GPU Accelerated Dictionary Attack Against SHA-1 Hashed Passwords

Simone Cosimo
simone.cosimo@studenti.polito.it

13 February 2023

Abstract

This document presents a possible strategy to perform fast and effective dictionary attacks against SHA-1 hashed passwords. The software, developed using the CUDA framework and running on GPGPUs, performs the hashing of every word present in a dictionary of commonly used passwords and compares it to a target hash. The result is then compared to standard CPU implementation to check the correctness and performance.

1 Introduction

In the cybersecurity field, a dictionary attack is a technique used by an attacker to retrieve the clear text password from a specific target hash.

A cryptographic hash function is a mathematical function that maps data to specific values in a deterministic way: if data passed as input is not changed, the output will not change either. But, if just one bit of the data as input is flipped, the output will drastically change. Furthermore, these functions are difficult to invert: by only knowing the output it is very difficult to discover the input that originally generated it. These features of a hash function are extremely useful when it comes to storing passwords, to protect them from breaches of the database in which they are memorized.

In fact, information systems usually store the digests of passwords for privacy and security reasons: in this way, an attacker that can get access to the content of the database will be faced with the difficult challenge of discovering the original clear-text password that generated the digest in its possession. Dictionary attacks are one of the possible solutions to this problem.

This project aims to explore a solution to parallelize dictionary attacks and improve basic solutions of the attack based on standard CPU implementations using the CUDA framework and NVIDIA GPUs.

2 Related works

2.1 SHA-1 algorithm

SHA-1, also known as Secure Hash Algorithm 1, is a hash function that transforms data producing a 20-byte result hash value known as a digest. The digest is typically represented as 40 hexadecimal digits for simplicity of display. This specific algorithm is been cryptographically broken but it remains widely used, even though its successors have been designed and can be used (they are more performance heavy).

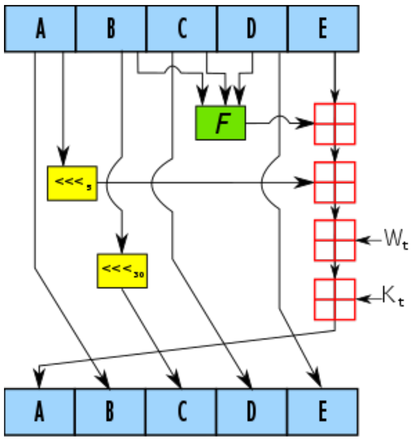


Figure 1: SHA-1 algorithm diagram - from Wikipedia

The algorithm works by means of blocks: data in input is dived into blocks of a fixed size of 512 bits and then each of these blocks goes through a specific transformation function. In the end, if the size of the data is not exactly a multiple of the block size, padding is added.

Figure 1 shows the basic diagram of the algorithm, let's analyze the main components:

- A, B, C, D, E: these are variables that represent the state of the algorithm. They are initialized to some fixed hexadecimal values and after the transformation of each block, they are updated.
- F function: a non-linear function that varies during computation of blocks

2.2 Dictionary attack

A dictionary attack is a possible strategy for the attacker to succeed in its scope to discover the clear-text password that generated the target digest. This technique is based on working with a dictionary containing commonly used clear-text passwords. This dictionary is then passed as input to software that calculates the digest for each password in the dictionary and constantly checks if the result is equal to the stolen target digest.
The success of this kind of attack is based on two main points:

1. The number and type of passwords: if the dictionary contains a lot of passwords the probability of success increases. Also, a dictionary of common words could not be enough. It's better if the content represents commonly used passwords by real users.
2. Software performance: dictionaries are usually very big to increase chances to find the target password. This leads to a very time-consuming computation if the software is not optimized. The risk is never being able to analyze the last words in the dictionary due to unpractical computation time and performance limits.

Most recent and performing CPU implementations of the attack are achieved using multithreading or extended instruction set, which helps the performance in a substantial way. One of the most known and efficient implementations is the hashcat suite which has both CPU and GPU software that not only limits to dictionary attacks but also implements brute force attacks and mask attacks.

3 Proposed methods

In this section are listed and explained the design and implementation choices that were made in order to improve the performance of a standard CPU implementation of a dictionary attack.
To implement a GPU version of the attack, the CUDA framework was used to fully use the parallelization features provided by the architecture of an NVIDIA GPGPU.

First thing first, the software needs to be launched with some parameters. In particular:

- the path to the dictionary file containing the clear-text passwords
- the number of passwords in the database (could be avoided, but there are some considerations about memory, more on this later)
- the target hash specified as a hexadecimal string

3.1 Memory organization

The first step is to find a way to read the file and save the passwords into a structure that will then be copied into device memory to execute the hash and the comparison with the target hash. A structure is needed because, together with the password itself, the length of the word is required for the hash computation, and getting it from the kernel itself wouldn't be too comfortable since no standard library methods are available from within the device.
The struct I ended up implementing is as simple as the following:

```
typedef struct {
        unsigned char word[50];
        unsigned int len;
} PWD_INFO;
```

It can be seen that the field containing the password is statically allocated as an array of a maximum of 50 characters. This is an assumption, and I think it is a fair one: a password is a user-friendly, mnemonic string and it is unlikely to be composed of 50 or more characters (in the 5 million most used passwords, every single one of them is shorter than 50 chars).

Once the struct is designed, the dictionary needs to be read and the information is then transferred to the device memory. During the development I tried different approaches:

- The first approach was to read the document in batches. For example, the first 256 lines were read, saved into the struct, copied to the device's global memory, and so on until the end of the file. This created some problems, one of them was the number of calls to the cudaMemcpy function, which created a lot of overhead and waste of time. Furthermore, the kernel call was executed multiple times on different data and profiler analysis showed more overhead and so, worst performance.

- The second and more efficient approach is to read the whole file at one time. This solves all the problems related to the first way of doing things: the call to cudaMemcpy method is one, and so is the kernel call, which is now capable of doing computation on all the data.

To enable the second strategy we need to know the number of passwords in the file before reading the dictionary in order to allocate the necessary memory in advance. This is the reason to ask for the number of passwords as one of the software parameters.

```
unsigned int alloc_size = (db_size > 75000000 ?
                           75000000 :
                           db_size);

while(std::getline(passwords, line)) {
    local_indata[size].len = strlen(line.c_str());
    strcpy((char *)local_indata[size++].word, line.c_str());
}
```

```
gpuErrchk( cudaMemcpy(cuda_indata,
                local_indata,
                alloc_size * sizeof(PWD_INFO),
                cudaMemcpyHostToDevice));
gpuErrchk( cudaMemcpy(d_target,
                target, 40 * sizeof(unsigned char),
                cudaMemcpyHostToDevice));

mcm_cuda_sha1_hash_batch_BUFFER(cuda_indata,
                            size,
                            d_target,
                            d_match);
```

This block code represents what was explained before, terminating with the call to the wrapper function that will then call the main CUDA kernel. The 75000000 value refers to the maximum number of PWD_INFO structs that can be saved into the device's global memory, which in this case is the NVIDIA Jetson Nano Developer Kit.

3.2 Blocks and threads organization

The main kernel is launched over a number of blocks that depends on the number of passwords present in the dictionary with the constraint that every block will spawn 256 working threads each. Each thread will calculate the hash of a single password and then transform the output into a hexadecimal string and compare it to the target hash.

```
#define THREADS 256
unsigned int thread = THREADS;
unsigned int block = (buffer_len + thread - 1) / thread;
kernel_sha1_hash_BUFFER<<< block, thread >>>(cuda_indata,
                                    buffer_len,
                                    block - 1,
                                    target, match);
```

3.3 SHA-1 device algorithm implementation

The implementation of the SHA-1 algorithm is GPU code ported from the CPU code originally made by Brad Conte. It follows the workflow of initialization, update, and finalization of a context, a data structure used to memorize the state and variables needed for the algorithm to compute the final correct digest.

```
typedef struct {
        unsigned char data[64];
        unsigned int datalen;
        unsigned long long bitlen;
        unsigned int state[5];
        unsigned int k[4];
} CUDA_SHA1_CTX;
```

Other kinds of optimizations were made using registers and local variables to help performance during computation.

3.4 Main kernel and device functions

The main kernel is really simple and its focus is to instruct every thread to calculate one password SHA-1 digest and then transform the result in hexadecimal form in order to be compared to the target. To achieve this the following actions are taken:

- Thread index is calculated to be able to work on specific data
- The hash is calculated by initializing the context, updating it by specifying the password in input and its length, and finalizing it by saving the digest in a local variable
- The final digest is then processed and transformed to be able to compare it with the target

If the comparison fails, the thread just returns and stops computation. If the comparison succeeds, the only thread that ”survived” the comparison loop will update a variable that indicates the index of the right password inside the dictionary.

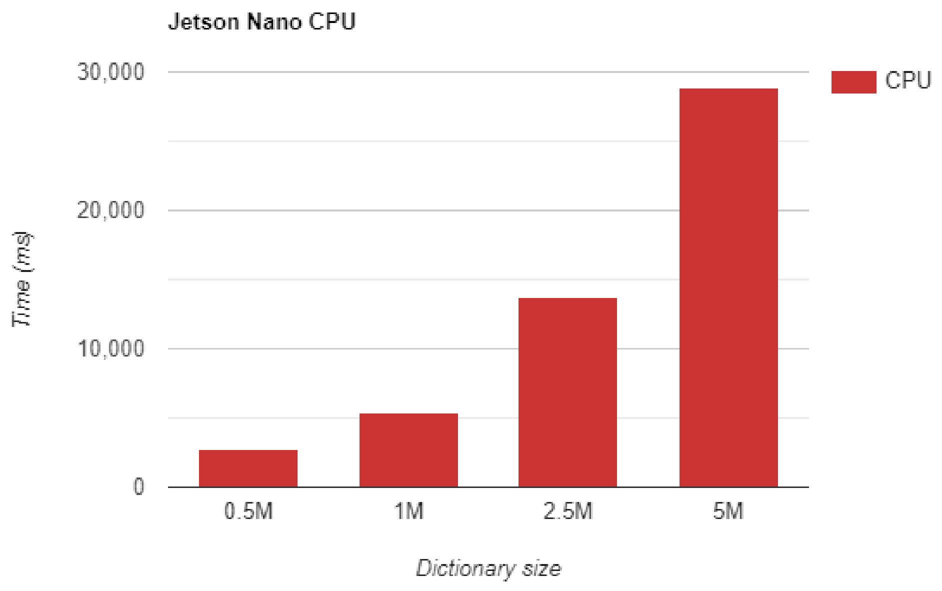
4 Experimental results

The software was executed on two systems, which specifications are reported in the table below.
The CPU software used to compare the performance improvement of the GPU implementation is a simple program that reads the whole file and memorizes information in the same PWD_INFO struct explained before and then starts a loop where each password is hashed, transformed and compared to the target.

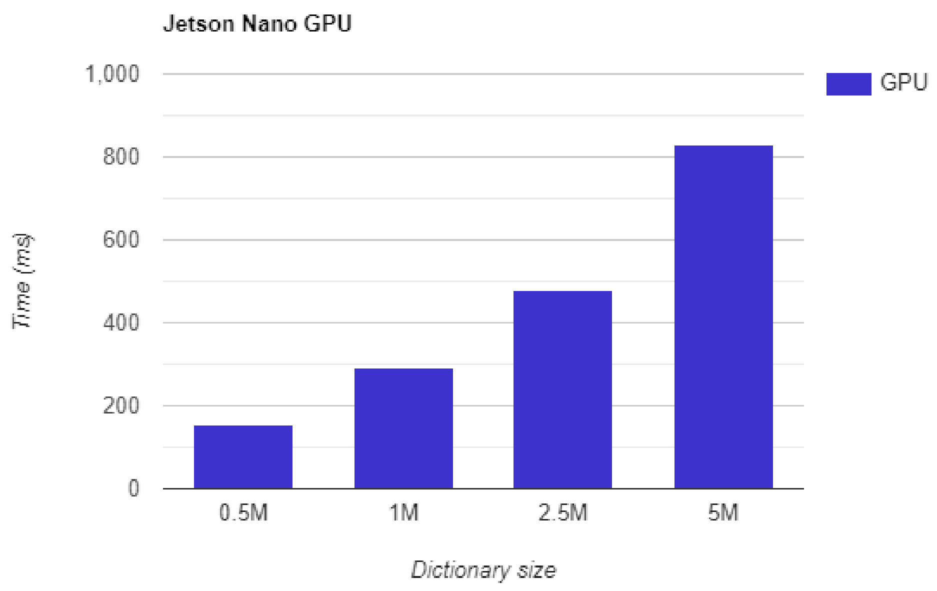
This CPU implementation is not parallelized, so the performance improvements need consideration: the increase in performance written below is higher than a real-case scenario because the CPU implementation is not using multithreading to perform computation, which would shorten computation time a lot. Since the scope of the project was to develop a GPU solution, I mainly focused on it.

Name	CPU	GPU
Jetson Nano	Quad-core ARM A57	128-core NVIDIA Maxwell
PC	AMD Ryzen 5 2600X 6 core	NVIDIA RTX 2070

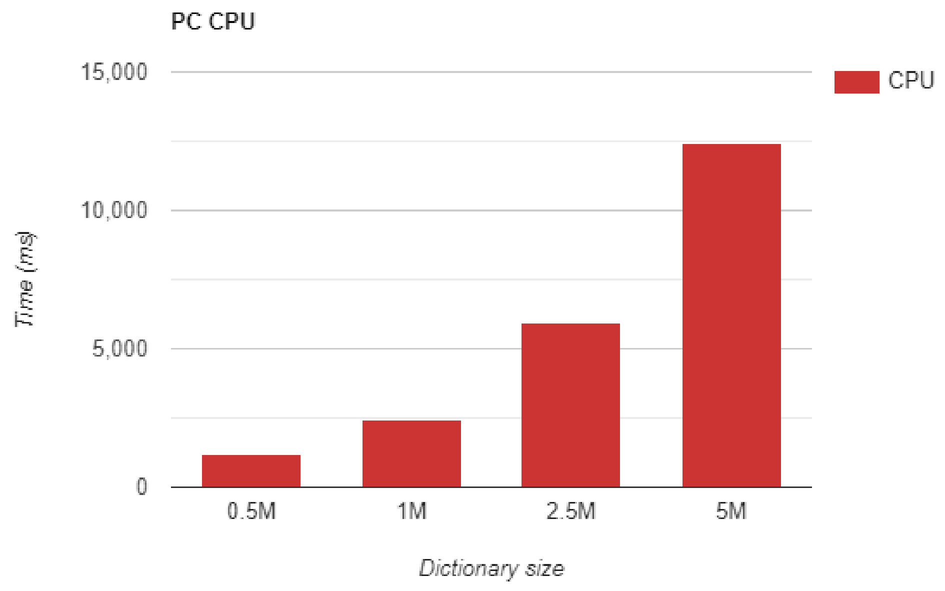
Tests are executed over dictionaries of different sizes, respectively 500 thousand, 1 million, 2.5 million, and 5 million passwords. The dictionaries are taken from the SecList repository and contain the most used passwords. Timings are considered as if the target hash to compute corresponds to the last password in the database, being this the worst-case scenario.

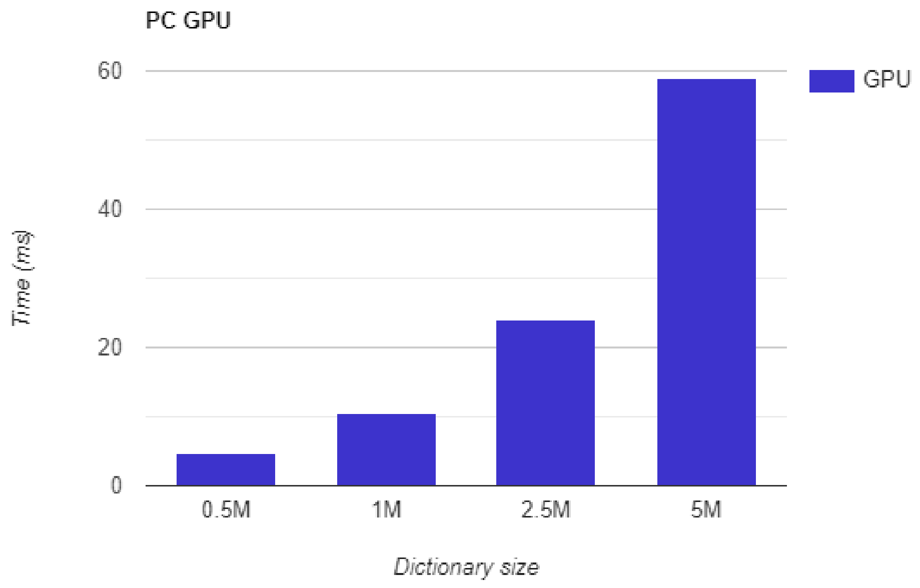


Jetson Nano CPU
CPU
30,000
20,000
10,000
0
Time (ms)
0.5M
1M
2.5M
5M
Dictionary size



Jetson Nano GPU
GPU
1,000
800
600
400
200
0
Time (ms)
0.5M
1M
2.5M
5M
Dictionary size





As we can see, the improvement is substantial: on the Jetson Nano system, considering the 5 million words dictionary, we hash and compare every password in circa 30 seconds with standard CPU software, while we do the same in a little more than 800 milliseconds using the GPU implementation.

In the PC case, results are similar, going from 12 seconds circa, to 60 milliseconds circa.
An important note needs to be done regarding the timings of GPU software: in these timings, memory operations such as data copies to and from the device are included.

5 Conclusions

With this project, I analyzed the performance improvements to standard CPU software for dictionary attacks using the CUDA frameworks and the GPU capabilities. The executed tests were performed on dictionaries with a large number of passwords but a real-life scenario (ones where standard CPU computation would take a lot of hours or days) would be the most interesting to analyze.

6 Bibliography

1. Hashcat Suite - `https://hashcat.net/hashcat/`
2. Brad Conte crypto algorithms - `https://github.com/B-Con/crypto-algorithms`
3. SHA-1 - `https://en.wikipedia.org/wiki/SHA-1`
4. SecList GitHub repository - `https://github.com/danielmiessler/SecLists`
5. NVIDIA Jetson Nano - `https://developer.nvidia.com/embedded/jetson-nano`
6. Executing Parallelized Dictionary Attacks on CPUs and GPUs - `https://moais.imag.fr/membres/jean-louis.roch/perso_html/transfert/2009-06-19-IntensiveProjects-M1-SCCI-Reports/HassanShayma.pdf`
7. Password Recovery Using MPI and CUDA - `http://tdesell.cs.und.edu/papers/2012_hipc.pdf`