Breaking 53 bits passwords with Rainbow tables using GPUs

Steven Meyer
System of Communication
Master Thesis
September 2011

EPFL Supervisor
Philippe Oechslin
EPFL/LASEC

Corporate Supervisor
Cédric Tissières
OS Objectif Sécurité SA
# Table of Contents

1. Table of Contents .................................................................................................................. 2
2. Abstract .................................................................................................................................. 4
3. Introduction ............................................................................................................................... 5
4. Preliminaries ............................................................................................................................ 6
   4.1 Hashes ................................................................................................................................. 6
      4.1.1 MD4 .................................................................................................................................. 7
      4.1.2 Attacks & Weaknesses ................................................................................................. 7
4.2 Windows Passwords ............................................................................................................... 7
   4.2.1 Principles .......................................................................................................................... 8
   4.2.2 Attacks & Weaknesses ................................................................................................. 8
4.3 Rainbows Tables ................................................................................................................... 9
   4.3.1 Time Memory Trade-offs .............................................................................................. 9
   4.3.2 Rainbows Table ............................................................................................................ 11
   4.3.3 Ophcrack ...................................................................................................................... 12
4.4 GPU ...................................................................................................................................... 13
   4.4.1 Technology .................................................................................................................... 14
   4.4.2 CUDA ............................................................................................................................. 14
5. Aim of the Project .................................................................................................................. 17
6. Solutions .................................................................................................................................. 19
   6.1 Table Generation .............................................................................................................. 19
   6.2 The Binarization ............................................................................................................... 21
   6.3 Cracking ............................................................................................................................. 23
7. Implementation ....................................................................................................................... 27
   7.1 Generation ........................................................................................................................ 27
      7.1.1 Structure ..................................................................................................................... 27
      7.1.2 Block calls .................................................................................................................. 27
      7.1.3 Kernel .......................................................................................................................... 29
      7.1.4 Sort & Merge .............................................................................................................. 30
      7.1.5 Optimizations ............................................................................................................ 31
   7.2 Cracking ............................................................................................................................ 33
      7.2.1 Structure ..................................................................................................................... 33
7.2.2 The cuts ................................................................. 34
7.2.3 The Kernels ........................................................... 35
8 Hardware ........................................................................ 37
9 Results ........................................................................... 38
9.1 Generation ..................................................................... 38
9.2 Cracking ........................................................................ 39
10 Conclusion ....................................................................... 41
11 Work accomplished & Personal Contributions ...................... 42
12 Acknowledgements .......................................................... 43
13 Appendix .......................................................................... 44
14 Bibliography ..................................................................... 45
2 Abstract

The rainbow tables’ methodology has reduced the complexity of cracking a problem from hours to seconds. Since its presentation in 2003 it has been used to crack the security of Microsoft Office, Windows, GSM and various hashes. Although technology follows Moore’s law and performances double every 18 months, the complexity of passwords and hashes tends to grow much faster. For instance, by increasing a password by one character, its complexity roughly increases 100 times; justifying the use of a new technology to crack passwords. In view of this assessment, we use the power of GPU (graphical cards) that are very efficient and cost effective on simple problems such as MD4 nonlinear Boolean functions.

In this project we employ CUDA GPUs to generate rainbow tables which are able to break all Windows passwords of eight characters with 99% of success, and crack them up to 10 times faster than with a CPU.
3 Introduction

For more than 15 years, Microsoft Windows has used a derivative of MD4 to secure access to a computer. Although researchers have found various ways to attack the secured storage password, Microsoft has maintained its implementation and has asked end-users to select more complicated and longer passwords. To date, the most effective way to retrieve a password is to employ rainbow tables, which use a time-memory compromise requiring only two-thirds of the time needed for a brute force attack and two-thirds of the memory needed for a dictionary attack.

New technologies, such as graphical cards used for computing, have enabled simple algorithm to be parallelized to a significant extent. GPUs (Graphical Processing Units) will not make an algorithm compute faster; however they will permit more instances of an algorithm to run simultaneously. It is thus key to determine which part of the generation and usage of rainbow tables can be parallelized, and how to best optimize this technology’s new algorithm, given its benefits (many computing cores) and its shortcomings (heavy overhead for each instantiation).
4 Preliminaries

In this chapter we will cover the basic knowledge needed to understand the CUDA technology and the Windows password storing mechanism.

4.1 Hashes

Hash functions are part of the deterministic one-way function family that is efficiently computable in one direction but whose inverse calculation is hard. One purpose of a hash is to mathematically hide a string of characters. It generally takes an input (the message) of an arbitrary length and produces an output (the digest) of a specific length: even the smallest change in the message generates a radical change in the digest, so it is hard to generate a message from its digest. The notion of hard implies that there are neither shortcuts nor an optimized way to break the security of the hash function other than by undertaking a brute force attack and trying all the possibilities. Hashes are commonly used to protect the integrity of a message, for instance by comparing the digest of a message sent and received where the message would be sent on an insecure channel and the digest sent on an integrity channel.

A hash function \( h : \{0, 1\}^* \rightarrow \{0, 1\}^h \) has the following properties:

- **Preimage Resistance:** given \( y \in \{0, 1\}^h \) it is hard to calculate \( x \) such as \( f(x) = y \).
- **Second Preimage Resistance:** given \( x_1 \) it is hard to find \( x_2 \) (with \( x_1 \neq x_2 \)) such as \( f(x_1) = f(x_2) \).
- **Collision Resistance:** it is hard to find \( x_1, x_2 \) such that \( f(x_1) = f(x_2) \).

For a perfect hash function, a Preimage attack or a Second Preimage attack have to be done in brute force and have a time complexity of \( 2^h \). On the other hand a Collision attack requires only a time complexity of \( 2^{h/2} \) using the birthday attack (which states that by picking randomly 23 people, there is a 50% chance that two of them will have the same birthday date).

Salting is a security enhancement to hashes to avoid some precomputational attacks. Before hashing a message (generally passwords), we add (concatenate) to it a random value. A hacker would then need to break not only the message; but also the message with the salt, which is longer and therefore requires more resources. Generally, the salt is kept in clear text with the digest, but if it is kept in a secure location (unavailable to the attacker), it can then also be used to add entropy to the message. Another advantage of using salt in hashes is that two identical messages with different salts generate different hashes.

In most recent systems and databases we store passwords as hashes. The main advantage of this process is that the system will not store the password (it is thus not able to retrieve it nor reveal it), but is able to verify a given password by comparing the stored hash and the hash of the given password.
4.1.1 MD4

MD4 is a hash function based on block cipher designed by Ronald Rivest [1] at the MIT in 1990. Its initial purpose was to be used in digital signatures, but it has since been used in many other fields such as password storage or unique file labeling. In the 90s, the main benefits in using MD4 was its performance on a 32 bit machine, the small size of the substitution tables and compactness of the code, making it easily implementable in software and in hardware.

The MD4 algorithm performs the following steps:
1. Addition of a padding to the input such that the length in bit would be congruent $448 \text{ modulo } 512$.
2. Appending of a “1” bit to the end of the message and a 64 bit representation of the length of the input.
3. Initialization of the four buffers of 32 bits each for a total of 128 bits.
4. Computation of three rounds with each 16 nonlinear Boolean functions.
5. Update and output of the buffers.

4.1.2 Attacks & Weaknesses

When MD4 was first published, it was considered secure and claimed for a Preimage attack a complexity of $2^{128}$ operations and for a Collision attack a complexity of $2^{64}$ operations. Nevertheless, RSA expressed some reserve by stating that MD4 could only provide moderate security as, being exceptionally fast, it was "at the edge" in terms of risking a successful cryptanalytic attack.

In 2006 RSA¹ and Microsoft² recommended not to use MD4 anymore. Subsequently, in March 2011, the IETF declared the MD4 RFC (1320) obsolete due to the number of successful attacks on MD4³.

The first attack against MD4 was executed by Vaudenay in 1994⁴, then by Dobbertin in 1996 with a complexity of $2^{20}$ operations⁵. Later on, in 2004, Xiaoyun Wang demonstrated that a practical Collision attack could be done on MD4 with a complexity as low as $2^9$.

The first Preimage attack against MD4 was accomplished in 2008 by Laurent when he established that MD4 was not a one way function and could be broken in $2^{100}$ operations. In 2010 MD4 was shown vulnerable to attacks with a complexity of $2^{69.4}$. As a result, it is no longer recommended to use MD4 to hash keys or passwords longer than 70 bits, as MD4 would weaken their strength.

4.2 Windows Passwords

In Windows 9x, the logging password is inconsequential since logging in is optional and the computer’s resource can be accessed without the password by simply canceling the login request. Note however that the Windows 9x platform is “secured” by LMHash password storing developed by IBM.

² http://msdn.microsoft.com/en-us/magazine/cc163518.aspx#S6
⁴ http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.56.9232&rep=rep1&type=pdf
⁵ http://www.springerlink.com/content/pkunhyvv7e5d9tbd/
Windows NT, on the other hand, uses NTLM based on MD4, a more secure method to store passwords. Windows NT was subsequently merged with the end-user version of Windows to create Windows 2000. Nevertheless, for backwards compatibility issues, Microsoft maintained the use of LMHash by default until Windows Vista, calculating and storing the password with LMHash. The LMHash and the NTLM passwords are stored in the SAM (Security Account Manager), which can be found in %SystemRoot%\system32\config directory. When a system boots, it is impossible to gain access to the file, as the system sits on it and restricts its access. Usually, the best way to have access to the file is to boot on another system. To limit access to the SAM, Microsoft added to Win NT4 SP3 the SYSKEY that encrypts the SAM with a key. It can be stored locally (which is useless), stored on an external drive (which requires to insert the drive at logon while entering the password), or derived from a password (which requires entering two passwords at logon).

4.2.1 Principles

The LMHash works as follows:

1. The password is uppercased
2. The password is set to 14 characters (padded with white space if too short or truncated if too long)
3. The password is separated in two blocks of seven characters.
4. The two blocks are used to create a DES key that will encrypt the string KGS#$% which results in two 8 byte ciphers texts.
5. The two ciphers are concatenated to create the LMHash value

In Windows 9x, the password was not used to authenticate the user, but to give access to encrypted files on the user’s profile, such as email or dial up passwords, and to connect to shared resources on other desktops. For Windows NT, 2000 and above, the LMHash was stored in the SAM (Security Accounts Manager).

The NTLM works as follows:

1. Each character of the password is converted to UTF-16 Unicode (for example adding 0x00 after an ASCII character).
2. The MD4 hash function is fed with the password.
3. The 16 bytes output is then stored on the system.
   The NTLM hash is always stored in the SAM.

4.2.2 Attacks & Weaknesses

The LMHash is an old technology with many weaknesses and flaws. The first and most obvious problem is the conversion of all characters to uppercase: this weakens the choice of strong passwords and significantly facilitates a brute force attack as it removes 26 characters from the set of 95\(^6\) from which a user could select a password. The 14 character length password is a weakness too: if the user picked a longer password, it would be truncated; if it was shorter, it would be padded with white space which, as explained below, causes problems. The most important problem is splitting the password into two sets of seven characters before hashing, as

\[\text{bytes} = (8 + 16) \times 95^{14}\]

---

\(^6\) We will assume 95 characters as it is the standard ASCII set on a US keyboard

\(^7\) [http://www.wolframalpha.com/input/?i=%288%2B16%29*95^8+bytes](http://www.wolframalpha.com/input/?i=%288%2B16%29*95^8+bytes)
it would simplify the work of a malicious person who would only have to find a password in a set of \(2 \times 69^7 = 14 \times 10^{12}\) possibilities instead of \(69^{14} = 55 \times 10^{24}\) possibilities. Another problem in the splitting of the password and the padding is that if the users’ password would be shorter than eight characters, then the second half of the password would be filled in with white spaces, making the hash very easily recognizable.

With today’s computational capabilities, an LMhash password can be broken in a matter of seconds!

The NTLM is a much more secure solution than the LMhash but has its own flaws. Since it is based on MD4 hashes, any weakness on the Hash algorithm is passed on NTLM. In NTLM, passwords are never salted before being hashed, which makes them vulnerable to rainbow table attacks. Another easy attack on a computer is perpetrated by replacing the digest of the unknown password of a user by a digest of a known password of the attacker in the SAM (the usage of the SYSKEY would not avoid this problem): the attacker can then login as the user, but won’t have access to encrypted files.

4.3 Rainbows Tables

In this chapter, we discuss the time Memory Trade-offs and Rainbow tables as an elegant solution to reveal passwords from hashes.

4.3.1 Time Memory Trade-offs

We can imagine two extreme scenarios to break encrypted or hashed values. First, the brute force solution, where we try all the different input possibilities and compare them to the hash/encrypted value until the correct one is found; this solution is very memory efficient since we only need enough memory to store the encrypted/hash value and our generated one, but it is not at all time efficient. The second solution is a dictionary attack by precomputing all the possible values and storing them on a drive: when confronted with an encrypted/hash value one only needs to make a lookup in the drive to find the correct input. This second solution has the inconvenience of needing a very big drive (for finding all passwords of eight characters long with 95 different characters we need \(1.592 \times 10^{17}\) bytes = 159.2 PB ≈ 1.7 \times \text{estimated data content of the deep web (≈91000 TB)}\).

In 1980 Martin Hellman suggested to break DES by using a precomputed table that would offer a compromise between time and memory [2].

Let’s assume that we are looking for the input of a hash function by only knowing the function and the output (we could do the same to find a key to a ciphering function with a known plaintext attack).

Just for the recall, a hash function has the following elements: a message \(m \in\) the set of all possible messages \(M\) of length \(k\); and a one way function \(h(m) = d\) with the digest \(d \in\) the set of all possible digest \(D\) of length \(l\). To use the time-memory tradeoff, we need a reduction function \(R_i(j) = n \mid j \in D, n \in M\). We use a set of reduction functions to convert the digest \(j\) into a message \(n\) according to the parameter \(i\) in a deterministic way (mainly it has to have the good length and the good character set). Therefore we have that \(R_i(h(m)) = n\) with \(m, n \in M\); for

\[7\text{http://www.wolframalpha.com/input/?i=%288%2B16%29*9}^{5^8}\]
simplicity we define $f_i(m) = R_i(h(m))$ that maps a message candidate to another message candidate as a hash-red for hash reduction combo. We call a chain of length $t$ the connection of $t$ hash-reds: $m_{k,j}^t$ means that we take the message $m_k$ and do operation $f_i$ recursively $j$ times (see Figure 1).

Figure 1: Time-memory tradeoff with $r$ tables of chain length $t$ with $n$ entry messages

To generate the table we simply pick $n$ messages $m_i$ and recursively compute the function $f$ on them, but instead of storing the whole table, we just store the first message $m_0$ of each line and the last one $m_t$. Then to retrieve the message from its digest, we first take the reduction of the digest $R_i(d)$ then do recursively the function $f_i$ on it; at every iteration, we compare our result to $m_{k,t} \forall k < n$. If for the iteration $s$ we get that $f_i(R_i(d))^s = m_{k,t}^s$ then we know that a possible message $m \mid h(m) = d$ is on the line $k$ and that its value $f_i(m_{k,0}^t)^{t-s}$ is easily computable.

The reduction function is an arbitrary function that transforms an element of length $l$ to an element of length $k$; since $l > k$, then collisions are possible, even if the starting messages are different. This results in all the rest of the row being identical, therefore reducing the number of messages covered by the table (this causes a merge). The probability of finding the good message in a table is given by:

$$P_{\text{table}} \geq \frac{1}{N} \sum_{i=1}^{n} \sum_{t=0}^{t-1} \left(1 - \frac{t}{N}\right)^{j+1} [2]$$

where $N$ is the number of possible messages, $n$ is the number of rows, $t$ is the length of a chain. We can notice that at a given point it is inefficient to continue adding rows as the probability of collision and merges increases. That is why it is important to use different tables with different reduction functions that will not merge even if there is a collision in messages. By using $r$ tables we get:

$$P_{\text{success}} \geq 1 - \left(1 - \frac{1}{N} \sum_{i=1}^{n} \sum_{t=0}^{t-1} \left(1 - \frac{t}{N}\right)^{j+1}\right)^r$$

The time price to generate the tables would be the number of chains, multiplied by the length of the chains, the number of tables and the price of each operation $n \times t \times r \times f_i$. The memory price would be $2$ multiplied by the chains, the number of tables and the number of characters per message $2 \times n \times r \times k$. 
<table>
<thead>
<tr>
<th>Strategy</th>
<th>Preprocessing</th>
<th>Memory</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute force</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Dictionary</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Tradeoff</td>
<td>$O(t \times n \times r \times f_i)$</td>
<td>$O(2 \times n \times r \times k)$</td>
<td>$O(r \times \frac{t}{2} \times f_i)$</td>
</tr>
</tbody>
</table>

Figure 2: Comparison of extreme strategies and the tradeoff

4.3.2 Rainbows Table

Since Hellman’s proposition of time-memory tradeoff, there have been some propositions of optimization such as the distinguished points [3] which requires the last element of the chain $m_{k,t}$ to have some specific characteristic (such as finishing with five 0): this way the system does not request a drive access at every step of the search to look up if the message is a potential end of chain.

A proposition that has really boosted the performance of the time-memory tradeoff is the concept of rainbow tables presented by Philippe Oechslin in 2003 [4]. Instead of using several different tables with each having its own reduction function, it uses only one big table where in each column the reduction function changes (and if we assign a color to each reduction, we get a rainbow see Figure 3).

Figure 3: Illustration and comparison of traditional tables and rainbow tables

This way, even if a collision happens in the same table, it won’t merge, since for a merge we need a collision at exactly the same column (the chances of a merge in case of collision is $\frac{1}{t}$). The probability of success is given by:

$$P_{table} = 1 - \prod_{i=1}^{t} \left(1 - \frac{b_i}{N}\right)$$ where $b_1 = 1$ and $b_{i+1} = N(1 - e^{-\frac{m_p}{N}})$

The structure of rainbow tables allows it to be much larger and longer, which accelerates the search. Indeed, we start by the end by doing $R_t(d)$ to the given digest $d$, then we do a lookup to see if the same value appears in the end of chain: if not, we continue with $f_i(R_{t-1}(d))$, then $f_i(f_{t-1}(R_{t-2}(d)))$, and so long and so forth. The total number of calculations needed is thus $\frac{t(t-1)}{2}$.
while with a traditional table we have $r(t - 1)$: if we consider $t = r$ then we can see that our rainbow table is twice as fast. In addition, the amount of calculations on a rainbow table increases quadratically up to $\frac{t(t-1)}{2}$ compared to linearly on a traditional table: thus, in an average scenario, the rainbow tables outperform all the more the traditional table.

![Diagram of a search](image)

**Figure 4:** illustration of a search. In this example we calculate hash-reductions until we arrive at $m_{3,t}$ that we find in our table, and map it to $m_{3,0}$ (details of the table lookup in Figure 14). We then do hash-reduction until we find $m_{3,2}$, which is our password.

### 4.3.2.1 Perfect tables

Perfect tables are rainbow tables on which we have removed chains that have merged. Compared to time-memory tradeoff tables, on rainbow tables it is very easy to find merged chains since they have the same end. As we will see later, it is important to sort rainbow tables by the end-of-chain to efficiently do the cracking, so the discovery of merges becomes trivial. It is therefore important to correctly estimate the number of merges that are going to happen in our calculation, to ensure that we will have enough chains left at the end to do the cracking (details of calculation in chapter 6.1).

### 4.3.3 Ophcrack

Ophcrack is the first distributed software using the rainbow table technology to break Windows passwords. It is developed by Objectif Sécurité under GPL license with the last version being 3.3.1, last updated in 2009.

Ophcrack is multi-platform which can break LMHash and NTLM either from the SAM or from a file. It can work as a stand-alone application or from a liveCD.

There exist a large set of tables that can either be downloaded for free or purchased, with password cracking going from 7 to 16 characters depending on the charset.\(^8\)

---

\(^8\) [http://ophcrack.sourceforge.net/](http://ophcrack.sourceforge.net/)
There are some alternative techniques to gain access to computers secured by the Windows login without having the password. One can replace the hash of the password in the SAM by another hash whose password is already known (the Kon-boot application has this functionality). The advantage of this method is its speed, as a replacement can be done within a second; its disadvantage, is that by overwriting the password, the person with the hacked account will notice the change (he won’t be able to login); also, the attacker looses the advantage of logging in with the same password on additional accounts (email, Facebook, etc...), assuming that the same password is used on these accounts; and finally if the password is used to encrypt some information on the system (with EFS) then the encrypted data is lost.

4.4 GPU

The Graphics processing unit (GPU) is a dedicated circuit on a computer used to compute and generate the image that is displayed on the users’ screen. Gaming software companies, movie companies and medical research have pushed computer manufacturers to produce better and more powerful GPUs in order to create more realistic and precise images. In the 80s, the GPUs were mainly rendering 2D bitmaps; then, in the beginning of the 90s, 3D cards started to appear and, at the end of the 90s, NVidia produced a GPU capable of calculating transformation and lightning that reduced the load of the CPU by delegating this heavy computational operation. The graphical card used a very cheap set of chipsets able to do some simple computing in highly parallelizable manner by doing the same kind of operation on many elements of the images. The GPU evolution then came naturally by allowing the CPU to delegate to it most of the rendering computation.

GPUs were initially programmable through APIs such as DirectX or OpenGL that are purely imaging oriented: however, developers discovered the power of GPUs and, wanting to use it to solve computational problems, went through the API, transforming their computational problem into traditional image rendering that could be understood by a GPU.

The GPU manufacturers saw a market opportunity with a new kind of parallel computing and started creating GPU cards dedicated to computing, adding some specific API which allowed to run native code on the cards.

---

9 http://www.piotrbania.com/all/kon-boot/
There are two main developer frameworks for GPUs: CUDA\textsuperscript{10} (Compute Unified Device Architecture) that is developed and maintained by NVidia; and OpenCL\textsuperscript{11} (Open Computing Language) that was initially developed by Apple but is now under the control of the Khronos Group. The capacity of both frameworks are roughly the same, with main differences being that OpenCL can run on more platforms and CUDA is heavily pushed by NVidia with good support, documentation, examples and integration with other products and having analysis and performance tools.

4.4.1 Technology

What makes the GPU technology different from the CPU is the parallelism paradigm. CPUs tend to parallelize instructions on some data (Multiple Instruction Single Data) with pipeline technology that is very complex since it must “guess” on what data it can perform several instruction simultaneously; on the other hand, GPUs parallelize data on the same instruction (Single Instruction Multiple Data), therefore minimizing the problem of controlling the instruction and, since the data is read in “logical” sequence, almost removing the need of cache (see Figure 6). The GPU thread is executed in a self-contained mode that frees the CPU and allows it to do some other work. For example, at the same price one could buy an Intel I3 540 with two very high performing cores or a GeForce GTX460 with 336 low performance (CUDA) cores. Even though the GPU threads are slower, the number of simultaneous threads largely compensates for this.

![Figure 6: comparison between a CPU and GPU. The GPU has many simple Arithmetic logic units (ALU), almost no cache and a very simple control system (image provided by NVidia)](image)

4.4.2 CUDA

CUDA integrates with most of the popular programming languages such as C, IDL, Python, Perl, Fortran, Java, Ruby, Lua and MATLAB by giving virtual instruction and memory access to interact with the GPUs.

\textsuperscript{10} http://www.nvidia.com/object/cuda_home_new.html
\textsuperscript{11} http://www.khronos.org/opencl/
When developing in CUDA, there are some notions that have to be considered and some vocabulary that has to be known. When we talk about a host execution, it implies code executed by the CPU; device execution implies that it is run on the GPU; the GPU code is called the kernel. Threads on a GPU are much smaller than on a CPU and un/loading threads is not a costly operation, therefore a pending thread can be put to sleep to allow another thread to work. In CUDA threads never live alone, they are handled in groups of 32 that are called warps, which are the smallest SIMD operation possible on a Device; warps are then set into blocks, and the blocks are loaded into grids.

The separation into blocks and grids is mainly to define the way threads are executed and the way memory is used. Every thread in a block has a local memory allocated only for it (registers of the thread) and, since the total memory is a constant, the quantity of memory available per thread varies according to the number of threads in a block. On the other hand, the latency for changing threads in the same block is really small and does not generate overhead compared to a change between threads in different blocks; the developer has therefore to make some fine tuning of the distribution of threads in block and in grid in order to optimize the performance of its application. There are several types of memory, with the usual compromises to consider, such as a bigger memory having a slower read/write access and a greater overhead. In the speed order, we first have the registers, then the shared memory that is as fast as register if there is no read/write conflict, then there is the constant memory that is GPU read only but has the advantage to be cacheable and therefore very fast, finally we have the local and global memory that are in the DRAM and have no caching possible (see Figure 8).
To compile a code written in C/C++ and CUDA we use the compiler nvcc; which recognizes some specific annotations that describe the handling of the memory or the resource needed by the code (host or device). The compiler separates the code into a host code that will be compiled by gcc and a device code that will be compiled by nvcc, then it links both and adds the CUDA runtime libraries.

The CUDA framework comes with a visual profiler that helps the developer understand how his application works and what kind of optimizations are still possible. With the profiler we can see the occupancy: the ratio between the number of active warps per multiprocessor to the maximum number of active warps; between the number of registers per thread and number of shared registers; between the CPU time, GPU time and the memory transfer time. More details on CUDA technology and programming can be found in the book [5] and illustration about the occupancy can be found in Chapter 7.1.5.
5  Aim of the Project

In this project we use the power of GPU computing to increase the performances of the Ophcrack software. This implies understanding the GPU technology and estimation of its capabilities, proposing a solution that is reasonable in price and time, creating a rainbow table generator using GPU technology, generating rainbow tables, and finally implementing cracking software that can use the newly created tables.

![Figure 9: approximation of time to create table depending on the character set.](image)

We wanted to have at least one table generated at the end of the six month project. Thus, after a general analysis of CUDA performances, we concluded that our cracker had to be lower than 55 bits (see Figure 9). We ultimately decided to do a cracker for an eight character long password, as it has been in the past considered as a good minimal length in policy settings\textsuperscript{12}, with all the standard ASCII graphical charset\textsuperscript{13} of 95 different characters since it would cover the four different complexities of passwords (low caps, high caps, numerical & special character).

![Figure 10: standard ASCII 95 graphical characters](image)

\textsuperscript{12} http://technet.microsoft.com/en-us/library/dd277399.aspx

\textsuperscript{13} http://tools.ietf.org/html/rfc20
One could argue that there are many more characters that commonly appear on keyboards (such as the accents: éâèôç) but most of the companies discourage the usage of these characters in passwords as it might be complicated to logon with another computer that would not have the correct keyboard.
6 Solutions

6.1 Table Generation

Since the generation of the rainbow table is very time consuming, it is important to mathematically verify our hypothesis before starting the computation, in order to ensure that we will get the expected results. To do so, we created a spreadsheet that makes the predictions for us (see Figure 11):

- N: represents the number of password bits that the rainbow table has to break; since we want all 95 keyboard character: 0123456789abcdefghijkmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&'()*+,-./:;<=>?@[^`\{|}~
and with a password eight characters long we get \( \log_2(95^8) \approx 53 \) bits password to break.
- t: is the chain length: the longer the chain length is, the more Hash-reds will be hidden between the start-of-chain and the end-of-chain, and the bigger the amount of computations to be done during the cracking phase.
- P: is the success probability of finding a password in our rainbow table. Because of the collisions and merges, going from a probability of success of 99% to 99.99% requires the table to be twice as big.
- l: indicates the number of different tables: even though with rainbow tables we have a much lower risk of collisions than with the traditional Hellman’s table, the best performance on a single rainbow table is 86% due to the quantity of merges, therefore it becomes necessary to create more tables. The difference between each table is again only the reduction function that has to be unique.
- m: represents the final size of our table. From [6] we have that \( m = 2^N \left( 1 - P \right)^t - 1 \).
- \( m_0 \): represents the initial value of m before the merges. This value is very important since it will define the input size on which we will do the computation. From [7] we have that \( m_0 = \frac{m2^{N+1}}{mt-2^{N+1}} \).
- cr: is a simple ratio that will be used in the future calculations that represents \( cr = \frac{mt}{2^N} \).
- \( P_1 \): is the success probability of a single table. From [6] we have \( P_1 = 1 - \left( 1 - \frac{m}{N} \right)^t \).
- effort: is a simple estimation of the quantity of operations to compute the tables given by \( \frac{m+m_0}{2} \times l \times t \). We can then estimate the quantity of time needed by multiplying the effort by the calculation speed.
• mean: represents the mean effort (in operations) to search and find in our tables. From [7] we have $\text{mean} = \left( 2(c_r - 1) + \frac{2-c_r^2}{c_r} \right) \left( \frac{t}{2c_r} \right)^2 \frac{1-e^{-c_r(l+1)}}{1-e^{-c_r}}$.

![mean work to search](image1)

![time to generate](image2)

![size of the tables](image3)

**Figure 12: several estimations depending on parameters $t$ & $l$ with a constant success probability (the smaller, the better)**

Given these plots, we could see that the number of tables ($l$) should be between 4 and 12 for the generation time to be reasonable, but smaller would be better for the search work. The chain length ($t$) should be big to avoid having to large tables (at the condition that $l$ is not too big to limit the search work). Thus we have decided to use $t = 100'000$ and $l = 4$. 
One way of reducing the time spent on generating tables is to avoid doing some hash-reductions on rows that will merge. To do so, we will do a cut: we stop at a given point of the generation, we remove the merged chains by sorting the list and removing the duplicates, and restart hash-reductions operations. The challenge is to find the optimal moment to do it during the generation. If it is done too early there won’t be enough merges and therefore not many rows will be removed; if done too late, many hash-reductions will have been performed for nothing:

\[
\text{Time}_{\text{total}} = m_0 \times \text{cut} + \frac{2^n}{m_0} \frac{c_0}{2} (t - \text{cut}) + \text{time}_{\text{sort}} + \text{time}_{\text{merge}}
\]

After calculation we have found that the optimal point to be at 40,000 hash-reductions giving us a gain of 3.7 days.

![Figure 13: illustration of the optimal cut to do the sort/merge](image)

6.2 The Binarization

At the end of the generation, we noted that the structure of the table did not allow an optimal usage: we therefore decided to transform it.

First we compressed the information by converting our end-of-chain to an unsigned long int with a bijective function; this is called the binarization process. Using ASCII, our eight characters are represented in 8 bytes; but actually we only have $95^8$ possibilities of combination of our eight characters, and $\log_2(95^8) \approx 53$ bits $\approx 7$ bytes; we could therefore store the same information using 7 bytes of an unsigned long int.

We then split our binarized value into a prefix and a postfix: storing each in a different file. The postfix was truncated to save storage space but, as a result we lost some data and when doing the search, the lookup results were less precise: we thus had more false positive. The prefix was used to create an index that indicated where to look for a postfix. Finally, we stored the start-of-chain in another file with the same offset than the postfix so, once we find a postfix, we know where the start-of-chain is (illustration of the usage of the binarized file in Figure 14).
To find the best parameters when taking into account the lookup time, the available space in the RAM and on the drive we developed a spreadsheet.

<table>
<thead>
<tr>
<th>post used</th>
<th>post</th>
<th>index</th>
<th>bin</th>
<th>start</th>
<th>Sum</th>
<th>collision</th>
<th>by index</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>23</td>
<td>3.683 GB</td>
<td>142.266 GB</td>
<td>355.666 GB</td>
<td>501.615 GB</td>
<td>6.87%</td>
<td>97</td>
</tr>
</tbody>
</table>

Figure 15: binarization table (in green are the parameters that the developer must enter)

As previously discussed, a password has $\log_2(95^8) = 52.559$ bits of information from which we use 23 for the postfix and the rest for the prefix. Out of the 23 bits that represent the postfix we only keep 16 (these two values correspond to post and post used in the Figure 15).

- The size of the index depends on $m$ (the number of entries on our final table as discussed in chapter 6.1); since the index will be using our end-of-chain, we need $\log_2 m$ bits to store a value, but for compositionality we round up to the closest byte: $\log_2(m) \approx 36.152$ bits $\approx 5$ bytes. So our index size is given by $2^{prefix} \times \lceil \log_2(m) \rceil$. It is an advantage to have the index loaded in the RAM while cracking, since it drastically increases performance.
- The size of the bin (for binary and containing the postfix) is calculated by multiplying the number of entries in our final table $m$ with the number of bits used by the postfix (16 in our case).
- The size of the start is calculated by multiplying the entries in our final table $m$ with $\lceil \log_2(m_0) \rceil$; we use $m_0$ instead of $m$ because start is indexed on $m_0$ values even if only $m$ values are still here at the end (because of the merges).
- Sum is simply the total space needed to store the whole table.
- The by index is the number of postfixedes that have the same prefix and therefore will be pointed by the same index (represented in the mid-tone color in Figure 14). Having a too big value is not recommended since each postfix under the same index implies access to the disk.
- The collision value represents the probability of having a collision and thus a false positive when doing the search because of the truncation of the postfix.

The calculation here is done for only one table, thus we should multiply the size of Index, Bin, Start and Sum by the number of tables used.
Given these plots, we could see that the *postfix* has to be at least 23 (given that the size of the index must fit in 4GB of RAM) but not too big to avoid too many collisions and search on the drive (by index). For the *postfix used* a too big value would create a too big bin file, but too small would generate too many collisions. We have thus concluded that the best option for our project is to set *postfix* to 23 and *postfix used* to 16.

To gain speed during the cracking, it is interesting to have the index in the RAM since it is quite small and is read for every potential end of chain. The Bin file is much bigger and won’t fit in the RAM in our case, but given the number of access (97 per potential end of chain in average), it is interesting to store it on an SSD that is much quicker than an HDD and has less latency.

### 6.3 Cracking

The search in a rainbow table is built on three consecutive phases: first we have to generate a possible endpoint of our digest; then we have to check if the newly generated endpoint exists in our table; and finally, if this is the case, we have to find the element just before our digest (thus the password) in the chain.

When retrieving a password from our rainbow tables, we have to correctly set different parameters and strategies for optimization.

It is interesting to note that with the rainbow table structure, it is much more cost effective to search the end of the table than its beginning. At the position $i \mid 0 \leq i < t$ we have to do $t$ –
\(i\) operations, which means that for approximately the same work we could either calculate the first row \((i = 0)\) or the \(\sqrt{2\ell}\) last ones \((i = \{t - \sqrt{2\ell}, t - \sqrt{2\ell} + 1, ..., t - 2, t - 1\})\); in a scenario of \(t = 100'000\), the work for the first row would be equal to cumulated work for the 446 last ones. As showed previously, the probability of finding our message at the start of the search is much higher than at the end; we can therefore see that with less than 10\% of the work the probability of finding the message is greater than 70\%. Therefore it might be worthwhile to start the search at the end of each table, process it backwards, then to search backwards the beginning of each table; this process is called the cut and is detailed in chapter 7.2.2.

![Figure 17: comparison of the work to do and the probability to find the password](image)

We have created a Gantt chart that can estimate the work that has to be undertaken. It uses characteristics of the table (such as its size and length), characteristics of the binarization (such as the size of the index, the collision probability and the by index), characteristics of the cracker (such as the cut) as well as characteristics of the computer (such as its drive speed and GPU speed).
Figure 18: Gantt chart representing the flow of cracking. We can notice that the process for 100 passwords is two times faster with the cut
Figure 19: Gantt chart representing the flow of cracking. We can notice that the process for 2 passwords is almost two times faster without the cut.

These plots are approximations and do not include other optimization done such as smallcut, passwordBlock and smallPasswordBlock explained in chapter 7.2.1.
7 Implementation

The implementation of our rainbow table solution is done in C/C++ and CUDA 4.0, which is the latest version of CUDA available (during the development phase we have used CUDA 2.3 since it enables device emulation that helps the GPU debugging and optimizations).

7.1 Generation

Rainbow tables are created in 7 separate phases:

1. Generation of a unique input as a start-of-chain.
2. Computation of the hash/reductions on the input.
3. Saving the start and end of chain to a file.
4. Sorting of the chain by its end.
5. Merging all the sorted files
6. If the table hasn’t been totally generated (and has been cut), then we restart at point 2 with the end-of-chain as an input.
7. Restarting at point 1 for a different table

7.1.1 Structure

At the launch of the generator, we first counted the number of GPUs (int nbrGPU()) to ensure the use of all possible resources during our generation. We then looped in every table, then again for every cut the chain generation; within this double loop we created a CPU thread (pthread_create()) for each GPU (it is therefore important to have at least as many CPU cores than GPU cards since a CPU will do some active poling and generate the star-of-chain for each GPU). Since Visual studio does not follow the POSIX standard we chose to use the Linux instantiation of the new thread; because of this, the code won’t be able to run on a Windows machine. As on each table we needed to have a unique input to avoid generating a collision, we sequentially generated all the possible strings in the alphabetical order. Then we actually calculated and saved the chains (details will be explained in the chapters 7.1.2 and 7.1.3), and finally we sorted and merged our chunks (details will be explained in the chapter 7.1.4). These operations were performed in parallel for each GPU.

7.1.2 Block calls

As explained previously, GPUs are very efficient when handling large data: it is therefore logical to give chunks of start-of-chain and computing the end-of-chain for all of them at the same time. The problem with big chunks of data is the overhead required to copy them from the memory of the CPU to the memory of the device, then back to the CPU and finally to the Drive. For the sake of efficiency, we therefore decided to interweave the operations. In order not to loose time, we used three identical structures (streams), with each of them doing a different asynchronous operation at a given time. We denoted each structure as a, b and c, and each one will do the following operations in this order (CPU)do Start-of-chain -> (BUS)copy Start-of-chain From CPU to GPU -> (GPU)calculate block -> (BUS)copy End-of-chain From GPU to CPU -> (GPU & BUS)wait for finish work -> (IO)write to file block.

The interweaved version follows this structure:
1. Do Start-of-chain[a]
2. Do Start-of-chain[b]
3. copy Start-of-chain From CPU to GPU[a]
4. loop
   a. copy Start-of-chain From CPU to GPU[b]
   b. calculate block[a]
   c. wait for finish work[c]
   d. write to file block[c]
   e. do Start-of-chain[c]
   f. copy End-of-chain From GPU to CPU[a]
   g. permutation c \rightarrow b, b \rightarrow a, a \rightarrow c
5. end loop
6. copy Start-of-chain From CPU to GPU[c]
7. calculate block[b]
8. wait for finish work[a]
9. write to file block[a]
10. copy End-of-chain From GPU to CPU[b]
11. calculate block[c]
12. wait for finish work[b]
13. write to file block[b]
14. copy End-of-chain From GPU to CPU[c]
15. wait for finish work[c]
16. write to file block[c]

From point 1 to 3, we did the initialization phase. During the loop, at every round we permuted c \rightarrow b, b \rightarrow a, a \rightarrow c. The wait for finish work[i] was here to ensure that process working on the queue i had finished all GPU work and transfers. For the steps from 5 to 16 we unrolled the pending operations.
7.1.3 Kernel

As explained in the Chapter 4.4.2, it was crucial for the GPU performance to not use too many registers per CUDA-thread in order to maintain an optimal occupancy (this being the ratio of the calculation time of the GPU and the elapsed time). We therefore tried to have as few parameters as possible when calling a kernel, in spite of small source code and fewer calculations.

The table is structured as the concatenation of all the start-of-chain and end-of-chain together without any separation. The kernel receives the table as parameter with only the beginning of chains, and returns the table filled with the end of chains.

The Kernel first converts the beginning of chain to Unicode by padding every character so \texttt{abcdef} is \texttt{0\times616263646566} in ANSI and becomes \texttt{0\times610062006300640065006600} and stores them in three unsigned \texttt{int} b0, b1, b2. At the end of the hashing, the digest is represented in four unsigned \texttt{int} a, b, c, d, that are used to compute the reduction with the actual chain iteration \texttt{i}, then mapped to the set of authorized characters \texttt{chars} of length \texttt{charLength}. It is important to note that to avoid an increase in the probability of collisions, the distribution of the reduction function is uniformed.

Figure 20: illustration of the overlapping in the block structure (done with NVidia CUDA profiler)

Figure 21: example of 10 hash-reductions. In red the beginning of chain, in green the end of chain
30

Implementation | Master Thesis

c = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&'()*+,-./:;<=>?@\^_`{|}~;

p0 = a ^ i;
p2 = b ^ (i >> 12);
p4 = c ^ (i >> 24);
p6 = d;

b0 = chars[p0 % charLength] << 16 | chars[(p0 >> 16) % charLength];
b1 = chars[p2 % charLength] << 16 | chars[(p2 >> 16) % charLength];
b2 = chars[p4 % charLength] << 16 | chars[(p4 >> 16) % charLength];
b3 = chars[p6 % charLength] << 16 | chars[(p6 >> 16) % charLength];

7.1.4 Sort & Merge

Having our rainbow table sorted by the end-of-chain is essential for detecting the merges and for the cracking. At the end of the generations, each GPU creates a file with its generated table (see Figure 21), each file is segmented into small files that can fit in the RAM and are sorted.

Given the size of the data to sort, instead of using the standard qsort in c (that has a performance of $O(n \log(n))$) we used multithread sort algorithm (that has a performance up to $O\left(\frac{n\log(n)}{p}\right)$ with $p$ being the number of threads). In our first implementation, we used the qsort_mt function written by Diomidis Spinellis\textsuperscript{14}: however, unfortunately, the algorithm was getting unstable with a set greater than 170 MB, generating errors. In a second version we simply used the standard qsort from our Linux implementation, which created one sorted file per thread: this solution was correct but not efficient since we had many more files to merge at the end. In the final version we used a multi-threaded qsort (dqsort) created by Bertrand Mesot that performs at 86% of the optimal performance.

\textbf{Figure 22: example of 10 hash-reductions sorted by the end-of-chain (in green)}

Once we sorted all our files (see Figure 22), we needed to merge them to get a final one. To do so, we used a sortbox that was sorted by all the first elements of each file, wrote the first entry of the sortbox, replaced it with the next entry in the file of provenance, resorted the sortbox and restarted. Since one merge couldn’t be distributed on several threads, to optimize it we merged a number of sorted files on each thread. We then had two options: we could either make a few merges on many files with a few threads with each merge taking a lot of time, or we could make many merges on a few files with many threads with each merge being quite fast (see Figure 23). This decision would depend on the performances of the computer doing the calculation.

\textsuperscript{14} http://libmt.sourceforge.net/
Figure 23: on the left is a single merge with a single thread on nine files. On the right are four merges in four threads (first three, then one) with three files each.

As a safety measure, after our problems with the previous algorithms, after every sort and every merge we verified that the list was correctly sorted with a small application that parsed the list and verified each entry.

7.1.5 Optimizations

At every step of the creation of our rainbow table we tried to find optimizations, such as the structure of block described in chapter 7.1.2. Here are some other optimizations we implemented. We used as much as possible of #define Directive for parameters that are employed in the whole application, instead of passing them into parameters through functions. We have set the set of authorized characters chars to a constant in the GPU to accelerate its access (see Chapter 4.4.2).

We have already discussed in 7.1.2 the importance of doing operations per block and of finding the correct size for the blocks. A block too big could take more time to load, unload and write than it would take for the generation and would thus reduce performance; on the other hand, a block too small would create an overhead that would be penalizing. After some tests we concluded that $4194304 = 2^{22}$ since it is important to use a power of 2 for it to be correctly separated between threads and blocks) could be a good value (see plot in Figure 24).
With 256 threads per block (the CUDA architecture recommends the use of a power of 2 for performance gain), we can use from 0 to 20 registers and keep an occupancy of 100%, while from 21 to 24 registers we keep an occupancy of 83.3%\(^\text{15}\) (see Figure 25). To enhance the performance of our function, we created four different kernels for the hash-reductions:

\begin{verbatim}
#include <cuda_runtime.h>
#include <iostream>
#include <vector>

using namespace std;

void doBlockCont(UCHAR *table, UINT4 chainLength, UINT4 tableID)
{
    // Code for doBlockCont
}

void doSmallBlockCont(UCHAR *table, UINT4 size, UINT4 chainLength, UINT4 tableID)
{
    // Code for doSmallBlockCont
}

void doBlock(UCHAR *table, UINT4 chainLength, UINT4 tableID)
{
    // Code for doBlock
}

void doSmallBlock(UCHAR *table, UINT4 size, UINT4 chainLength, UINT4 tableID)
{
    // Code for doSmallBlock
}
\end{verbatim}

\(^\text{15}\) With a NVIDIA card with a compute capability of at least 2.0 (which are all the cards produced since end of 2008)
For the merge function we performed simple optimization such as using read and write buffers. Instead of reading every new line and writing every line once sorted, we read by block and kept it in a buffer, then wrote the output in another buffer and flushed it to the file when full. These optimizations are inspired from [8].

7.2 Cracking

Given that the final objective of this research was to crack a password, we needed to create a second application, which was able to fulfill this purpose. The application developed in this project will be merged with Ophcrack, although, at present, it is mainly a proof of concept and a performance calculator. The password cracking is performed in the following phases:

1. Retrieving the hashes from the SAM (not done in this project).
2. Calculating all the possible end-of-chains.
3. Finding the correct beginning of chain.
4. Retrieving a candidate password for the given hash.
5. Verifying that it is not a false-positive.

7.2.1 Structure

In order to optimize the cracking process, we found two places where we could take advantage of the computing power of GPU: when calculating the end-of-chains, and when retrieving a password. For this we had to create blocks of data to be processed. We excluded the process of verification of false-positive because of its small size compared to the overhead of using the GPU.

As discussed in chapter 6.3, we had to reach a compromise in the length searched within a table. Indeed, while we should not search the whole table at once, since we lose efficiency as we go forward in the exploration, we also know that loading a table in memory can be very time consuming and that changing tables several times could lead to an inefficient search. We have thus defined a cut value that represents (if it exists) the optimal place to stop searching in a table and load another one.
We have also defined a `smallCut` that avoids generating too many end-of-chains, `smallPasswordBlock` that splits block the work for the GPU (which enables parallelization between CPU and GPU) and `passwordBlock` that limits the number of simultaneous passwords by taking into account the quantity of free memory, the number of passwords to search and the computational time.

A pseudo code of the structure would then be:

Loop For every `cut`
  Loop for every `table`
    Loop for every `smallCut`
      Loop for every `passwordBlock`
        Calculating all the possible end-of-chains.
      Loop for every `smallPasswordBlock`
        Finding the correct beginning of chain.
        Retrieving a candidate password for the given hash.
        Verifying that for non false-positive.
    End loop
  End loop
End loop

7.2.2 The cuts

Before doing the cracking we undertook a small benchmark to see if it was worth doing a cut in our table search. This benchmark takes into consideration the GPU speed, the disks speed and the number of passwords, then returns if it is worth cutting the tables or not. The formula to find the optimal cut (if the table is only cut once) wants to minimize the total work by finding the quantity of unknown passwords and multiplying it by the number of operations done:

\[
\begin{align*}
\text{cut}_{\text{optimal}} &= \min(cut \times \sum_{i=0}^{\text{nbTables}} \text{nbPass}_i + (\text{chainLength} - \text{cut}) \times \sum_{\text{nbTables}/2}^{\text{nbPass}_i}) \\
\text{nbPass}_i &= \begin{cases} 
\text{initial number of passwords} & \text{with } i = 0 \\
\text{nbPass}_{i-1} - \left(1 - \left(1 - \frac{1}{\text{chainLength}}\right)^{\text{cut}}\right)\text{nbPass}_{i-1} & \text{with } 0 < i \leq \frac{\text{nbTable}}{2} \\
\text{nbPass}_{i-1} - \left(1 - \left(1 - \frac{1}{\text{chainLength}}\right)^{\text{chainLength} - \text{cut}}\right)\text{nbPass}_{i-1} & \text{with } \frac{\text{nbTable}}{2} < i \leq \text{nbTable}
\end{cases}
\end{align*}
\]

With \(1 - \left(1 - \frac{1}{\text{chainLength}}\right)^t\) representing the probability of finding a password with at the cut \(t\) of the search and \(\text{nbPass}_i\) to number of unfound passwords at the iteration \(t\).

In our scenario of 4 tables and a chain length of 100’000 the optimal cut is at 33166.
The smallCut is here to optimize the number of passwords being cracked simultaneously: it takes into account the quantity of free memory and the number of passwords in order to decide how many passwords can be done in a block and the minLength and maxLength of the chain. If smallCut is too big then we may do too much computation for a password, which has already been found, and if it is too small then we may have some overhead with the GPU.

Finally, if the number of passwords to break is larger than what the memory can handle, then they are segmented into passwordBlock.

7.2.3 The Kernels

We want to generate all the potential end-of-chains to find if one matches the end-of-chain in the loaded table. This means that we had to make a chain of length 1,2,3 ... t − 1, t; this situation is not recommended when working with GPUs which require SIMD, since each different length of chains is handled by different instructions. We decided to do two end-of-chain searches per kernel with at each call one long chain and one short chain. We therefore combined in a kernel the chain t − 1 with the chain 1, the chain t − 2 with the chain 2, proceeding with all the other chains in the same way.

After having generated a potential end-of-chain, we did a lookup in the table and retrieved the start-of-chain if the entry existed (see Figure 14). Then we calculated the hash-reds until we
found the searched password (given the number of merges and false positive, this operation happened often). This second phase could also be done by the GPU (and was handled by another thread so the CPU could continue to look for the next start-of-chain Figure 28), by putting in a buffer the pending passwords to find and launching the kernel when the buffer was full. We used the same technique as previously by joining the lookups two by two (one short and one long); and although all the length were not the same, it was still a good optimization.

Figure 28: illustration of the Loop for every smallPasswordBlock. On the upper plot we have a single threaded cracker. On the lower plot we have one thread for finding the start-of-chain (drive access) and one thread for finding the password (GPU). The white spaces are CPU work. We can notice that the size of doBlockEnd increases with time as it starts with short chains, and the size of doBlockStart diminishes with time as it starts with the largest chains.

Finally we verified if the found password was correct by hashing it and comparing the result to the given hash (this operation was done on the CPU as the overhead would have been too big).
8 Hardware

To do the computation, we purchased a dedicated computer with:

- 1 eVGA Classified SR-2 motherboard.
- 2 Intel Xeon Processor E5645 with a total of 12 cores and 24 threads.
- 48 GB of DDR3 SDRAM
- 3 2TB HDD 7200 rpm
- 1 160GB SSD
- 3 nvidia GeForce GTX 590 with a total of 6 GTX 500 GPUs, 9 GB of memory, 3072 CUDA cores

The machine runs Debian 6.0 without the graphical interface and the compilers are gcc 4.4.5 and nvcc for CUDA 4.0.
9 Results

9.1 Generation

During the previous table generation made for Ophrack, (VISTA8 chapter 4.3.3), the generator based on an optimized MD4 for CPU could compute $1 \times 10^9$ hash-reduction per seconds with a quad core xeon CPU. In this project, we can compute $8.75 \times 10^8$ hash-red per seconds with a single GPU giving a total of $5.25 \times 10^9$ hash-reductions per seconds with our machine.

Given that our machine has 48GB of RAM, we decided to sort blocks of 40GB. The sort was distributed (as explained in chapter 7.1.4) on the 24 CPU threads, taking 28 minutes per block.

Finally, for our merge, we found there might be two bottlenecks, which were either in the access to the drive for reading and writing, or in the CPU calculation for the sorting. After some testing on our machine we found that the best solution was to merge 3 files per thread (as illustrated on the right side of Figure 23).

![Figure 29: estimation of generation time](image)

For our first table we generated a chain length of 30’000 instead of the 40’000 optimal length (found in chapter 6.1), making the generation 5 hours longer in total but enabling us to determine whether there was a mistake in the generation at the first cut 4 days earlier. Our merge estimation was quite good as instead of having 30500 blocks in the second cut we had 30545 which is less than 0.15% of error.

During the project, we had to restart the generation once after the first cut because of two bugs found in our application. There was first a problem in the merge buffer (chapter 7.1.5) with an incorrect offset, then the first multi-thread sort algorithm used (chapter 7.1.4) had an issue with big datasets.

At the end of this project, due to a yet unknown error, instead of having generated a single table (out of four) with 68% of success rate, we have generated a smaller table that had a theoretical success rate of 35%.
9.2 Cracking

The cracker programmed during this project is a proof of concept: through it we checked if by using the power of GPUs to exploit a rainbow table we could find a speed benefit to the cracker.

For the performance tests, we compared the performance of the Ophcrack cracker 3.3.1 with the GPU cracker using rainbow tables from the VISTA8 set. Our tests were done using an Intel Core 2 Quad Processor with a GeForce GTX 460 (336 cores) machine as both material are high-end and affordable. To check the performance in different scenarios, we undertook the tests with different number of passwords (there are eight tables with a chain length of 55'000 in VISTA8).

We noticed that, for an unknown reason, the work on the GPU during the Retrieving of a candidate password was much longer than expected and does not at all correspond to the estimation in Figure 18.

![Comparison between CPU and GPU cracking with vista8](image)

*Figure 30: with passwords equally dispersed on the four tables at random chain length. Details of calculation in the Appendix.*
We can notice than the speed generally increases with the number of passwords. When we were cracking one password, the performance varies from a speedup range of 0.7 to 11.4; this is mainly due to the good performance of the GPU when the password is at the beginning of the table rather than in the end.

The good performance comes also from the parallelization of the work of the GPU and CPU, which tries to avoid one waiting for the other and would create a bottleneck.

During the testing we had problems due to the caching of the drives by the OS. To ensure that this problem did not compromise the results, we alternatively tested an execution on the CPU and GPU, and we copied random files from the drive between the experiments.

We have also tested the cracker with our newly created table (that is single table and only has a theoretical success rate of 35% as explained at the end of chapter 9.1); the average success rate observed is 22% and the average cracking time is 23 seconds per password on 100 passwords. It is interesting to note that with a bigger table and therefore a better success rate, the cracking would be much faster since it would not need to continue cracking for found passwords.

Since all the calculations are done by block, there is systematically too much work done. This value depends on the position of the password in a block and the position of the block in the table. In a best case scenario, the password found would be the last one of the block and no extra work would have been done. In a worst case scenario, the password would be the first one of a block and the extra work could go from 145’000’000 hash-red (99.93 of the total work) for the first block to 955’000’000 hash-red (19% of the total work) for the last block.

The mean work to break a 53 bit password with brute force is $6.6 \times 10^{15}$ operations, and the mean work to break the same password with a rainbow tables requires $2.7 \times 10^9$ operations; this implies that rainbow tables require roughly 1’230’000 less operations than brute force to break a password in average. If we compare the speed difference for cracking a password with brute force ($8.78 \times 10^8$ hashes per second) and rainbow table ($2 \times 10^8$ hash-red per second including the loading of the tables and the access to the drives) we can see that rainbow tables are still 280’000 faster than brute force. In theory we would therefore need 44 days to break a password with brute force and 14 seconds with rainbow tables.
10 Conclusion

During this project we have successful increased by 8.75 times our computational speed for the rainbow table generation and by up to 10 times the computational speed for the cracking. We have also started the creation a rainbow table that can crack passwords 280'000 faster than with brute force.

The GPU technology can provide a very good performance but works with a different paradigm than CPU so different factors need to be accounted for before jumping into it:

The operation done by the GPU should be simple; doing a division or a modulo (such as during the reduction) slows down the performance, but on the other hand operations such as nonlinear Boolean functions perform extremely well.

The quantity of memory available for each thread is relatively small. It is therefore important to keep the number of variables low to ensure that occupancy stays maximal. A performing way to do so is to create different functions for different variables states, this way we can save a few GPU register.

Coming from a graphical environment, GPUs expect to have the same operation to perform on different memory segments. Different operations, i.e. operations not having the same length (such as the situation of the cracking) or conditional statement in the code (such as if), are not well parallelized and will jeopardize the performance of GPUs.

Using a GPU necessarily creates an overhead, because of the initialization of the GPU and the memory transfers between CPU and GPU and back. Therefore one should only use the GPU if there is a big chunk of data or many operations. During the project we often did more work than necessary (while finding the potential password during the cracking) to ensure that the overhead would not decrease performance. The creation of block is fundamental when working with GPU (since an image wouldn’t be a simple pixel but a matrix) and must be taken into account as early as in the design phase of an application development.

In June 2011, David Graham from Errata Security published a research about GPU cracking; he explained that even though GeForce cards (CUDA) outperform Radeons cards (OpenCL) when doing mathematical GPU computing, Radeons have specific instruction for integers (generally used in cryptography) and VLIW technology that make it up to 3 times faster when cracking passwords\(^\text{16}\). It could therefore be interesting to port the generator and the cracker from CUDA to OpenCL in a future study.

If one were to develop a new hash algorithm that would be GPU cracking resistant (by not simply increasing the number of rounds), it would require a lot of memory; for instance, it would need to store all the previous states of the variables and should also function with complicated operation such as modulo, division and square root on floating point values. Finally, it should also use conditional and jump statements that would cause branching to the GPU workflow. This way, the GPU will not be able to take advantage of its architecture to efficiently crack passwords.

11 Work accomplished & Personal Contributions

In this project we have worked with existing content and strategies developed by Objectif Sécurité. The added value of this project is the following:

- Understanding of GPU technology.
- Benchmark of GPUs and performance analysis.
- Creating a spreadsheet to estimate rainbow table capacities and performance as well as to find the optimal parameters.
- Implementation of a GPU powered rainbow table generator.
- Starting the creation of rainbow tables which are able to break all Windows passwords of eight characters with 99% of success.
- Implementation of a “Binarizer” for big rainbow tables.
- Implementation of a cracker using GPU for the Vista8 tables and the newly created tables.
- Analysis of the performance of the cracker.
Acknowledgements

I would like to thank Philippe Oechslin and Cédric Tissère, for their supervision of my work. The permanent help, the personal and financial investment made by Objectif Sécurité allowed this project to be successful. Thanks to Bertrand Mesot and Sébastien Mathieu for all their help during my work with them and their patience with my development problems. Finally, I would like to thank my family for their unconditional support.
13 Appendix

In the chapter 9.2 we have compared the performance of the CPU Ophcrack application and the GPU proof-of-concept.

To run the Ophrack, we have used the following command:

```
./ophcrack -f pwd -n 5 -p 1 -g -b -d /mnt/ssd/scratch/tables/ -t
vista_eight,0,1,2,3,4,5,6,7 -u
```

For the experiment with one password we have chosen ten random passwords and tested one at the time. For the experiment with ten passwords, we have sequentially taken four blocks of ten passwords and tested a block at the time. For the experiment with 100 / 1000 passwords, we have taken one block of 100 / 1000 passwords and tested the block:

![Figure 31: cracking time for vista8](image)

<table>
<thead>
<tr>
<th>set</th>
<th>password</th>
<th>GPU</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 password</td>
<td>falsh070</td>
<td>219 sec</td>
<td>294 sec</td>
</tr>
<tr>
<td></td>
<td>falsh330</td>
<td>9 sec</td>
<td>32 sec</td>
</tr>
<tr>
<td></td>
<td>falsh168</td>
<td>39 sec</td>
<td>27 sec</td>
</tr>
<tr>
<td></td>
<td>falsh119</td>
<td>30 sec</td>
<td>51 sec</td>
</tr>
<tr>
<td></td>
<td>falsh992</td>
<td>6 sec</td>
<td>18 sec</td>
</tr>
<tr>
<td></td>
<td>falsh746</td>
<td>179 sec</td>
<td>437 sec</td>
</tr>
<tr>
<td></td>
<td>falsh810</td>
<td>126 sec</td>
<td>830 sec</td>
</tr>
<tr>
<td></td>
<td>falsh522</td>
<td>237 sec</td>
<td>2699 sec</td>
</tr>
<tr>
<td></td>
<td>falsh238</td>
<td>10 sec</td>
<td>51 sec</td>
</tr>
<tr>
<td></td>
<td>falsh001</td>
<td>88 sec</td>
<td>110 sec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>94 sec</td>
<td>455 sec</td>
</tr>
<tr>
<td>10 passwords</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>flash000 ... flash009</td>
<td>301 sec</td>
<td>630 sec</td>
</tr>
<tr>
<td></td>
<td>flash010 ... flash019</td>
<td>258 sec</td>
<td>639 sec</td>
</tr>
<tr>
<td></td>
<td>flash020 ... flash029</td>
<td>523 sec</td>
<td>2106 sec</td>
</tr>
<tr>
<td></td>
<td>flash030 ... flash039</td>
<td>306 sec</td>
<td>1062 sec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>347 sec</td>
<td>1109 sec</td>
</tr>
<tr>
<td>100 passwords</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>flash000 ... flash099</td>
<td>3213 sec</td>
<td>22102 sec</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000 passwords</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>flash000 ... flash999</td>
<td>21819 sec</td>
<td>236092 sec</td>
</tr>
</tbody>
</table>
14 Bibliography


