GPU Accelerated Dictionary Attack Against SHA-1 Hashed Passwords

Simone Cosimo – s303333

GPU Programming course

February 2023

Topics

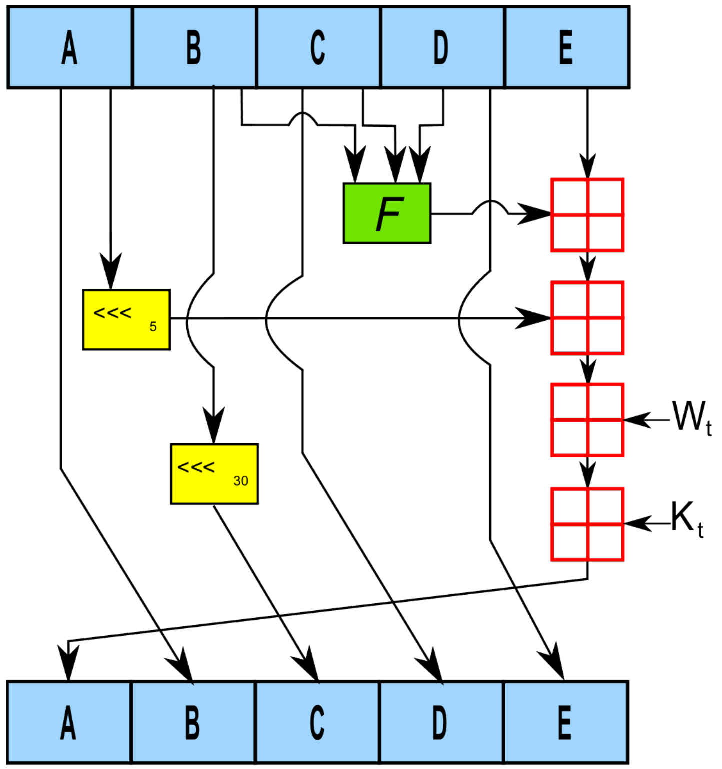
- Hash functions and SHA-1 algorithm
- Dictionary attack
- GPU implementation
  - Memory organization
  - Blocks and threads
  - Main Kernel
- Results vs CPU implementation
- Conclusions and Improvements

Hash functions

A cryptographic hash function is a mathematical function that maps data to specific values.

- Same input data always produce the same output, called digest
- **It works processing data in blocks** (perfect for parallelization)
- Depending on the algorithm, it produces outputs with different lengths
- **Should be complex to invert**
- Used by information systems to store passwords

SHA-1 Algorithm



- Output is always **20 bytes** long
- Block size is **512 bit**
- It initializes a state with fixed values
- The state is used to transform the block and it is updated
- A non-linear function is used to combine the state and the data
- **Padding is added if input data is not multiple of block size**

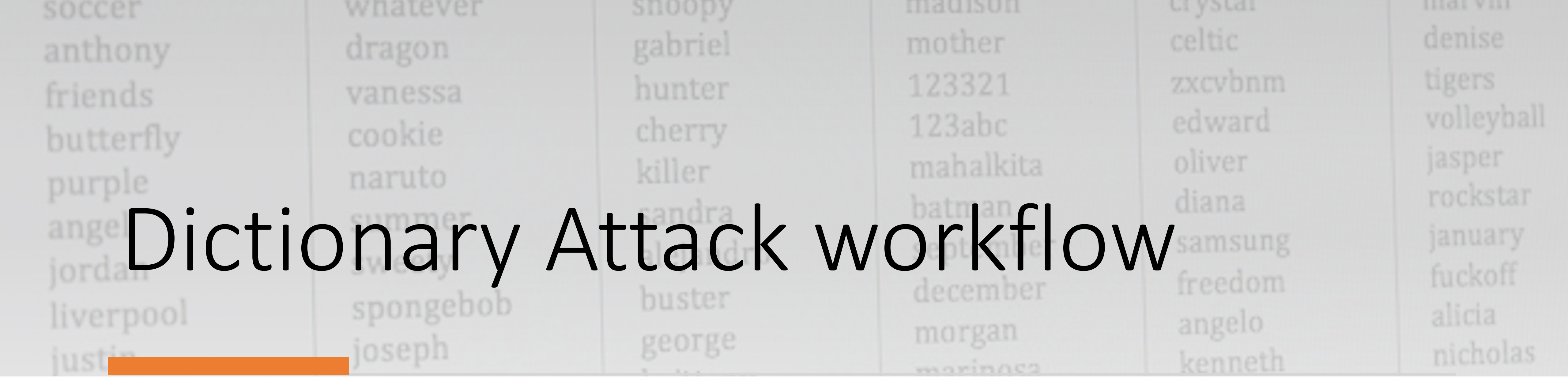
Dictionary Attack

A dictionary attack is a strategy used by an attacker to retrieve the original clear-text password starting from a stolen digest.

This attack is based on a dictionary, a set of words or commonly used passwords.

Every word present in this password is given to a hash functions and the result compared to the target digest.

If a word is hashed into the same sequence of byte of the target digest the password has been found, otherwise the password wasn’t in the considered database.



Dictionary Attack workflow

- Attacker gains possession of passwords digest and the algorithm that was used to generate them
- Attacker uses a dictionary of commonly used passwords
- Each word in the database is hashed and compared to the target digest
- If a match is found, the password is known
- Otherwise, password is not in the database or protection techniques were used during digest computation (e.g. **salt**)

GPU Implementation

Dictionary attacks are a perfect example of how the GPGPU capabilities to parallelize execution can improve performances.

Common CPU implementations can only spawn few threads to hash words in the dictionary, while GPUs are more resourceful under this point of view, guaranteeing better performance.

This is important because **a successful dictionary attack is often executed with very big dictionaries** (millions of passwords) that lead to high execution time.

Memory Organization

- Dictionary file is passed as input and each password is store in the **PWD_INFO** struct
- The struct size is of **54 bytes**, assuming no password is longer than 50 chars
- The length of each password is memorized since the hash algorithm needs it

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

Memory Organization

- The file is read completely and data is copied to device memory with a single **cudaMemcpy** call
- The main kernel is called on this data, together with the target hash
- This means that the maximum number of passwords that can be hashed is

#passwords = gpu_global_memory / 54B

Memory Organization – failed attempts

At first, I was reading the dictionary in batches of 256 lines until EOF.

Profiling this strategy showed a lot of overhead for the multiple **cudaMemcpy** and **cudaLaunchKernel** calls that were executed, since everything was in a loop and data was overwritten in batches of the above mentioned size.

Solution? Reading the file in **one time** (if possible) reduces execution time significantly

Blocks and threads

The main kernel is called using a number of blocks based on the amount of passwords in the database. Each block has **256 threads**.

```
#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);
```

Main Kernel

The main kernel that is executed is doing the following:

- Get thread id
- **Each thread hashes one password based on its id**
- Each thread converts the digest into a hexadecimal string **while comparing it to the target digest**
- If they differ, the thread returns. The only thread that will “*survive*” the check, will set a variable indicating the index of the found password

Main Kernel – stopping condition

The stopping condition of the server checks (and returns) if:

- The current thread id in the block is higher than 256
- The working block is the last one and **the thread id is higher than the amount of passwords to hash**
- The variable used for the index of the found password is already containing a result

Main Kernel – optimizations

- **Shared memory** is not used since the operation to copy the input data into it only creates overhead: this is because I do not copy all the calculated hashes back to the host but I only ”return” one single integer value
- **Uses of registers**: both the main kernel and the functions for the SHA-1 algorithm uses local variables to speed up computation using registers
- **The stopping condition** checks if the variable containing the password index has been set: if it is, the computation for that specific thread doesn’t even start. This is an aleatory optimization but it gives an average time reduction of 50ms

Results vs CPU implementation

To perform a fair comparison the following rules need to be considered:

- The SHA-1 implementation is the same, just adapted for the GPU
- The CPU implementation is a basic one, that **do not use multithreading or extended ISA**
- The timings do not consider the reading of the dictionary
- **GPU timings include memory operations**

Results vs CPU implementation

The tests were performed under these conditions:

- Dictionaries were taken from the SecList repository, they contains commonly used passwords
- **Timings are measured under worst-case scenario**, meaning target hash represents the last password in the database
- Under best-case scenario, CPU implementation is always faster since it stops when the hash is found

Tested Systems

PC

CPU: AMD Ryzen 5 2600X 6 core

GPU: NVIDIA RTX 2070

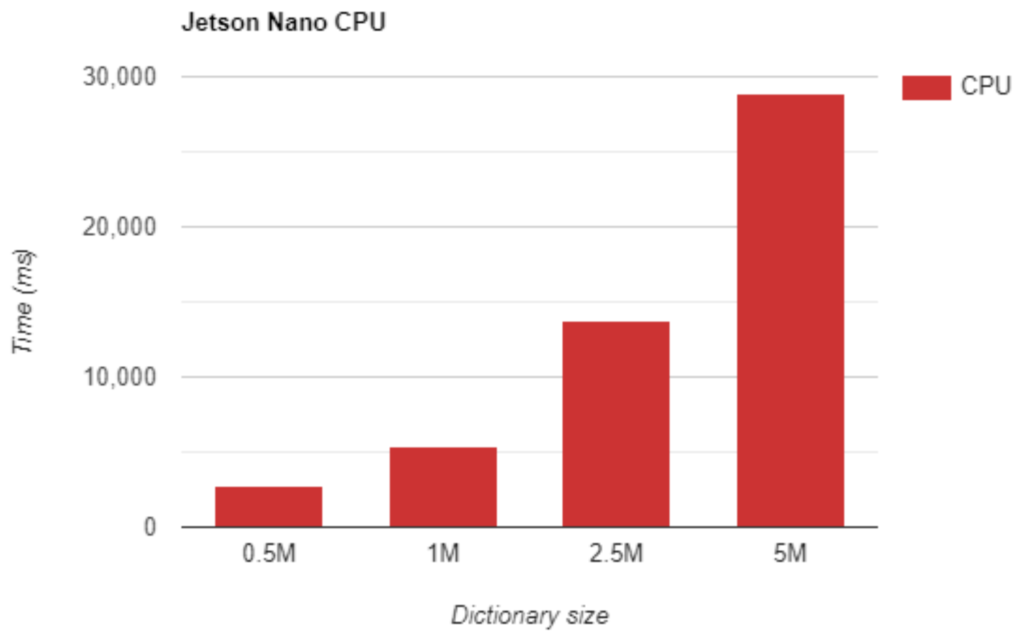
Jetson Nano:

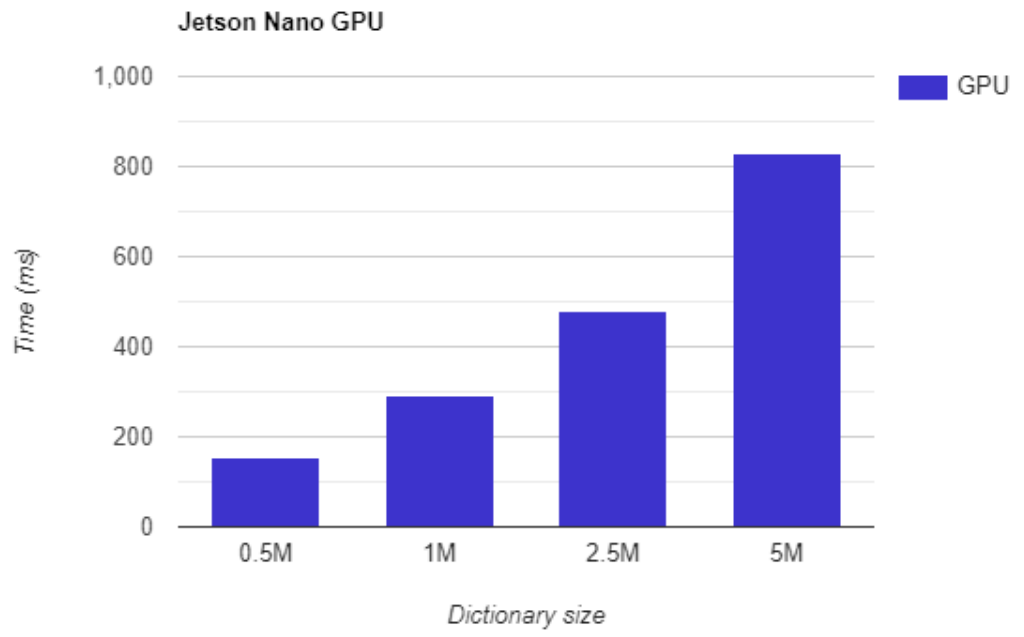
CPU: Quad-core ARM A57

GPU: 128-core NVIDIA Maxwell

Jetson Nano CPU vs GPU

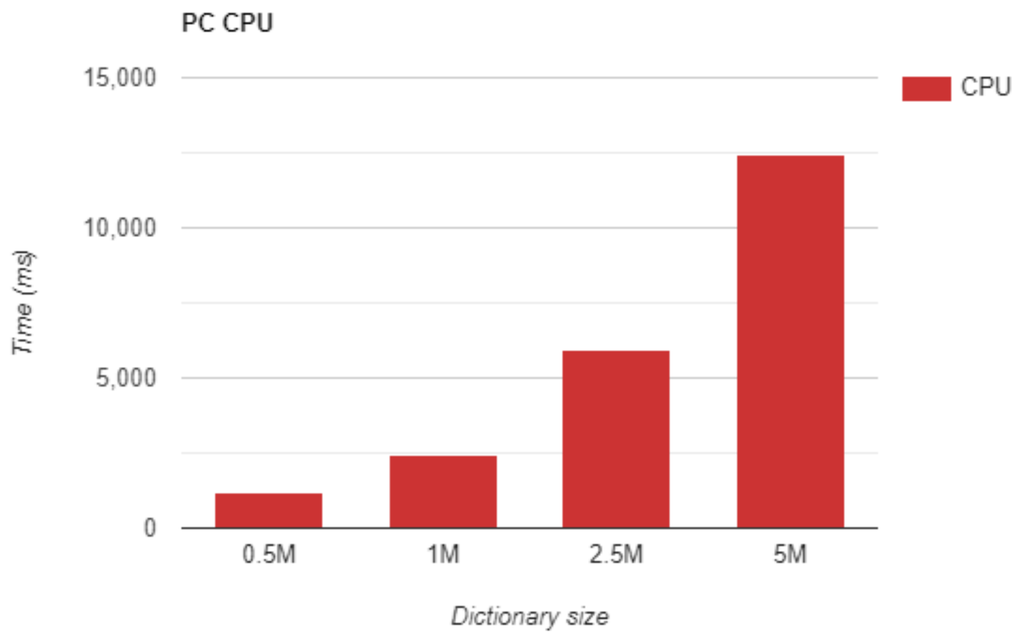
- CPU: 5 million passwords are hashed and checked in **29000ms** circa
- GPU: 5 million passwords are hashed and checked in **820ms** circa

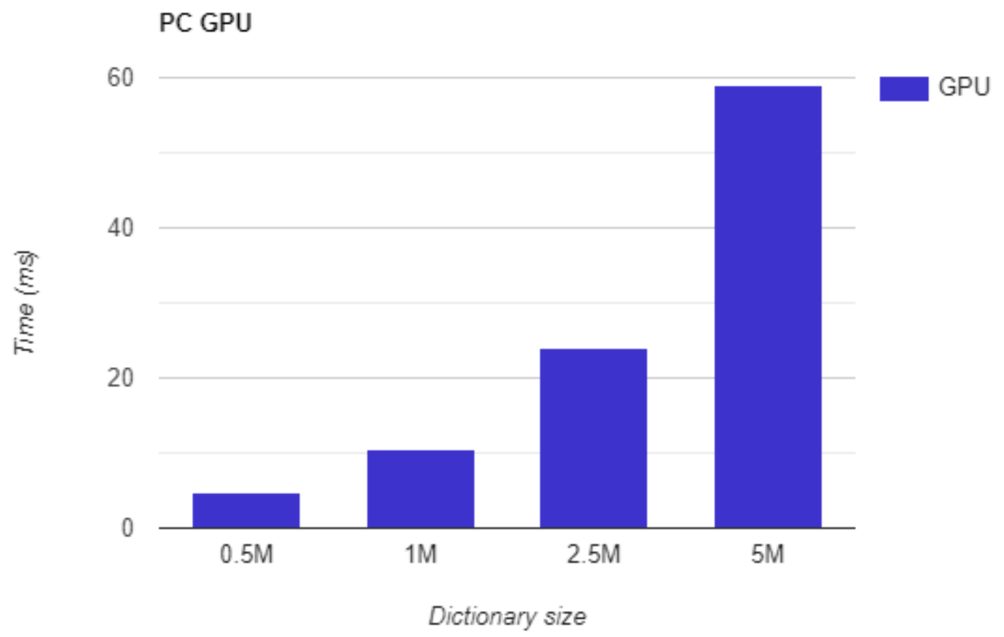




PC
CPU vs GPU

- CPU: 5 million passwords are hashed and checked in **12500ms** circa
- GPU: 5 million passwords are hashed and checked in **59ms** circa





Conclusions and Improvements

- The comparison, to be fair, should be done using a CPU implementation with multithreading or extended ISA
- Saved digests are usually salted in order to prevent (or make more difficult) a dictionary attack: an improvement could be handling this case
- If password is not found, a brute force implementation could be necessary: each thread could generate a password, hash it and compare it to the target digest