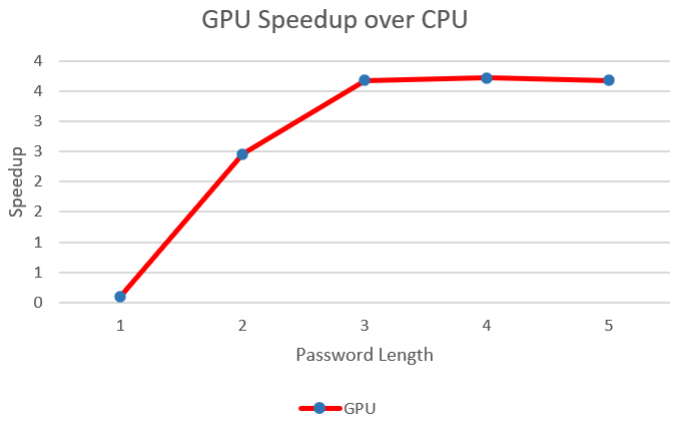

Fig. 8. The speedup of the GPU compared to the CPU runtime.

here by multiplying the previous length time by the alphabet size.

There are clear scalability issues when looking at the CPU implementation as with it running sequentially it has clear problems with the exponential increase in password combinations. Anything over a password length of 4 is just going to be far too long to run realistically. Although it does not have the same run time issue as the GPU when we look at how the added hash function slows down the run times. As noted when running just the GPU without calling the MD2 we see our run times get cut in half.

There are also issues in the implementation of the GPU code as it does not map one combination to a thread therefore not taking anywhere near full advantage of the computing power. Using a single block and 62 threads is not even a stone's throw to what could possibly be done on a GPU. Research has been run on a cluster of GPU's running Hashcat as noted in the article written by Hive Systems to further test the compatibility of this problem with the GPU. They used a buyable cluster of A100's from amazon for their best results. Although based on the run times they see about a 3x speedup from a 4090 to a cluster of eight A100's hosted on AWS. Whereas the speedup of ten A100's compared to the initial cluster of eight is only 1.5x. Showing that there is some diminished returns in this approach at the very top end of computing power. The theoretical maximum that a brute force algorithm could reach is mapping one combination to a thread and then expanding this with MPI to multiple GPU's. With this you could scale your way into technical quantum computing.

IV. Conclusion

In conclusion we started with a CPU implementation of a brute force search algorithm that recursively calls as it finds a given length password combination. Once found this then gets initialized and ran through the MD2 hash function to get our current password hash. This approach then had to change as we shifted to the GPU with it not handling recursion well at all memory wise. As there is an allocation done per thread at the kernel call and not dynamically as the thread is running. At some point in the total of ten million iterations the recursive gpu code will run out of memory and end running. There are two noted ways to handle this with one being a stack and iteration while the other being an array to keep the state as seen with how I implemented this. Not being able to run a password length larger than 3 meant a change needed to

happen so a nested for loop tree was created to run any given length attempt. The main issues came up when trying to get the 1:1 mapping on the GPU. I'll use an example of a password length of 4 to show this. We are able to locate the index 0 value by (TID / (total combinations / alphabet size)) and able to find the last index with (TID % alphabet size). This left us with the two inner indexes to find, which proved to be very difficult and eluding. Although hardcoded for this proof of concept it shows the sheer speedup capabilities of the brute force search on the GPU when only parallelizing the loop of alphabet size. This leads into the train of thought that to take full advantage of the GPU computation you would need to map a single password combination to a thread and would thus be able to spit out millions of combinations in a single tick. The scalability can be further expanded my implanting the above topics with MPI once the mapping of threads to combinations has been done. Some future work that could be done looking back after working on this would be exploring the breaking down of the recursive CPU code into the stack variation as noted in research. The viability of this on the GPU comes with a lot of questions as you want to parallelize entire problems in single ticks with millions of threads that contain no knowledge of each other. There may lie issues in the inner index mapping as the combination "abaa" would hold the same value as "aaab" if trying to mathematically distinguish each TID to a specific combination like in a normal two for loop problem. It's the expansion into more than 3 loops that causes the issue in theory.

REFERENCES

[1] A, Georges. "Everything You Need to Know about the MD2 Hash Function." NetChunk Blog, 4 Nov. 2022, Accessed 3 Dec. 2023.

[2] Wikipedia Contributors. "Brute-Force Search." Wikipedia, Wikimedia Foundation, 1 Apr. 2019.

[3] Stec, Albert. "Deep Dive into Hashing — Baeldung on Computer Science." Www.baeldung.com, 9 Aug. 2020.

[4] "Password Entropy and How to Calculate It — NordVPN." Nordvpn.com, 18 Aug. 2023.

[5] "MD2 Hash Function." Wikipedia, 7 July 2020.

[6] by, Written. "MD5 vs SHA256: Which Is Better? Speed, Safety, …." InfosecScout.

[7] Wikipedia Contributors. "Hash Function." Wikipedia, Wikimedia Foundation, 8 Sept. 2019.

[8] Linn, John "Privacy Enhancement for Internet Electronic Mail:Part III – Algorithms, Modes, and Identifiers", August 1989

[9] Kaliski, Burton "The MD2 Message-Digest Algorithm". April 1992