GPU-based password cracking

Marcus Bakker, Roel van der Jagt

February 5, 2010

University of Amsterdam
System and Network Engineering

Abstract

In this research the following question is answered: what should KPMG advice their clients regarding to password length and complexity, now GPU-based password cracking has become a reality. To be able to answer this question, tests with different tools and hashes were performed on a system with four high end GPUs. The test system showed an improvement of a factor fourteen in brute force speed in comparison with modern CPUs. KPMG's advice to their clients regarding password length and complexity should be one of the following (this applies to all the hashes that were researched):

- Nine or more characters with lower and upper case letters, digits and punctuation marks.
- Ten or more characters with lower and upper case letters and digits.
- Twelve or more characters with lower case letters and digits.

Contents

1 Introduction

KPMG is a world wide company which offers Audit, Tax and Advisory services. As part of the Advisory services, Kleynveld Peat Marwick Goerdeler (KPMG) gives recommendations to their clients about password length. With Graphics Processing Unit (GPU)-based password cracking and the fast development of GPUs on the horizon, this advice may have to change. GPU-based password cracking is several times faster than Central Processing Unit (CPU)-based cracking. Due to this, passwords can be compromised much faster. With this research we hope to make the relation clear between the increasing computation power of GPUs and the need for stronger passwords. KPMG's main question is: "what should we advise our clients regarding password length and complexity, now GPU-based password cracking has become a reality?". This main question is divided into the following sub questions:

- What are the theoretical differences between a CPU and GPU?
  - What are the technical differences?
  - What is CUDA and does how it work?
  - How is this reflected in the various forms of cracking (brute force, dictionary attack., rainbow tables, etc)?
- What tools are available for GPU based cracking?
  - Which relevant hashes are supported?
  - Are these tools actively developed / maintained?
  - Are these tools actively supported?
  - Is distributed GPU cracking supported?
- What is the actual performance gain of GPU versus CPU-based cracking?
  - What is already known about this?
  - What is the performance / price ratio?
  - What is the performance / power consumption ratio?
  - Are GPU based tools really faster compared with CPU tools?
  - What are the performance differences between GPU tools?
- What is the recommended password length for clients of KPMG?
  - Does this varies per hash?
  - How does the future looks like (for example within two years)?

During this research we will try to answers to the questions above. To guarantee a successful final result, we started with theoretical research in section 2 and 3. In these sections we will discuss password strength and expected cracking time (section 2.1), secure storing of passwords (section 2.2), properties of a secure hash function (section 2.3), performance of CPU- and GPU-based craking (section 2.4 and 2.5), and the GPGPU versus CPU (section 2.6). In the next chapter we will discuss the theory behind the hashing algorithms used in this research and their security (section 3). After two sections with theory, we proceed to an overview of the tools we used during this research and their characteristics. Next in section 5 (Test approach) we will discuss the method used for testing the tools. The results from these tests are provided in section 6 (Test results). We conclude this report with a conclusion and discussion.

The appendixes contain the raw test results, the specification of the test system, the script used to generate the random passwords and the commands used to run the password cracking tools.

2 Theory of GPU-based password cracking

In this chapter we will discuss the theory which forms the bases for secure passwords and GPU-based computation. We will start with secure passwords and password strength. We will cover the properties which make passwords strong and how this strength can be measured. Next in section (2.2) we will discuss why hashes should be stored in a secure manner. In section (2.3) we will discuss the properties of secure hash functions. Next in the sections (2.4 and 2.5) we will discuss the performance of traditional CPU-based cracking and the modern GPU-based cracking. In the last section (2.6) we will talk about GPU-based computing. We will cover the basic principles and the differences with traditional CPU-based computations.

2.1 Strong Passwords and expected cracking time

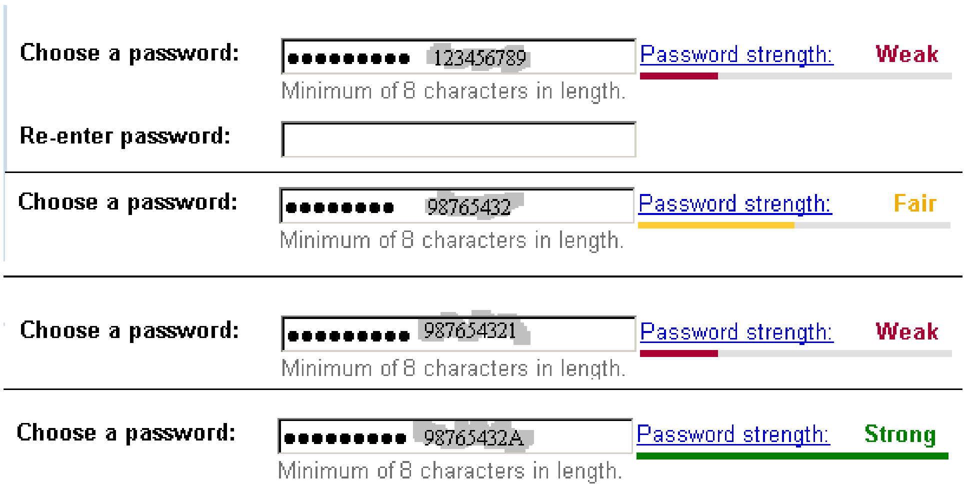
In this section we will discuss the properties which make a password strong, why it is hard to apply strong passwords and we will define a measure of password strength. We conclude this chapter with a overview of expected cracking times for passwords with different properties.

Password strength can be measured in the amount of time needed to guess or brute-force a password. The amount of time needed depends on the complexity of the password. There are several factors that make a password less complex and are therefore more easy to brute-force. These factors are[1, 2]:

- Passwords based on words are vulnerable for dictionary attacks.
- Well known passwords are easy to guess.
- Passwords based on sequences are easy to guess (like 123456789, asdf, qwerty, etc).
- Short passwords are easy to brute-force, because there are less possibilities.
- A password based on only small letters, capital letters or numbers have a small key-space. This makes it more easy to brute-force, just because it limits the possibilities.
- Do not use repeating characters (like *aa11* or *a12ba*). A password with many repeating characters is easy to guess, or someone can easily see the password by watching over the users shoulder [3].

Figure 1 illustrates this in a more graphical way. It shows the rules written above.

Taking the constrains above into account, we can conclude the following: a good password must be hard to guess, has an acceptable length and the characters are chosen from a large key space. Making the password harder to guess can be done by making use of a random sequence of characters. But random passwords have two problems. It is hard to generate real random passwords [5], and for humans its hard to remember random passwords with an acceptable length (humans will remember 7 ± 2 characters [6]). Having the characters in the password being chosen from a large key space can be relative easy achieved. By having at least lower case and upper case letter, a number and a special


Choose a password: ●●●●●●●●● 123456789 Password strength: Weak
Minimum of 8 characters in length.
Re-enter password:

Choose a password: ●●●●●●●● 98765432 Password strength: Fair
Minimum of 8 characters in length.

Choose a password: ●●●●●●●●● 987654321 Password strength: Weak
Minimum of 8 characters in length.

Choose a password: ●●●●●●●●● 98765432A Password strength: Strong
Minimum of 8 characters in length.

Figure 1: The image is part of the Gmail subscription process. It shows some examples of weak and stronger passwords.[4]

character. This simple process contributes to the complexity of the password in a fairly easy way.

The complexity of a password is measured in entropy. The entropy of a password is computed by the length of the password and the used key space. The entropy for a key space can be calculated using the following formula: $\log_2(n)$[5]. With n representing the number of characters in the key space. It provides the following entropies for the most used keys spaces [5]:

Character Pool	**Available Characters (n)**	**Entropy Per Character**
digits	10 (0-9)	3.32 bits
lower case letters	26 (a-z)	4.7 bits
upper case letters and digits	62 (A-Z, a-z,0-9)	5.95 bits
all standard keyboard characters	94	6.55 bits

Table 1: The entropy for often used key spaces[5]. The values given in this table are for one character.

Adding more characters will increase the entropy. Every extra character to a password adds a certain amount of bits to the entropy. For example: an eight character password with all standard keyboard characters. The strength of this password can be computed by multiplying 6.55 bits (the entropy of the used key space) with 8 characters. The result is 52.4 bits of entropy ($6.55 * 8 = 52.4$)[5].

To give an idea of computation times in comparison with the entropy of a password, see figure 2. This graph is based on a computation speed of two billon (2.000.000.000) passwords per second. This number is an assumption based on data available in section 2.5.

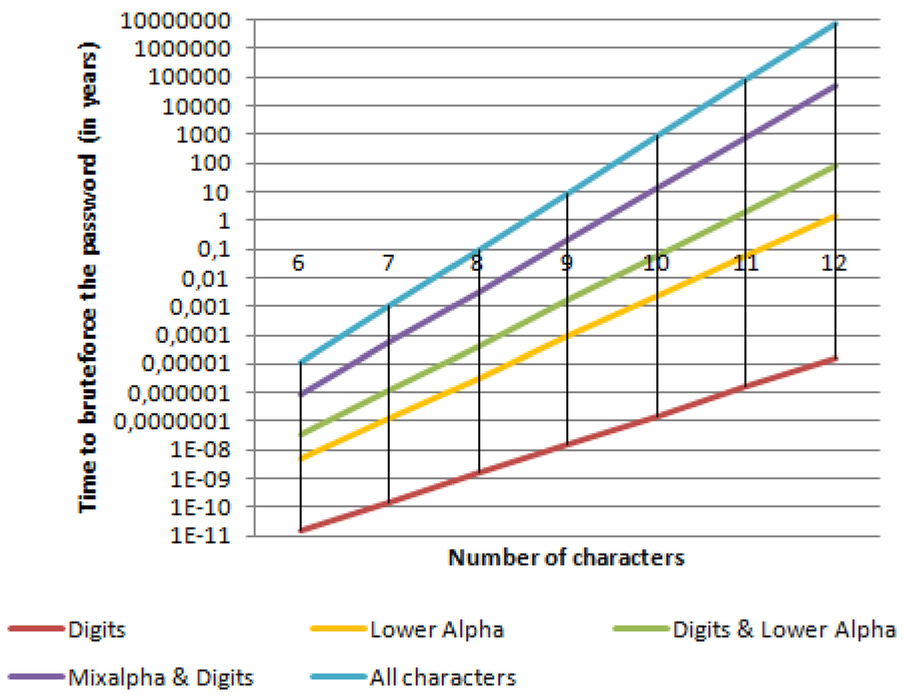


Figure 2: The relation between entropy and computation time for an average cracking system, with a computation speed of two billion password per second. This value is based on data available in section 2.5.

2.2 Secure storing of passwords

In the previous section (section 2.1) we discussed strong passwords and cracking times. Despite having strong passwords, the ability to crack those passwords remains. Due to this, it is very important to store passwords in a secure manner. In this section we will discuss why secure storing of password is important and how this can be achieved.

It is important to note that login credentials are stored in some kind of database. This could be a simple text file or a full fledged database like MySQL. If somehow an adversary captures this database, he may be able to retrieve the passwords. Performing malicious actions with the available credentials could harm a company or an individual person. To prevent this, the passwords of these credentials should not be stored as plaintext. Instead the passwords should first be run through a hashing algorithm (preferable with a random salt) before it is stored.

There is however still a security problem with storing the hash of a password. The fact remains there is no easy way from the hashed password to the plaintext. But, if a weak password is used, it is reasonable easy to recompute the corresponding plaintext password. Finding the plaintext password is a simple process of trying all possible password, until the hashed password is found. The adversary is back in the game.

To prevent the adversary from finding the corresponding plaintext password, we should take an extra security measure to be really secure. Only hashing a password is not enough. Next to hashing the passwords, we should encrypt the database storing the login credentials. Now the odds for the adversary have dramatically decreases. First he has to be able to decrypt the encrypted database with the proper key. If he succeeds in decrypting the database, he still has to crack the hashed passwords.

2.3 Properties of a secure hash function

In the previous section (section 2.2) we disused why and how to store passwords in a secure manner. One level of security is achieved by running the plaintext password through a hash function before it is stored. However, the hash function has to be secure as well. An insecure hash function is of no use if we would like to achieve secure storing of passwords. In this section we will discuss the properties of a hash function and which properties make it secure.

A hash function has at least the following two properties:

- **Fixed length output**: h maps an input x of arbitrary finite bitlength, to an output $h(x)$ of fixed bitlength n[7].
- **Ease of computation**: given function h and an input x, $h(x)$ is easy to compute[7].

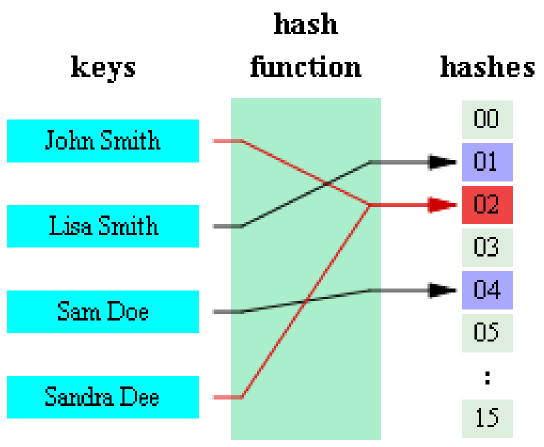


Figure 3: A graphic representation of a hash collision[8]. The figure shows two values pointing to the same hash value.

For a hash function to be secure, it has to meet the following three requirements. The value n is the fixed length of the hash value.

- **collision resistance**: finding a pair $x \neq x' \in \{0,1\}^*$ such that $H(x) = H(x')$ should require $2^{n/2}$ hash computations[9, 7]. This means it is very hard to find two values (which may both be chosen freely) which give the same output, using the same hash function. A graphical example of a collision can be found in figure 3.

- **2nd preimage resistance**: for a given $x \in \{0,1\}^*$, finding a $x' \neq x$ such that $H(x) = H(x')$ expected to require 2^n hash computations[9, 7]. This is almost the same as collision resistance. The difference is that one value is given. The other value may be chosen freely. Finding a collision this way takes 2^n computations.

- **preimage resistance**: for a given $y \in \{0,1\}^n$ finding a $x \in \{0,1\}^*$ such that $H(x) = y$ expected to require 2^n hash computations[9, 7]. This is also known as a one way function. If an output is received, it should be impossible to reverse the process to find the original input.

2.4 Performance of CPU-based password cracking

Despite of having a secure hash function as discussed in chapter 2.3, the possibility remains to crack those hashes using brute-force attacks. The time needed to crack a hash is related to the performance of the used hardware. A hash can be cracked using a CPU or a GPU. We will first discuss the performance of traditional CPU-based password cracking. Due the time constraint we consulted a source on the internet. In section 2.5 we will discuss the performance of GPU-based cracking. Later in this report we will also discuss our own GPU-based password cracking performance data.

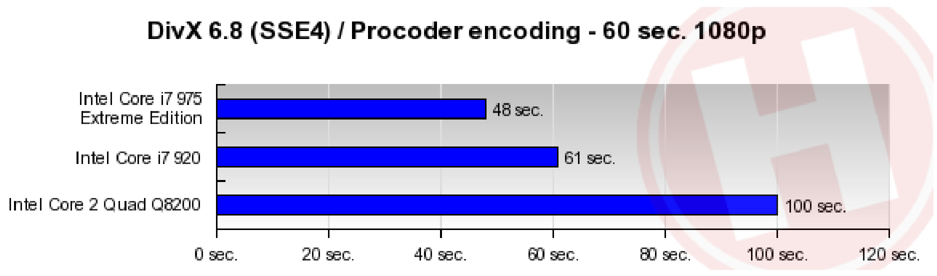


Figure 4: CPU benchmark based on the SSE4 processor feature. This benchmark shows the Procoder Encoding computation time of a 60 seconds DivX video. Because password cracking makes use of SSE2, the ratio between the processors may differ for password cracking. [10]

The data from the consulted source is based on an Intel Core 2 Quad 8200 (figure 5 and table 2). This data is extrapolated using the Streaming SIMD Extensions version 4 (SSE4) benchmark in figure 4. The result of this extrapolation can be found in figure 5 and table 2. The Intel core i7 920 (used in our test system) is 1.639 times faster compared to the Intel Core 2 Quad 8200 and the the Intel core i7 975 EE 2.083 times.

A test performed with the Streaming SIMD Extensions version 2 (SSE2) version of BarsWF bruteforce on our system (which has a Intel core i7 920) gave us a result of 230 million passwords per second on average. According to the data in figure 5 and table 2 it should be around 280 million password per second. This deviation might be the cause of the exploration being based on SSE4 instead of SSE2.

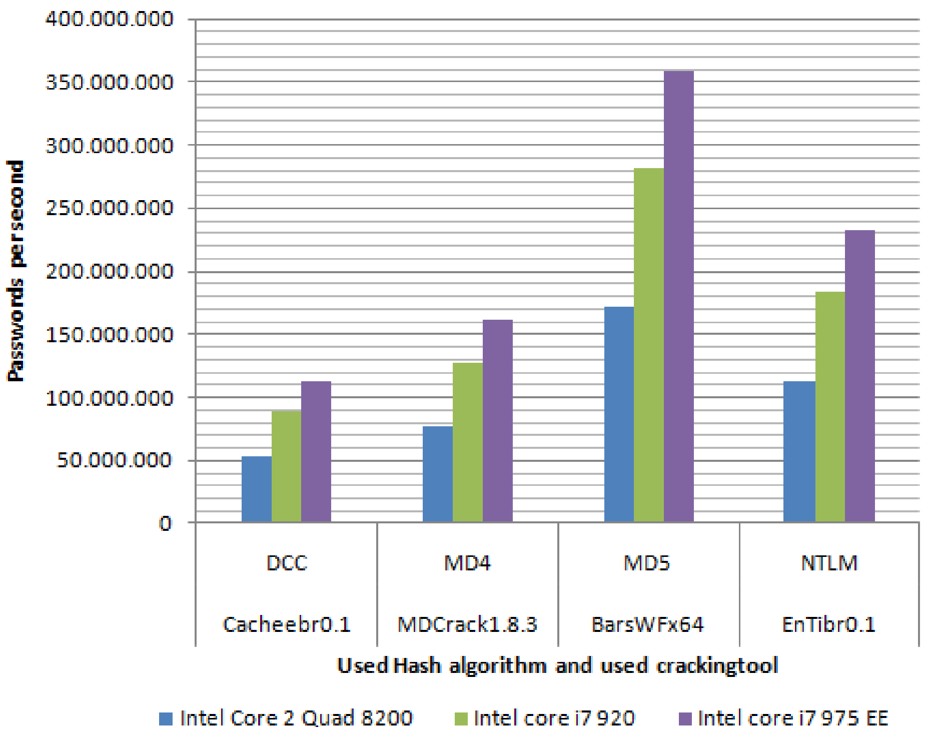


Figure 5: Passwords per second for 4 cores. HT is enabled if available. The data for the Intel Core 2 Quad 8200 are extracted from a previous research done by Nathanael Warren[11]. The other data is computed from these data.

2.5 Performance of GPU-based password cracking

In the previous section (section 2.4) the performance of CPUs was discussed. In this section the topic is the performance of GPUs. The data about the performance, is retrieved from benchmarks distributed by the manufactures of the cracking tools. By making use of this data a first notion on the differences between CPU-based password craking and GPU-based cracking can be made. In figure 6 the data from several benchmarks is collected [12, 13, 14, 15].

Notice in figure 6 the differences in passwords / second between the hashes on the same hardware and tool. Some hashes take more time to compute. This will slow down the process of password cracking. Also another conclusion can be made on the differences between the same hash on different hardware. The hardware developed in the last years has made real big improvements on performance. This performance improvement is also seen in the cracking speed. The Nvidia 8800GS is the oldest card in this overview, and also the slowest one. The Nvidia 9600GT is a next generation graphics card. All other cards are of latest generation and therefore much faster.

Hash	**Tool**	**Intel Core 2 Quad 8200 (p/s)**	**Intel core i7 920 (p/s)**	**Intel core i7 975 EE (p/s)**
Domain Cached Credentials (DCC)	Cacheebr0.1	53.880.000	88.327.869	112.250.000
Message-Digest algorithm 4 (MD4)	MDCrack1.8.3	77.632.944	127.267.121	161.735.300
Message-Digest algorithm 5 (MD5)	BarsWFx64	172.000.000	281.967.213	358.333.333
NT Lan manager (NTLM)	EnTibr0.1	112.000.000	183.606.557	233.333.333

Table 2: Passwords per second for 4 cores. HT is enabled if available. The data for the Intel Core 2 Quad 8200 are extracted from a previous research done by Nathanael Warren[11]. The other data is extrapolate from this data.

2.6 GPGPU versus CPU

Looking at the two previous sections (section 2.4 and 2.5), it can be concluded that GPUs are much faster in password cracking compared to CPUs. In this chapter we will explain how General-Purpose Graphics Processing Unit (GPGPU) makes this difference in performance possible. In addition we will discuss the architectural differences of GPUs and CPUs, followed by the differences in the way they are programmed. Next in this section we will compare performance / price and performance / power consumption ratios. We conclude with an example application of the GPGPU.

Traditional programming is done on a CPU. This kind of programming is called general purpose programming. However there is a "new" class of programming becoming popular. It makes use of the enormous parallel computation power of GPUs. A GPU consists of several of hundreds parallel processing units, running almost one hundred threads each. These large amounts of processing units available in a GPU, delivers huge amounts of computation power. A common CPU has at most four processing cores, with sometimes two threads per processing unit (Intel Hyper Threading). This difference is clearly shown in figure 7 and 8. The purple block are the processing units.

A GPU is specialised in Single Instruction, Multiple Data (SIMD) computations. These kind of operations are essential for computer graphics rendering. CPUs also have the capability to execute SIMD computations, but their specialisation remains Single Instruction, Single Data (SISD) computations.

The high availability of SIMD computations and the large amount of computation power made available through the thousands of threads, makes a GPUs very interesting for password cracking.

The difference in computation power between CPUs and GPUs can be clearly seen in figure 9 (measured in Giga Floating Point Operations Per Second (GFLOPS)). The speed in which the computation power of GPUs grows versus CPUs is also clearly visible. Instead of a doubling in computation power every 18 months, as the Moore's law describes, a GPU increases four times in computation power every 18 months[18, 19]. There are several cause for this enormous growth in computation power:

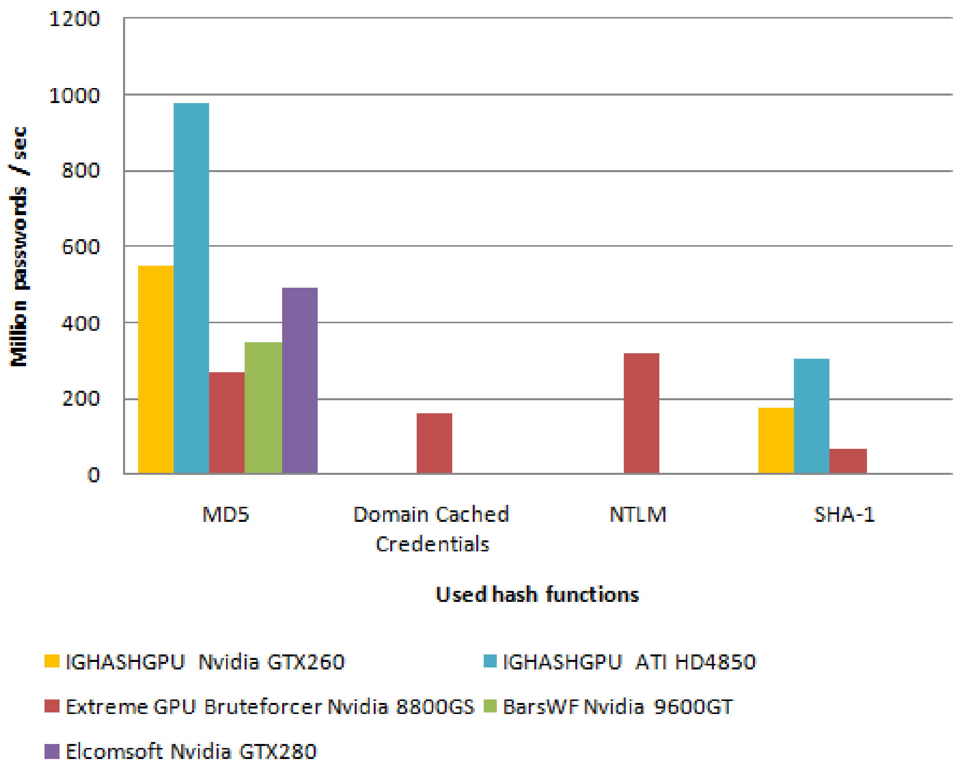


Figure 6: The performance of several tools, each ran on different hardware. Because of the differences in hardware, comparing the tools is not possible. But does gives a good illustration on the performance of GPU-based password cracking[12, 13, 14, 15].

- The way GPUs make use of pipelines and the static operations, the kind of operations in these pipelines requires less cache. This leaves room for adding more computational units [19].
- CPUs have to preserve backward compatibility with older instruction sets. When a new instruction set is introduced, for example Streaming SIMD Extensions version 5 (SSE5), all the previous instructions sets have to remain to work. Thus, they are stacking one generation of instructions on top of the other with every new generation of CPUs. GPUs on the other hand leaves old technology behind. This reduces the instruction set and hence the complexity of the processing units. The result is a much faster processing unit. But it also makes the underling architecture differ between the generation of GPUs. The software developers will however not notice this difference. The changes in the architecture are taking care of by the underlying libraries. It will require the end user to compile the software for every GPU with a different architecture. The user will however not notice this process, he will only see a progress bar [16].

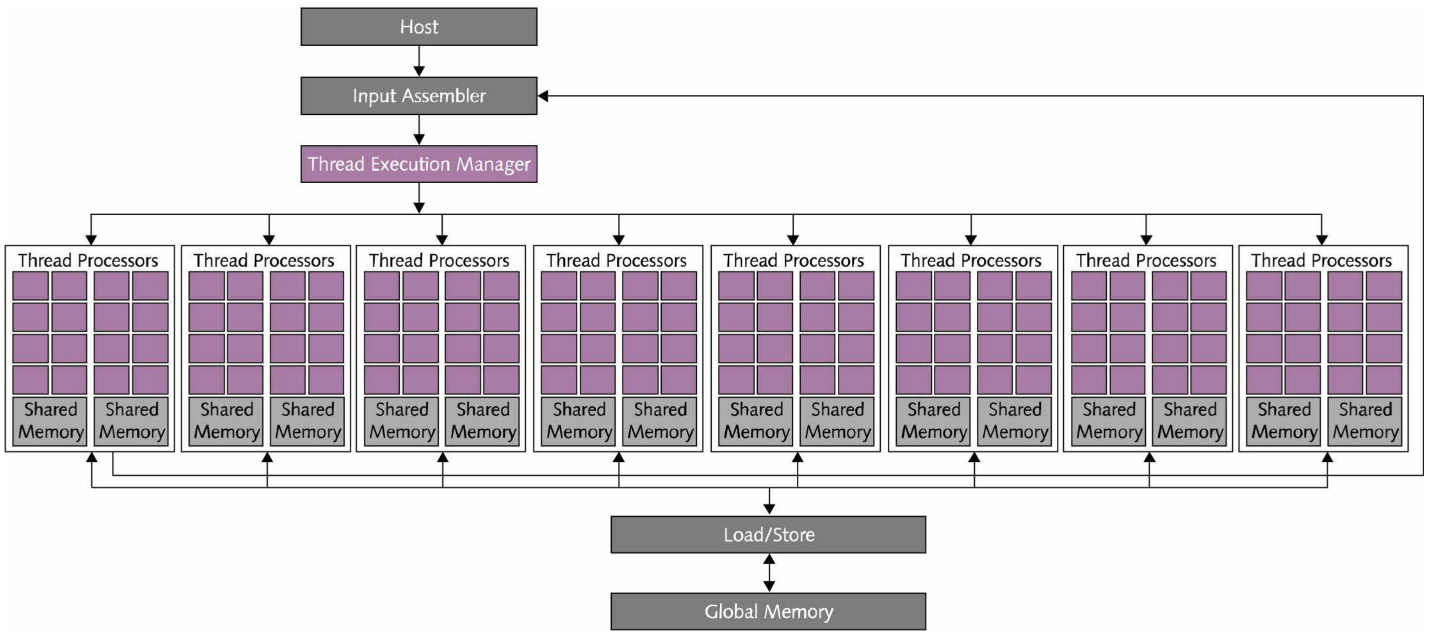


Figure 7: The Nvidia GeForce 8 graphics-processor architecture [16].

GPUs have a lot of computation power and a large amount of GFLOPS. But programming an application for a GPU is totally different compared to traditional programming. It is much harder.

Thanks to companies like Nvidia and Apple we can make use of Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL) to program these GPUs much easier. In short CUDA is a parallel computing architecture, that allows programmers to develop software for GPUs, with a well known standard programming languages, like C and C++. With the CUDA architecture comes a special developed C compiler and libraries.

OpenCL is a framework designed by Nvidia in cooperation with Apple to give developers a standard way to program GPUs. Thus, it is possible to use the same code to write software for Nvidia's CUDA and ATI's Stream architecture. Which are both GPU architectures to provide GPGPU.

Not every algorithm can however benefit from the computation power of GPUs. A very strict requirement is that the algorithm can be cut into many separate chunks. Remember that GPUs are operating in a highly parallel fashion. Another restriction is the limited amount of register and caches available for the threads in GPUs. Making use of the off-chip memory decreases the performance dramatically. To be able to make use of the computation power, the program must compute on the data available in the registers and caches[16].

Some examples of application that are perfect for GPGPU are [19]:

- Linear Algebra;
- Simulation of physical processes;
- Real-time Moving Picture Experts Group (MPEG) video compression;

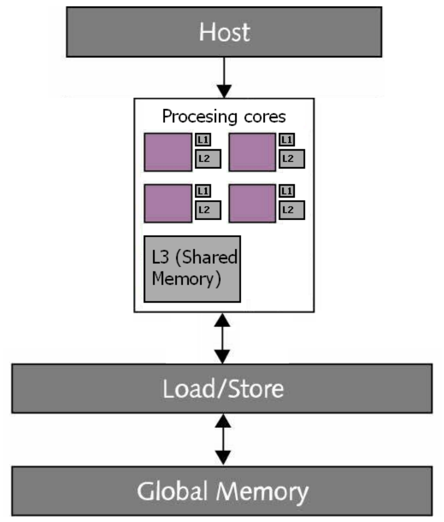


Figure 8: The CPU architecture in general.

- And of course password cracking.

The GPU provides great performance if used in the right way, but how does it compare to the CPU in relation with power consumption and price. In table 3 the performance/price and the performance/power consumption ratios are compared between GPUs and CPUs. Notice the serious benefits of the GPU verus the CPU on all fronts.

	Nvidia GT295	**Intel Core i7 920**	**Intel Core i7 975**
	GPU	**CPU**	
Power consumption[1] (Watt)	289	130	130
Performance (GFLOPS)	1788.48	44,8	55,36
Price[2] (€)	400,-	227,-	825,-
Performance/Price ratio (GLOPS / €)	4,47	0,197	0,0671
Performance/Power consumption ratio (GLOPS / Watt)	6,19	0,345	0,426

Table 3: The performance/price and performance/power consumption ratios for the different processing units. Higher is better. The power consumption is based on the TDP value. The performance is measured in GFLOPS. The reference date for the prices is 26 January 2010 [17, 20, 21, 22, 23].

We conclude this chapter with an example of parallel computation written by T.R. Halfhill in "Parallel Processing With Cuda" [16]

For example, consider an application that has great potential for data parallelism: scanning network packets for malware. Worms,

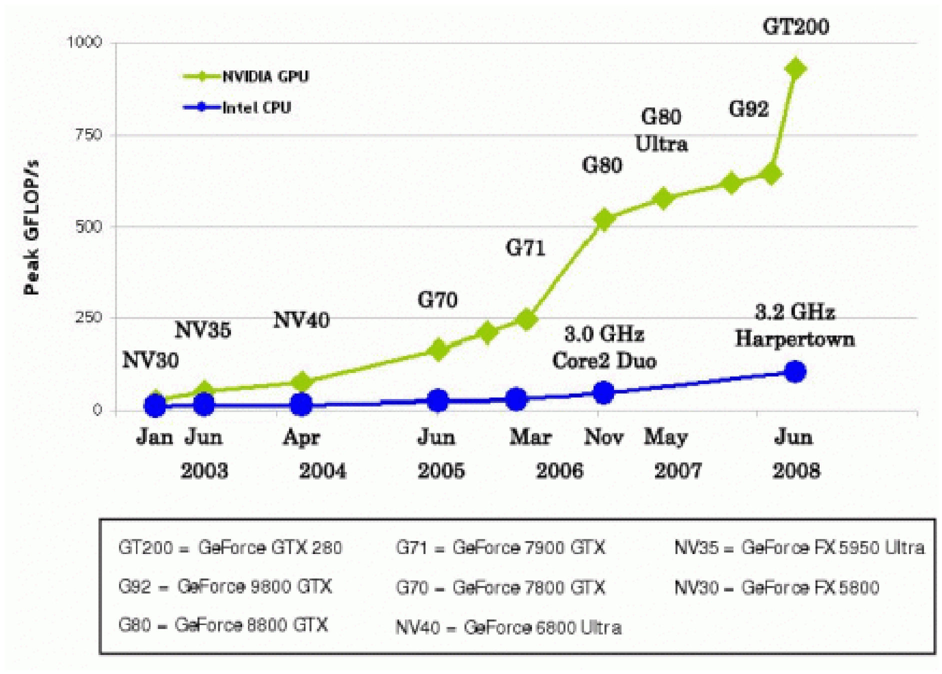


Figure 9: The GFLOPS of a CPU versus a GPU (from Nvidia)[17]

viruses, trojans, root kits, and other malicious programs have telltale binary signatures. An antivirus filter scans the packet for binary sequences matching the bit patterns of known viruses. With a conventional single-threaded CPU, the filter must repeatedly compare elements of one array (the virus signatures) with the elements of another array (the packet). If the contents of either array are too large for the CPUs caches, the program must frequently access off-chip memory, severely impeding performance.

With CUDA, programmers can dedicate a lightweight thread to each virus signature. A block of these threads can run on a cluster of thread processors and operate on the same data (the packet) in shared memory. If much or all the data fits in local memory, the program need not access off-chip memory. If a thread does need to access off-chip memory, the stalled thread enters the inactive queue and yields to another thread. This process continues until all threads in the block have scanned the packet. Meanwhile, other blocks of threads are simultaneously doing the same thing while running on other clusters of thread processors. An antivirus filter that has a database of thousands of virus signatures can use thousands of threads. [16]

2.7 Conclusion

Summarising from the text above we can conclude the following. A strong password is long, makes use of a large key space and is hard to guess. The time it takes to brute-force a password greatly depends on the strength of the password.

A strong password itself is not enough to be truly secure. The password itself has to be stored in a secure manner as well. Secure storing of passwords consist of two levels. Level one is achieved by running the plaintext password through a hash algorithm before it is stored. To add an extra level of security, the hashed passwords should be stored in an encrypted container.

Next to the secure storing of passwords. The hash function that is used to store the password has to be secure as well. A secure hash function meets the following three requirements:

- **collision resistance**: finding a pair $x \neq x' \in \{0,1\}^*$ such that $H(x) = H(x')$ should require $2^{n/2}$ hash computations[9, 7].
- **2nd preimage resistance**: for a given $x \in \{0,1\}^*$, finding a $x' \neq x$ such that $H(x) = H(x')$ expected to require 2^n hash computations[9, 7].
- **preimage resistance**: for a given $y \in \{0,1\}^n$ finding a $x \in \{0,1\}^*$ such that $H(x) = y$ expected to require 2^n hash computations[9, 7].

Taking all these security measures and requirements into account. The possibility remains to crack a hash. This can be achieved by brute-forcing, with CPUs or GPUs. There is a great performance difference of hash cracking with GPUs versus CPUs. GPUs are much faster, consume less power per GFLOPS per Watt and the performance price ratio is much bigger.

The performance difference is the cause of the huge parallel computation power of GPUs. Which gives the capability to run thousand of threads simultaneously. An other cause is the capability to leave old instruction sets behind. Which prevent stacking one generation of instruction on top of the other. This leaves room for more processing units, thus more computation power.

To be able to program a GPUs, GPGPU was introduced. Nvidia has developed the CUDA architecture to provide GPGPU and there is Stream from ATI. However, not every algorithm can benefit from the computation power of GPUs. A very strict requirement is that the algorithm can be cut into many separate chunks. Which is perfectly feasible for hashing algorithms.

3 Hashing algorithms and their security

During this research we brute-forced several types of hashes. In this section we will discuss specific properties of these hashes. NT Lan manager (NTLM) and Domain Cached Credentials (DCC) are based on Message-Digest algorithm 4 (MD4). And the Oracle 11g hash is based on Secure Hashing Algorithm 1 (SHA-1). Most hashes suffer from weaknesses and design mistakes. Which will be discussed for every hash. We will start with the Microsoft hashes (NTLM and DCC). Next we will discuss Message-Digest algorithm 5 (MD5) and MD5 crypt. After this the Oracle 11g hash is discussed. Finally we will discuss two measures to increase the time needed to brute-force the hashes.

3.1 NTLM

In Microsoft Windows, login credentials are saved in the Security Accounts Manager (SAM)(only for local accounts) or in the Active Directory database (for domain accounts). In the past, passwords were stored twice: the LAN Manager compatible password and NTLM password. Both are stored in a different way.

The LAN Manager compatible password can contain up to 14 characters and are case insensitive. Furthermore, the password is based on the Original Equipment Manufacturer (OEM) character set. The password is split up into two parts of seven characters, each part is stored in eight bytes. This gives a total hash length of 16 bytes. It should be noted that it is not a real hash. The seven character password is used as a key to encrypt a static string (*KGS!@#$%* [24]) using the DES cipher.

The LAN Manager compatible password is relative easy to crack because of the little entropy that is available. When cracked, it is relative easy to compute which characters of the password are lower and upper case. In the past a LAN Manager password was always stored besides the NTLM password (as long as the password contained 14 characters or less). This functionality can be disabled. In the modern versions of Windows (Windows 7 and Windows 2008) this function is disabled by default.

The NTLM password is based on the Unicode-16 character set, it is case sensitive and can be up to 128 characters long. The password is computed by using the MD4 algorithm. The MD4 algorithm computes a 16-byte digest from a variable-length plaintext password[25].

Besides the weakness introduced by storing the NTLM and corresponding Lan manager (LM) hash, NTLM has an other serious weakness. The MD4 hashing algorithm used by NTLM, is known to have some serious flaws. Den Boer and Bosselaers already showed this in 1991 (one year after its introduction)[26]. In the same year MD5 was introduced by R. Rivest to replace MD4. Other researcher have later revealed further weaknesses of MD4[27].

3.2 Microsoft Domain Cached Credentials

When a user logs in on a Microsoft domain using a Microsoft Windows computer, its credentials are cached on its local machine. The Domain Cached Credentials (DCC). This is done for the reason to accelerate the login procedure, and to be capable to logon on to the computer even if the domain controller can not be reached. For example when the computer is disconnected from the corporate network, or when the domain controller has crashed[28]. Microsoft Windows does not save the plaintext credentials, but it keeps a salted hash. The hash is computed using the following function:

- hash = MD4 (MD4(password in Unicode) + lowercase(username in Unicode))[29, 30]

At the website Passcape[3] we found that all Windows versions since Vista make use of a new algorithm, but we could not verify this:

- hash = PBKDF2_SHA(MD4 (MD4(user password)+lowercase(user name)), iterations)[29]

By default, the iterations value is equal to 10240. Please note that the number of SHA passes will be several times greater than this value [29].

Compared to the NTLM hash, the DCC hash is more secure. As result of the salted hash and the extra MD4 round. But just like NTLM, DCC suffers from serious flaws of MD4.

3.3 MD5

A very popular hashing algorithm is MD5. It is also an internet standard [31]. It is not like NTLM or Microsoft DCC used for one particular application but in a wide variety of applications.

The message digest computed with MD5 has a length of 128 bits. This seems to be very secure, but researchers (Sotirov, Stevens, Appelbaum, Lenstra, Molnar, Osvik and de Weger) have shown serious flaws in MD5 [32]. It allowed them to create a valid rouge CA certificate. Despite these known flaws, MD5 remains a very popular hashing algorithm. Its popularity makes it an interesting hash to crack.

3.4 MD5 crypt

MD5 crypt is a hashing algorithm used in Unix to store user passwords in the shadow password database. It was developed by Poul-Henning Kamp to store passwords in FreeBSD. It should be noted that MD5 crypt is not really a hashing algorithm, but it makes use of MD5.

MD5 crypt makes use of a salt and a number of MD5 iterations[33]:

> *The MD5-based crypt() scheme uses the whole passphrase, a salt which can in principle be an arbitrary byte string, and the MD5 message digest algorithm. First the passphrase and salt are hashed together, yielding an MD5 message digest. Then a new digest is constructed, hashing together the passphrase, the salt, and the first digest, all in a rather complex form. Then this digest is passed through a thousand iterations of a function which rehashes it together with the passphrase and salt in a manner that varies between rounds. The output of the last of these rounds is the resulting passphrase hash.* [33]

[3] www.passcape.com

The final output starts with the version identifier '1', the salt, a '$' separator, and the 128-bit hash output. All encoded as a base64 ASCII string. In most Unix and Unix like operating systems this output (and the corresponding user) is stored inside the file */etc/shadow*.

MD5 crypt is more secure compared to plain MD5. It adds different kinds of salting and a thousand MD5 iterations. The MD5 iterations slows down the cracking speed by a factor thousand. For example, a cracking speed of three billion per second for plain MD5 drops down to thirty million (which is slow).

3.5 Oracle 11g (salted SHA-1)

Oracle makes use of a new hashing algorithm to store passwords with their database software Oracle Database 11g. Two researchers at The Security Lablog [34] and P. Finnigan's [35] discovered what kind of hashing is applied. It is a salted SHA-1.

SHA-1 is the successor of MD5. It was published by the National Institute of Standards and Technology (NIST) in 1995. The message digest has a length of 160 bits.

The Oracle 11g hash is computed by taking the plaintext password concatenated with a random salt of length 10 in binary format and run it through SHA-1. The result is the Oracle 11g hash.

The Oracle 11g credentials and thus also the hashes are stored inside the *sys.user* table at the field *spare4* as a HEX value preceded by 'S:'. The message digest itself is a HEX value of length 40. Concatenated to that value is the salt. Which is a HEX value of length 20.

A salted SHA-1 hash is stronger as a non salted SHA-1 hash. But besides the salting of an Oracle 11g hash, SHA-1 suffers just like MD4 and MD5, as discussed earlier, from weaknesses. Three researchers (Wang, Yin and Yu) showed the first serious flaw of SHA-1 at CRYPTO 2005 [36]. In the security community is said SHA-1 is broken. Luckily there is already the successor of SHA-1, SHA-2. And even the successor of SHA-2 is coming in 2012. Despite the availability of SHA-2 at the time Oracle 11g was published, they did not made the wise decision to use the more secure SHA-2 instead of the less secure SHA-1 hashing algorithm.

There is however another weakness in the way passwords are stored in Oracle Database 11g. They made the same mistake as Microsoft did with their NTLM hash, by storing the LM hash next to the NTLM hash. Besides the Relative secure Oracle 11g hash, also the far less secure Data encryption standard (DES) encrypted password is present in the *sys.user* table next to the new Oracle 11g hash.

3.6 Protect against brute-forcing

All the hashes discussed above can be brute-forced. Often the amount of computation needed to crack, can be shortened by making use of the weaknesses in the hashes. In this chapter we will discuss how salted hashes or more complex algorithm based on known hashing algorithms can be used to slow down brute-forcing process.

How does a salt prevent a hash from being easily brute-forced? The salt was originally introduced to prevent against rainbow table attacks. But this also seems to prevents against brute-force attacks. During a brute-force attack often a list of hashes is cracked. Passwords are generated, hashed and compared with all the hashes in the list. It allows for parallel brute-forcing, every hash has to be generated only ones and can be check against the whole list of hashes. But what if you apply a salt? This will break the possibility to use parallel brute-forcing. Because every hash in the list has a different salt. The salt will result in different hashes for the same passwords.

Another possibility to prevent hashes against brute-force attacks, is applying a complex algorithm based on an already existing hashing algorithm. MD5 crypt is a nice example of this (as discussed in section 3.4). The MD5 hash can be brute-forced in an acceptable amount of time. But by applying the MD5 hash a thousand times, the times it takes to crack the hash also increases with a factor thousand.

So even if the hash can be brute-forced, applying a salt or a more complex algorithm based on a known hashing algorithm, the hashes are still usable for secure storing passwords. Although, salting only prevents parallel brute-forcing. I does not protect the ability to crack a single hash a time.

3.7 Conclusion

As discussed above, it is very clear that none of the hashing algorithms mentioned are bullet proof. They all suffer from some kind of weakness. A weakness in the hashing algorithm, or by making the mistake to store an older less secure version of the hashed password next to the new one.

These problems make the hashes vulnerable to attacks. The computation time needed to crack a password stored in the hash is shortened. But by adding a salt or by using a more complex algorithm around the hash, the time to brute-force can be increased.

4 GPU-based cracking tools

In this section we discuss the tools used during our tests. An overview of these tools can be found in the first section (4.1). In the final section (4.2) we will discuss the support for distributed cracking.

4.1 Cracking tools overview

During this research several tools were selected a tested. The tools were selected on the following characteristics:

- GPU support;
- Support, maintenance and active development;
- Supported hashes.

Based on those characteristics the tools in table 4 were selected. Some interesting thing can be said about this table among other:

- Most tools do not support GPU-based password cracking, including *Cain and Abel*[4] and *John the Ripper*[5].
- The tools GPU md5 Crack and Multihash CUDA Brute Forcer do not support multiple GPUs. Having a test system with multiple GPUs made them less interesting for our research.
- Some tools were very immature, like Distributed Hash Cracker. They did not ran out of the box. We skipped the Linux tools, because of the time we needed to fix these problems and the high availability of Windows tools.

4.2 Support for distributed cracking

Part of the research was distributed GPU-based hash cracking. It is supported by two tools:

- Distributed Hash Cracker;
- Elcomsoft (only the commercial version has support for distributed cracking).

The Distributed Hash Cracker has an excellent paper written by A. Zonenberg [37], explaining the inner workings of there distributed hash cracker.

The Distributed Hash Cracker has support for GPUs and CPUs to crack hashes. The basic principle is fairly simple. There is one master server waiting for jobs to arrive. A jobs consists of a specific hash value to crack. If a jobs has arrived it starts handing out work units to compute nodes. A work unit consists of a specific character range to crack (for example aaaaa-zzzzz) and the hash value that has to be cracked. The compute node will return a success or failure message, depending on the result of the crack attempt. On a failure message, a new work unit is handed out to the compute node. On a success message the

[4]http://www.oxid.it/cain.html
[5]http://www.openwall.com/john/

Name	URL	Hashes	Re-sume [a]	Table [b]	OS	License	Active developed	Support
Elcomsoft	www.elcomsoft.com	MD5, NTLM, DCC	X	-	WIN32	MD5 is free. Complete package is commercial.	Active	Yes
GPU md5 Crack	bvernoux.free.fr/md5/index.php	MD5	-	-	LIN32 WIN32	LGPL v3	Latest update: 09-15-09. **1***	Unknown **2***
Multihash CUDA Brute Forcer	cryptohaze.com/bruteforcers.php	MD5, NTLM	-	-	LIN32/64 WIN32	Unknown (source will become available)	**1***	Forum
Extreme GPU Bruteforcer	www.insidepro.com/eng/egb.shtml	MD5, NTLM, DCC	X	-	WIN32	Shareware	Active	Yes
Distributed Hash Cracker	rpisec.net/projects/show/hash-cracker	MD5, NTLM	X	-	LIN32/64	BSD	Latest update: 10-23-09, **1***	Forum
IGHASHGPU	golubev.com/hashgpu.htm	MD5, NTLM, DCC, ORACLE 11g	-	-	WIN32	Free for non commercial use	Latest update: 01-09-10, **1***	**2***
BarsWF bruteforce	3.14.by/en/md5	MD5	X	-	WIN32/64	Unknown	Latest update: 01-10-09, **1***	Forum
RainbowCrack	project-rainbowcrack.com/index.htm	MD5, NTLM, ORACLE 11g	?	X	WIN32	Source available	Unknown	**2***

1* New features are planned

2* There's only an email address available on the website

Table 4: An overview of tools reviewed and tested during this research.

[a]The tool has resume support at the point where the crack session left of.

[b]The tool makes us of a rainbow table to crack the hash.

master server will stop handing out work units to the compute nodes. It returns to the idle state, waiting for new jobs to arrive.

The big advantage of dividing the load by means of work units, is scalability and simplicity. Another strategy could be a parallel algorithm, where each compute node will compute a small part of the whole computation. This approach has several problems:

- All compute nodes should be synchronised.
- It requires a good mechanism to recover from failures, when one or more nodes suffers from an error.
- What if the distributed cracking system is heterogeneous? Some nodes will be finished much faster compared to other nodes. The distributed system scales down to the slowest node if there are no mechanism to take care of this behaviour.

There are probable many more problems with a parallel algorithm. But most important, the final system will be very complex and difficult to understand. Making use of work unit has a big favour in reducing complexity and taking care of scalability problems.

4.3 Conclusion

We used the following tools for our test: BarsWF bruteforce, IGHASHGPU, Extreme GPU Bruteforcer and Elcomsoft. The other available tools did not have GPU support or did not run on Microsoft Windows. Microsoft Windows was our main choice of OS, because of the high availability of tools.

Two of the tools also support distributed cracking. For one of these, this feature is available in the commercial version (Elcomsoft) and the other is free available (Distributed Hash Cracker).

5 Test approach

In this chapter we will discuss how we have tested the cracking tools. We will start with a explanation on the test method. In the next section (section 5.2) follows an overview of the password set used in the test.

5.1 Test method

The test performed had to met the following requirements:

- The hashes we had to crack were: MD5, MD5 crypt, DCC, NTLM, Oracle 11g.
- The passwords had to be of length: 6, 8, 10 and optional 12.
- There had to be five different categories of passwords:
  - 0-9;
  - a-b;
  - a-b, 0-9;
  - a-b, A-B, 0-9;
  - a-b, A-B, 0-9 and punctuation marks.
- There had to be five passwords for every different password length and category.
- Different tools had to be used to crack the hashes.

With the above requirements in mind we defined the following test method:

- We generated a total of 100 random passwords in 20 different categories. We chose to make use of random passwords in different categories to exclude any influences on the test results. Some tools may for example start with cracking from *z* and some tools may start from *a*.
- A password category consists of a specific character set (for example a-b, 0-9) and password length (6, 8, 10 and 12).
- The corresponding hashes (MD5, MD5 crypt, DCC, NTLM, Oracle 11g) for every password were generated.
- Different tools were used to crack the hashes: BarsWF bruteforce, IGHASHGPU, Extreme GPU Bruteforcer and Elcomsoft. The time it took to crack a hash was measured.
- When the results where collected, we started analysing them, to answer the main research question.

5.2 Password set

In this section the set of passwords used during the tests is given. We wrote a Python script (appendix D) to generate the passwords and hashes to be shore they were random.

Key space/length	6	8	10	12
0-9	789740 455278 651773 109642 511881	85650786 63099104 74567940 46000701 54223502	0119425077 6433423789 2762342329 2760161644 0124732837	371787564926 685225732402 763251588665 695128406731 590747117552
a-b	bnlwoh fxlomv pncsol kqmenf wlwuuo	eihptwjw zbppirmj uyxdebyu ruakcqjm symrtqym	sbbebgjxry kpqfsfzziz mylpkghyuz pqcfmfdknc hlxrpydpzp	fxjoctruxlze zpejprmignwf gyocxxizzzzl pzcexkphgyys kffrvaqsxrgb
a-b, 0-9	ylfw21 mqy1ll 5czznz 82q31w 16tfdj	gprjepsw 7nasv4k4 9tihpdae ws9n2vfg bzd9kulm	nu08oowe90 Yqq6mycDOq 427MSE58Ba MA12sKeHs6 hrVwaypVTr	r4t9vwpyuymt qbg3s9tdei75 0yrfnc2ygqo7 5c3iiptl6ugc i695fuzahh5z
a-b, A-B, 0-9	CqBAQh wTTCiG IyNSTP D6NR5W vPlS1x	nfwC50P2 Pz2aNRLT jfEeUyG8 jYeffeHD QT0Z0cr0	oVzjq2aCb8 Yqq6mycDOq 427MSE58Ba MA12sKeHs6 hrVwaypVTr	aFqGGQothBdR pKIprkGZDBMb 8OFX2bGC7d2f AMANEesDJoPK xgI5ZkaTvYDN
a-b, A-B, 0-9, punctuation marks	⟩,%w+7 F)EVZm 3⟨DW'8 de∼'1t $∼2D!{	HD'I5fP$ 'hxgFM,4 uE⟨An_(y 6Nˆl[1@w %EZ5?E+∗	⟩GM-6⟨:zdr Z0U}G%I6aC }I6"1(⟨r[t cUfWy%⟩97) gIjtgQP'∼{	+B)V,{.!yD4J MfJ'.!RDUDpS A:EBl@C?bUwX]N}lto")\—)r 9:z2{Gl'rAr4

5.3 Conclusion

The test made use of 100 different passwords to crack. Consisting of different lengths and character sets. For every password we generated five different hashes: MD5, MD5 crypt, DCC, NTLM and Oracle 11g. This provides a total of 500 different hashes, available for cracking. Furthermore, we made use of four different tools to crack those hashes: BarsWF bruteforce, IGHASHGPU, Extreme GPU Bruteforcer and Elcomsoft. The time it took to crack one of these hashes was measured for every tool. When the results where collected, we started analysing them, to answer the main research question.

6 Test results

Following the test approach of the previous chapter (chapter 5), the results of the tests are presented in this section. In section 6.1 we will present the results from appendix B (raw test results) by means of graphs. In the next section (section 6.2) we conclude with an overview of processed passwords per second.

6.1 Performance - Cracking time of a password

The graphs below are based on the test data available in appendix B. Every bar in the graph presents the average from the available data from the five tests to limit the effects on the results. Every character set and tool is presented as a different bar.

All the graphs (figure 10, 11, 12 and 13) show an increasing computation time, when the password complexity and length is increased. The most complex and longest password were not tested, because of the limited amount of time available for the research. Some hashes may take up to a weak to crack.

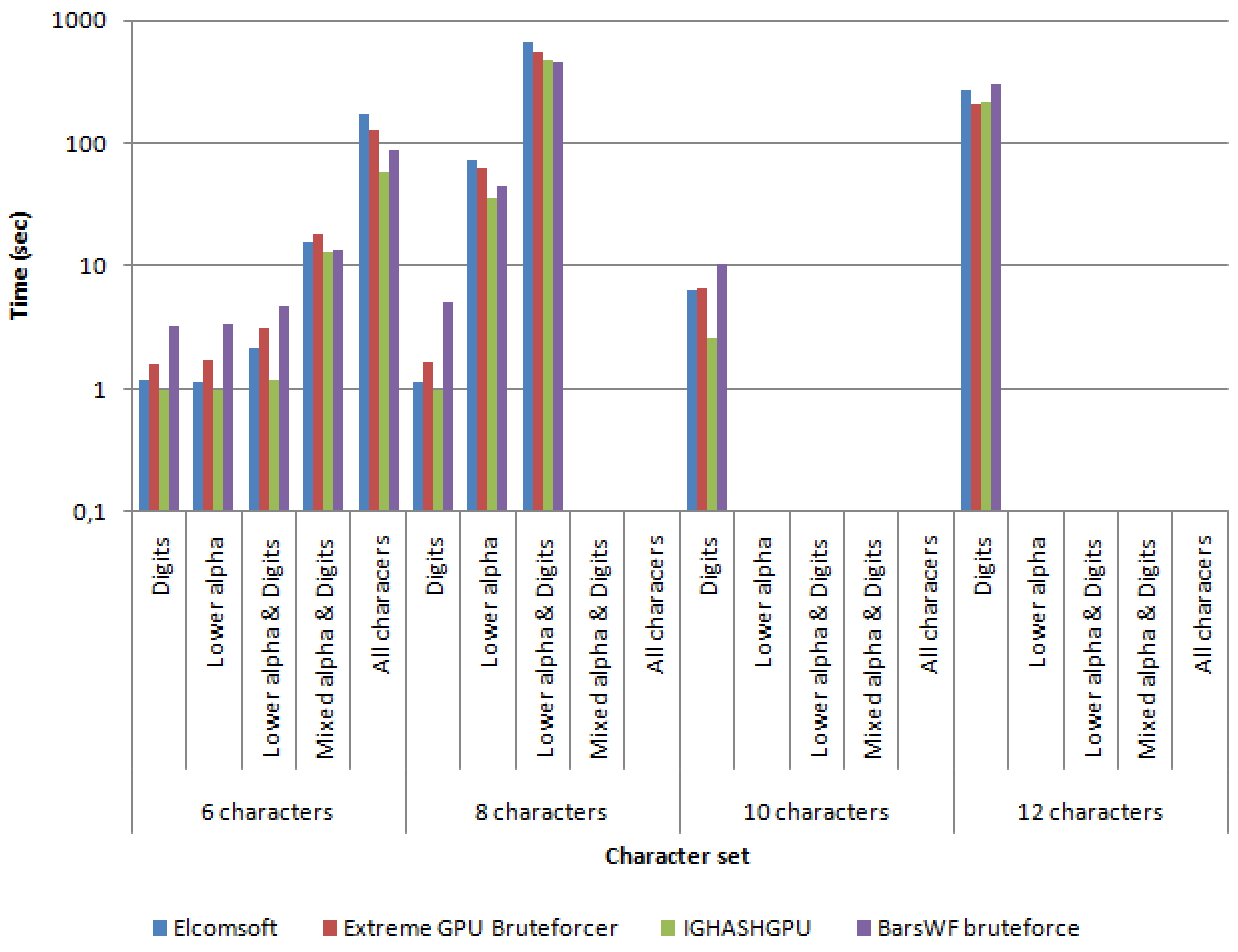


Figure 10: The time it takes to crack MD5 hashes with different password lengths and character sets. The coloured bars represent the different tools used to crack the hash.

The first graph (figure 10) presents the time needed to crack a MD5 hash. The time measurement by BarsWF bruteforce may have an error of zero till two seconds (BarsWF bruteforce requires a key interrupt before it exits. On the background we ran a Python script giving every two second a key interrupt. The time was measured from the moment BarsWF bruteforce exits). When this time is taken into account, there is no significant difference in time between the tools. The small differences in time, are caused by the algorithm used to search through the key space.

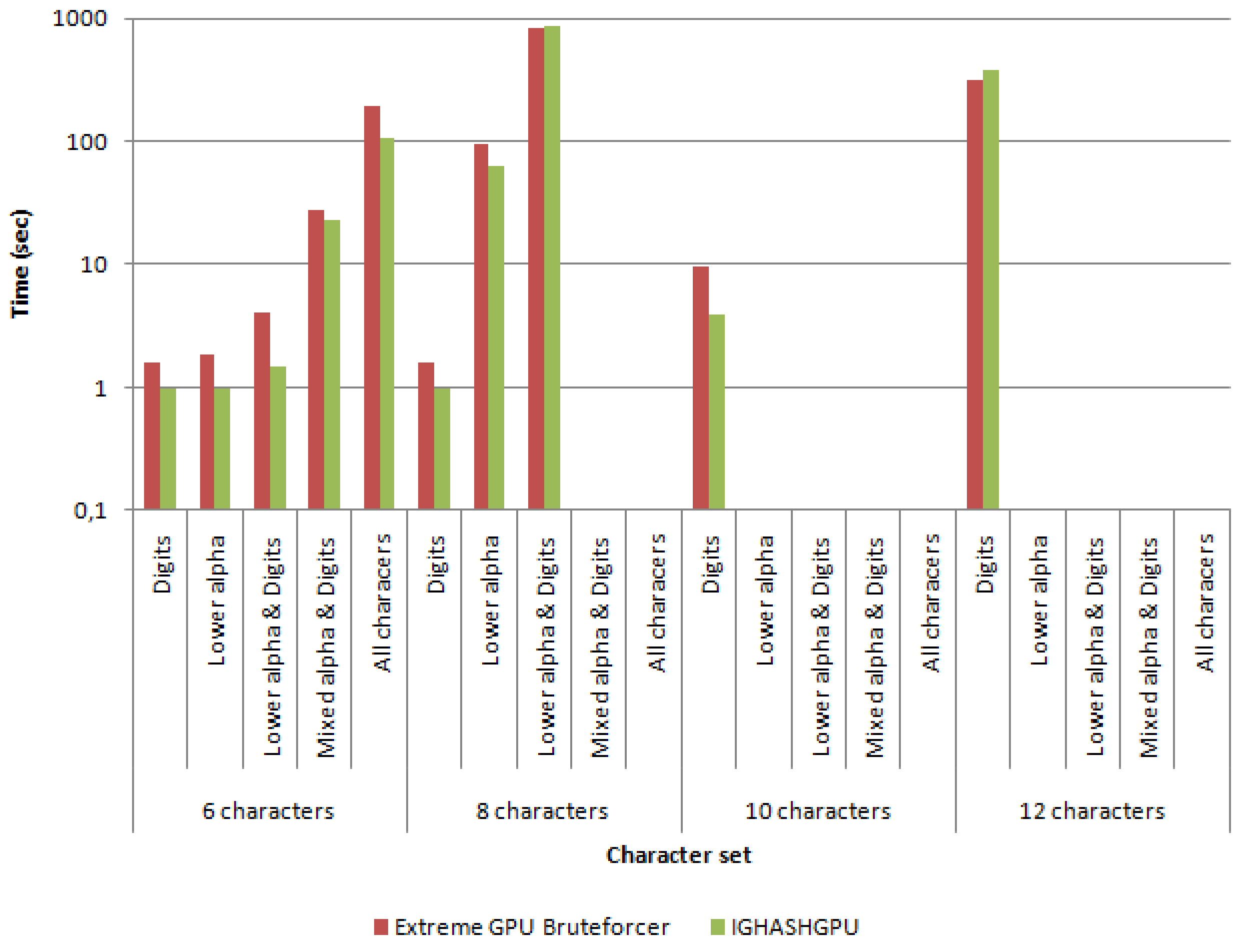


Figure 11: The time it takes to crack Domain Cached Credentials with different password lengths and character sets. The coloured bars represent the different tools used to crack the hash.

The second and the third graph (figure 11 and figure 12) are very similar. In general IGHASHGPU finds the password a little bit faster as Extreme GPU Bruteforcer does, but the differences are very small. Extreme GPU Bruteforcer is only faster in the following two cases:

- 8 characters - Lower alpha and digits;
- 12 characters - Digits.

The DCC hash takes more time to crack compared to the MD5 and NTLM hashes. The difference in cracking time is caused by the complexity of the hash function, as also mentioned in section 2.5.

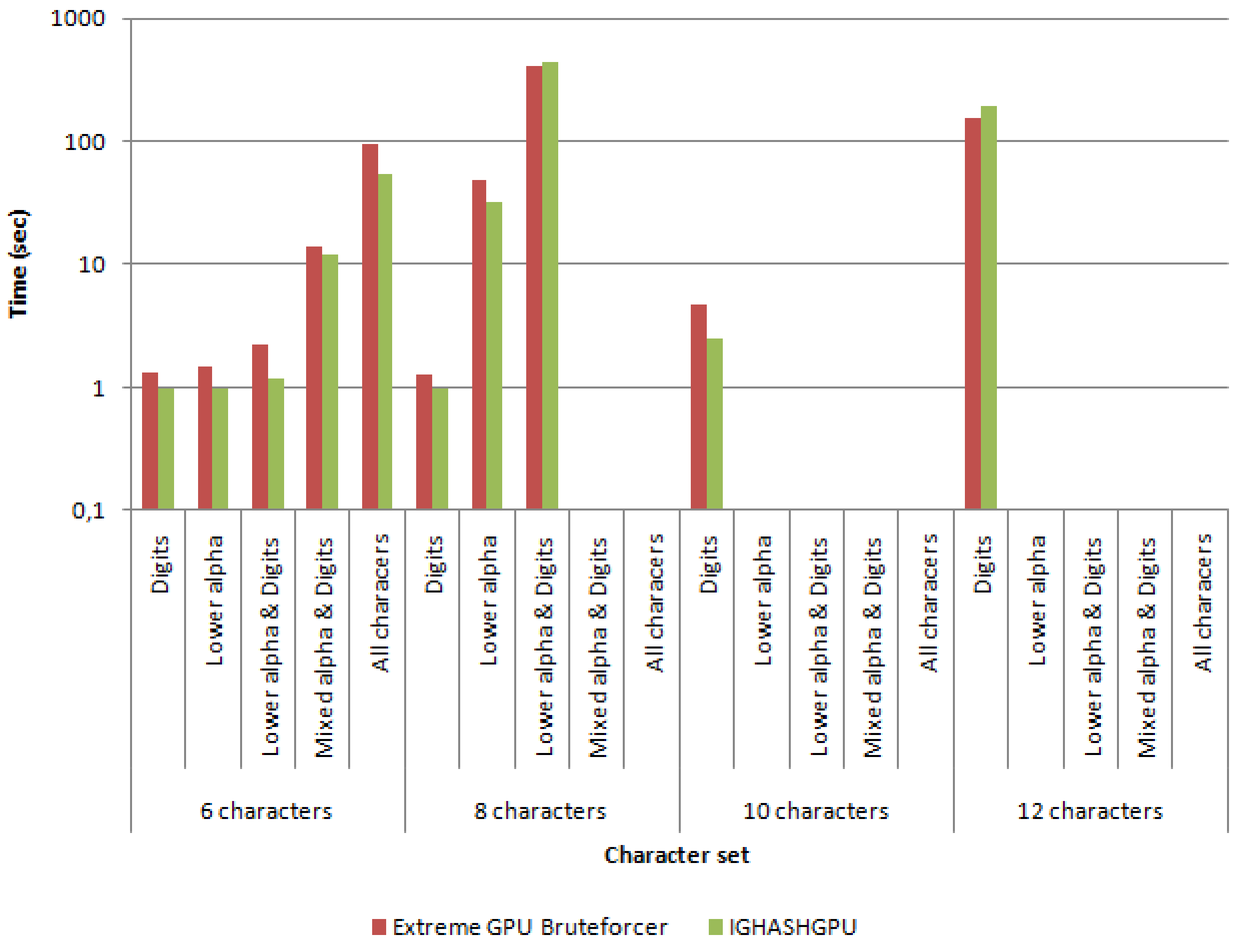


Figure 12: The time it takes to crack NTLM hashes with different password lengths and character sets. The coloured bars represent the different tools used to crack the hash.

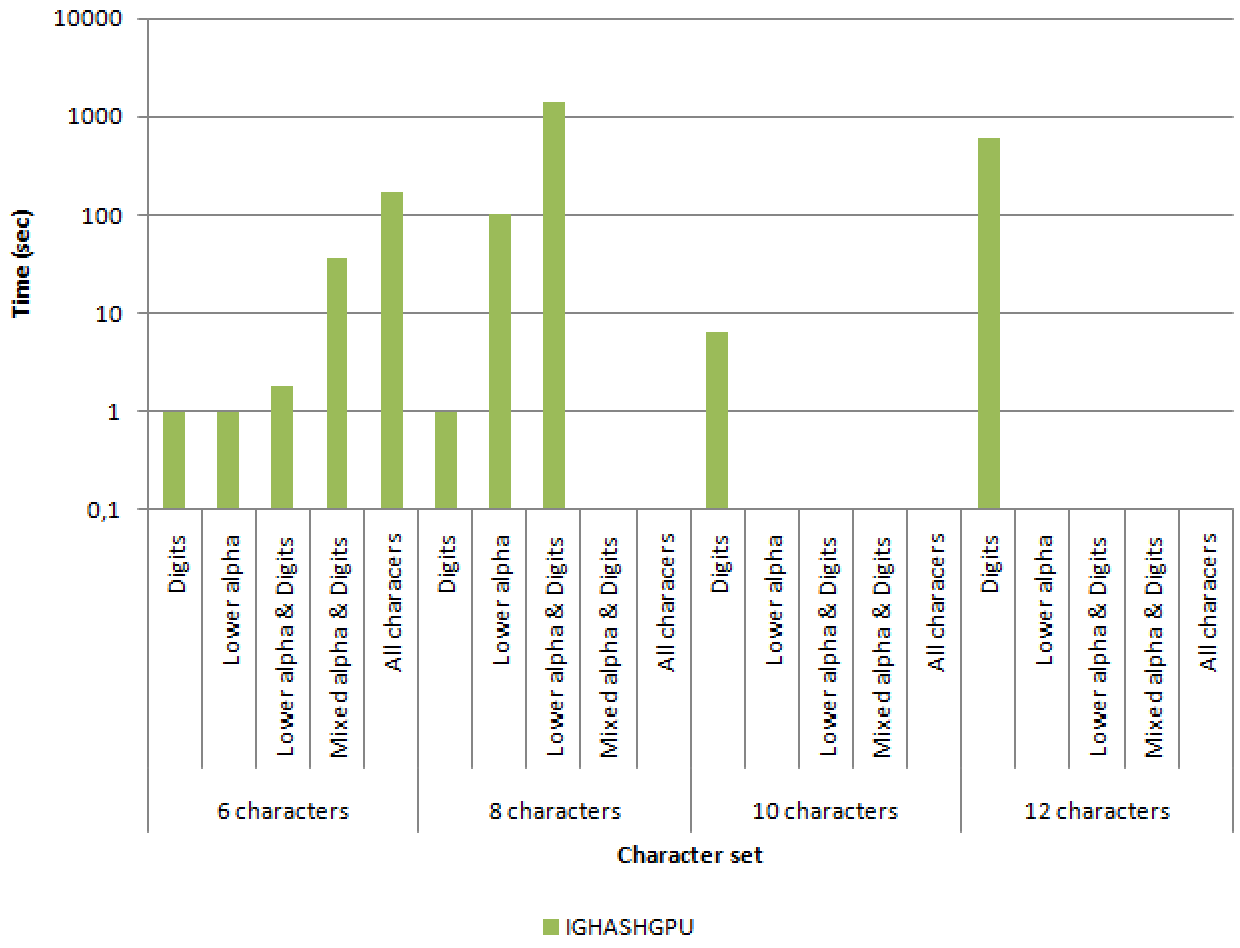


Figure 13: The time it takes to crack Oracle 11g hashes with different password lengths and character sets. The coloured bars represent the different tools used to crack the hash.

The last figure (figure 13) presents the time needed to crack an Oracle 11g hash. The characteristic of the graph are comparable with the graphs before. There is however one important difference. Cracking an Oracle 11g hash takes more time in comparison with the other hashes.

6.2 Performance - Processed passwords per second

In the previous section (6.1) we discussed the cracking performance measured in the time takes to crack a hash. Another measurement for the performance can be measured in the amount of passwords processed per second. An overview on the amount of passwords per second can be found in table 5 and figure 15.

Hash	**Tool**	**Performance** (Million password / sec)
MD5	Elcomsoft	1750
	Extreme GPU Bruteforcer	2260
	IGHASHGPU	2770
	BarsWF bruteforce	3200
NTLM	Extreme GPU Bruteforcer	3050
	IGHASHGPU	3050
DCC	Extreme GPU Bruteforcer	1556
	IGHASHGPU	1565
Oracle 11g	IGHASHGPU	968

Table 5: Average processed password / second for every hash and tool. Based on these values it is possible to compute the expected time it takes to process all passwords of a certain key space. A graphical representation can be found in figure 15.

The amount of passwords processed for a MD5 hash highly differs for every tool. Probably not every tool is equally efficient in computing the MD5 hash. The performance difference of the NTLM hash is much more stable, both tools process exactly the same amount of passwords per second. The amount of passwords per second for DCC is almost the same as for both these tools.

NTLM and MD5 perform almost equally. Compared to the DCC and Oracle 11g hash they both process less passwords per second. This is the cause of the hashing algorithm complexity, as already mentioned in the previous sections (section 6.1 and section 2.5).

6.3 Conclusion

The time it takes to crack a hash increases, when the password complexity and length is increased. MD5 and NTLM perform almost equally if it comes to the amount of password cracking attempts per second. Next follows DCC to be followed by Oracle 11g as the slowest hash to crack. The differences in cracking time between the cracking tools is to small to be of any significant meaning.

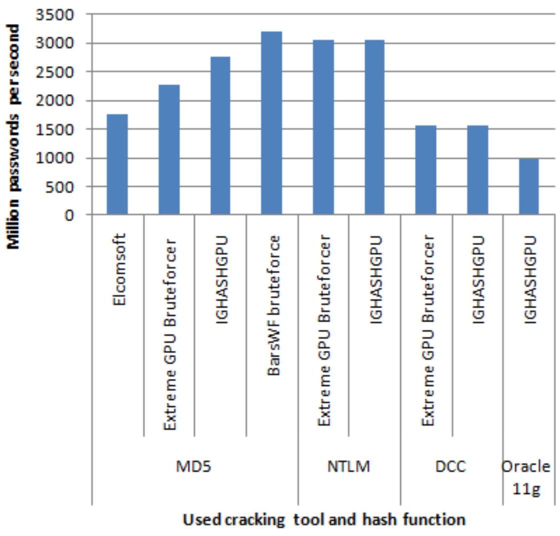


Figure 14: Processed password / second for two Nvidia GTX295. Based on these values it is possible to compute the expected time to process all passwords of a certain key space. The exact values can be found in table 5.

7 Conclusion

In this chapter we will answer the research question mentioned in the introduction (chapter 1) and draw some conclusions.

7.1 What are the theoretical differences between a CPU and GPU?

There are some essential differences between the architecture of CPUs and GPUs. CPUs are generic processing units. CPUs have a large instruction set and instruction are executed in series. They are specialised in SISD. GPUs are more specialised processing units, with a much smaller instruction set and with a very limited usage of memory. GPUs are executing instruction in parallel (they contain a few hundreds of parallel GPU cores) and are specialised in SIMD.

Password brute-forcing is based on SIMD, and can be computed in parallel, with a limited memory usage. This makes GPUs very useful for hash brute-forcing. Because of the limited amount of time available for this research, GPU-based dictionary attacks and rainbow tables were not studied.

To be able to use the GPU for general purpose, programmers made use of the graphics libraries available for the graphics cards. But the possibilities were limited and developing programs was very difficult. This changed with the introduction of CUDA by Nvidia. CUDA is parallel computing architecture for general purpose programming on graphics cards. CUDA includes special libraries and a C compiler. The libraries are sitting on top of the graphics card architecture to ease the programming for developers. The C compiler compiles the code the a specific GPU architecture. Having the C compiler taking care of the different GPU architectures adds the possibility to use the same code on different GPU architectures.

7.2 What tools are available for GPU based cracking?

The market for GPU-based password cracking is still young. The following four applications are able to brute-force passwords using multiple GPUs:

- BarsWF bruteforce;
- Extreme GPU Bruteforcer;
- IGHASHGPU;
- Elcomsoft.

Well known tools like John the Ripper and Cain and Able do not support GPU-based cracking. For this reseach we needed support for the following hashes (as defined in the project definition):

- NTLM;
- DCC;
- MD5;

- salted MD5;
- MD5 crypt;
- Oracle 11g (salted SHA-1).

All the tools above support at least one of these hashes, except MD5 crypt which was not supported. They are all active developed, maintained and supported. Two of the tools are commercial (Extreme GPU Bruteforcer (costs €50,-) and Elcomsoft (€599,- for 20 clients)), the other tools are free, sometimes only for non-commercial use. Not one of the tools published there source code. Elcomsoft delivers the most complete software package, but is also the most expensive one. Elcomsoft is the only tool which supports distributed GPU-based cracking. A complete overview of the cracking tools can be found in table 4.

7.3 What is the actual performance gain of GPU versus CPU-based cracking?

Because of the large amount of processing cores available in GPUs, the amount of GFLOPS available in GPUs is much larger compared to CPUs (a complete overview of GFLOPS can be found in table 3). This difference in performance is also visible in the results of our research, as can be seen in figure 15.

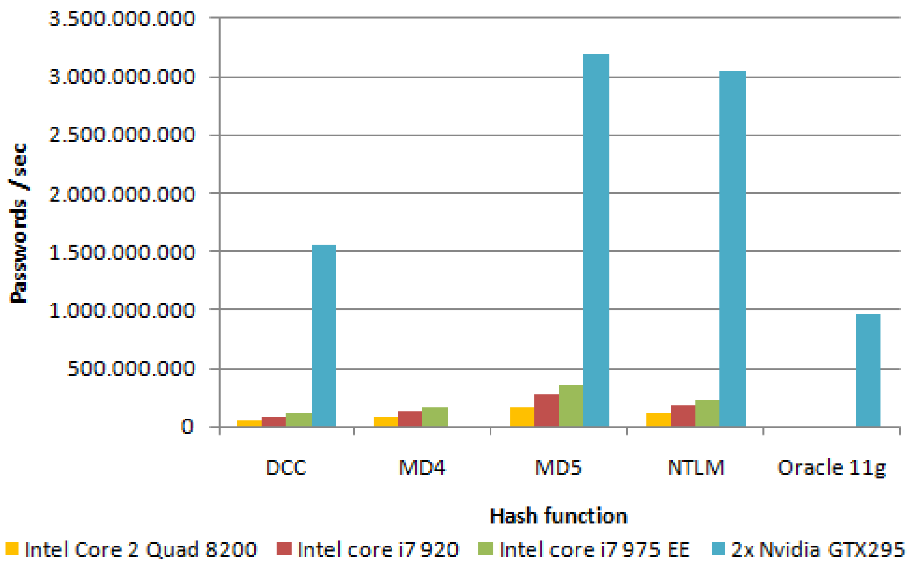


Figure 15: CPU vs GPU based on processed password / second for several CPUs and two Nvidia GTX295. This figure is a combination of the data available in table 2 and 7.

In table 6 the performance / price ratio and performance / power usage is presented. In this table performance is measured in passwords per second. Notice, in table 3 the same data is presented, but instead the performance is measured in GFLOPS. Our test system processed fourteen times more password per second on its GPUs in comparison to its CPU. The GPUs used are even forty times faster if measured in GFLOPS (based on table 3). It is clear GPUs lose some performance on non floating point operations.

	Nvidia GT295	**Nvidia GT295**	**Nvidia GT295**	**Nvidia GT295**	**Intel Core i7 920**
	Oracle11g	**NTLM**	**DCC**	**MD5**	
Power consumption[6] (Watt)	289	289	289	289	130
Performance (million passwords / sec	484	1550	788	1500	230
Price[7] (€)	400,-	400,-	400,-	400,-	227,-
Performance/Price ratio (passwords / sec / €)	1,21	3,88	1,97	3,75	1,01
Performance/Power consumption ratio (passwords / sec / Watt)	1,67	5,36	2,73	5,19	1,77

Table 6: The data in this table is almost the same as in table 3 and shows the performance/price and performance/power consumption ratio for the different hashes. Higher is better. But the performance measured in password per second instead of GFLOPS. The performance data is retrieved from our research, these can be found in section 6 [17, 20, 21].

Next to performance differences between GPUs and CPUs there are also performance differences between tools. Especially between the performance of MD5 cracking tools. BarsWF bruteforce is the fastest tool available for MD5 based hashes, but only if measured in password per second. If measured in time needed to crack a hash, there is no significant difference.

Hash	**Tool**	**Performance** (Million password / sec)
MD5	Elcomsoft	1750
	Extreme GPU Bruteforcer	2260
	IGHASHGPU	2770
	BarsWF bruteforce	3200
NTLM	Extreme GPU Bruteforcer	3050
	IGHASHGPU	3050
DCC	Extreme GPU Bruteforcer	1556
	IGHASHGPU	1565
Oracle 11g	IGHASHGPU	968

Table 7: Average processed password / second for every hash and tool. Based on these values it is possible to compute the expected time it takes to process all passwords of a certain key space. A graphical representation can be found in figure 15.

The differences in the amount of passwords per second are caused by the implementation of the hashing algorithm. These implementations differ per tool in efficiency. The performances of all cracking tools (in password per second) are shown in figure 15 and table 7. The other tools perform equally, comparing on basis of password per second.

When the performance of the different tools is compared on basis of the time needed to find a password, more differences in performance are visible. But the differences are really small and therefore not significant. Most of the differences are caused by the algorithm used to walk through the key space. A complete overview of the performance result can be found in graphs (figure 10, 12, 11 and 13.

7.4 What is the recommended password length for clients of KPMG?

KPMG gives advise to there customers to have passwords expired in a fifth of the time it takes to crack a password (computing all the passwords in the key space). This means if you have a password which can be cracked in ten months, the password expiration time must be two months. With the default expiration time of 90 days for Active Directory, the computation time for the whole key space must be at least 15 months [38]. One of following passwords are secure if you apply those rules on the passwords in figure 16 and by taking the near future into account:

- Nine or more characters with a key space containing all characters.
- Ten or more characters with a key space containing lowercase characters, uppercase characters and numbers.
- Twelve or more characters with a key space containing lowercase characters and numbers.

Figure 16 is based on a average of two billion passwords per second. Comparing these values with table 7, which shows the actual passwords per second as measured doing the research, the following can be concluded.

- MD5 and NTLM are 1.5 times faster;
- DCC is 1.25 times slower;
- Oracle 11g is two times slower.

In spite of the differences in speed, we advice for all the reviewed hashes the same password length. This is because of the available margins (this margin is caused by the rounding up of the password length). This means that MD5 does have less margin left in comparison with Oracle 11g.

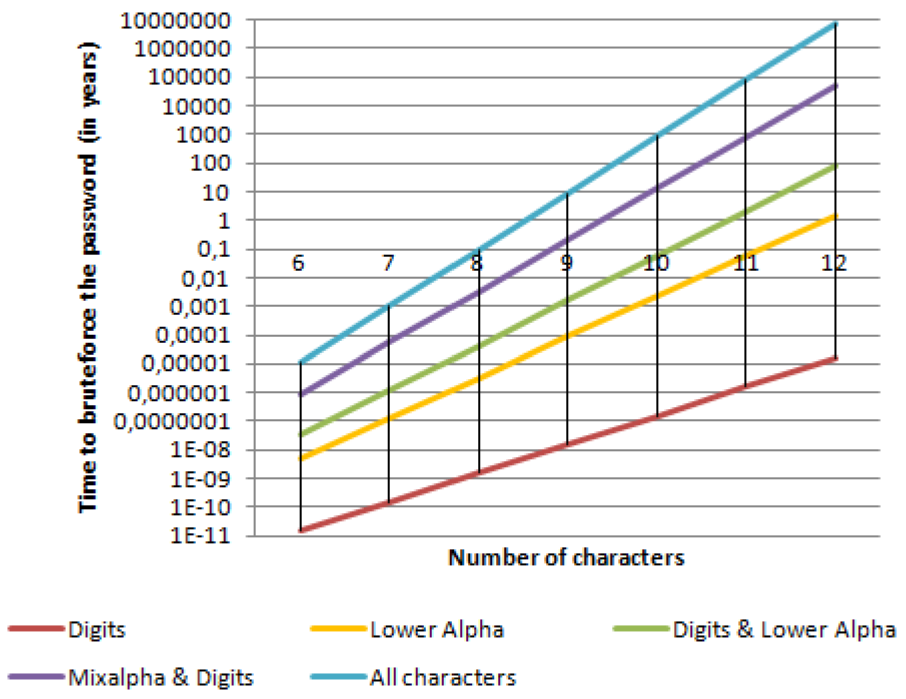


Figure 16: The relation between entropy and computation time for an average cracking system with a computation speed of two billion password per second. This value is based on data available in section 2.5.

8 Discussion

In the previous section the research questions are answered and some conclusions are drawn. In this section we will discuss some notes on the performance test executed. Also some tips about future research is given. We conclude this chapter with a look into the future of GPU-based password cracking.

During this research we executed a test to measure the password cracking performance on two Nvidia GT295 graphics cards. There are however a few side notes we have to mention, which may have influenced our results:

- The salt used during the Oracle 11g SHA-1 test was to short conform the Oracle 11g implementation. We do not expect this to have influenced the results, because of the small performance impact we have seen by adding a salt to a hash in the crack process.
- We generated a whole list of hashes we hoped to be able to crack, but due the limited amount of time, we could not crack all those hashes. Cracking all those hashes would have provided a better view on the performance of the test system.

During this research we did not test rainbow cracking and dictionary based cracking using GPUs. Because of the limited amount of available time. But, because of the huge amount of available processing power, it may also be interesting to use GPUs for these kind cracking tools. However we do not expect to see significant improvements in the time it takes to crack a hash. The many I/O operations to the table will form the bottleneck of the cracking process. We only except to see an improvement in time, if the I/O bottleneck can be avoided. This can be achieved by making the chains really long (in the order of several billion). With the performance of GPUs in mind, the time it would takes to recompute these chains will be short.

Finally we will give our future look about GPU-based password cracking. We expect that in the near future GPU-based password cracking will get much faster. During this research we made use of the GT200 series from Nvidia. Nvidia expects its new GT300 series to be released in the first half of 2010 [39]. ATI already released its new 5000 series. The first benchmarks of these graphics card are significant. Their fastest dual GPU card computes over more than 6 billion password per second with MD5[40]. Which is 4 times faster compared to the GPU used for this research. The new series from Nvidia are expected to provide the same performance gain.

Figure 17 shows the expected computation times for a system with two dual core graphics cards from the Nvidia GT300 or ATI 5000 series. The amount of password per second rises to a level of thirteen billion (13.000.000.000) per second. In spite of the performance increasement. The advice on password length and complexity remains the same.

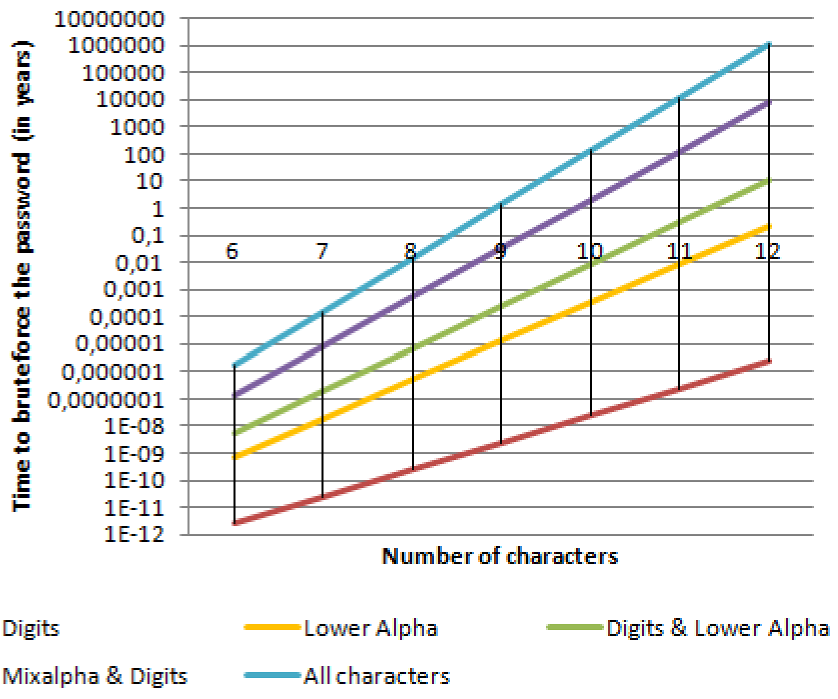


Figure 17: The future of the computation time for a system based on two modern graphics cards from the year 2010. This system would have a computation power of around thirteen billion MD5 or SHA1 passwords per second.

9 Acknowledgement

We would like to thank KPMG for the ability to execute our research and received encouragement, guidance and support from the initial to the final level. This enabled us to develop an understanding of the subject. In special we would like to thank our supervisors Marc Smeets, Stan Hegt and Pieter Ceelen.

Lastly, we offer our regards to our teachers and supervisors from the University of Amsterdam in any respect during the completion of the project.

Marcus Bakker & Roel van der Jagt

References

[1] Password advices, January 2009. URL https://www.google.com/accounts/PasswordHelp.

[2] Password advices, January 2009. URL http://www.netadvies.nl/advies/wachtwoord.html.

[3] Restrict repeating characters, January 2009. URL http://publib.boulder.ibm.com/iseries/v5r1/ic2924/index.htm?info/rzakz/rzakzqpwdlmtrep.htm.

[4] Google. Gmail, January 2009. URL http://www.gmail.com/.

[5] Random password strength, January 2009. URL http://www.redkestrel.co.uk/Articles/RandomPasswordStrength.html.

[6] Human memory limitations and web site usability, January 2009. URL http://webword.com/moving/memory.html.

[7] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography.* CRC Press, fifth edition, October 1996.

[8] Hash collision, January 2009. URL http://upload.wikimedia.org/wikipedia/commons/5/58/Hash_table_4_1_1_0_0_1_0_LL.svg.

[9] Thomas Fuhr and Thomas Peyrin. Cryptanalysis of radiogatun. Technical report, DCSSI Labs & Ingenico, 2008. http://eprint.iacr.org/.

[10] Hardware.Info. Cpu compare, January 2009. URL http://www.hardware.info/nl-NL/productdb/viewbenchmarks/.

[11] N. Warren. Benchmark comparison - pc password recovery tools, January 2009. URL http://www.cerberusgate.com/benchmark_comparison.htm.

[12] S.M. Aleksandrovich. World fastest md5 cracker barswf, January 2009. URL http://3.14.by/en/md5.

[13] InsidePro. Extreme gpu bruteforcer, January 2009. URL http://www.insidepro.com/eng/egb.shtml.

[14] I. Golubev. Ighashgpu v0.62, January 2009. URL http://www.golubev.com/files/ighashgpu/readme.htm.

[15] Elcomsoft Co. Ltd. Lightning hash cracker, January 2009. URL http://www.elcomsoft.com/lhc.html.

[16] T.R. Halfhill. Parallel processing with cuda. *Microprocessor Report*, jan 2008.

[17] Nvidia, January 2009. URL http://www.nvidia.com/.

[18] Y. Liu E. Wu. Emerging technology about gpgpu. In *Circuits and Systems - Asia Pacific Conference*, pages 618–622. IEEE, dec 2008.

[19] D. Geer. Taking the graphics processor beyond graphics. *Computer*, 38: 14–16, sep 2005.

[20] Intel. Ark — your source for information on intel® products, January 2009. URL http://ark.intel.com/.

[21] Tweakers.net. Tweakers.net pricewatch, January 2009. URL http://tweakers.net/pricewatch/.

[22] GPUreview. Gpureview - nvidia geforce gtx 295, January 2009. URL http://www.gpureview.com/GeForce-GTX-295-card-603.html.

[23] Intel. Processors, January 2009. URL http://www.intel.com/support/processors/sb/cs-023143.htm.

[24] The ntlm authentication protocol and security support provider, January 2009. URL http://davenport.sourceforge.net/ntlm.html#theLmResponse.

[25] Microsoft. Ntlm user authentication in windows, January 2009. URL http://support.microsoft.com/kb/102716.

[26] B. den Boer and A. Bosselaers. An attack on the last two rounds of md4. *Crypto*, pages 194–203, 1991.

[27] San Jose State University. Md4. URL http://www.cs.sjsu.edu/~stamp/crypto/PowerPoint_PDF/15_MD4.pdf.

[28] R. Chen. Microsft technet - windows confidential cached credentials, January 2009. URL http://207.46.16.252/en-us/magazine/2009.07.windowsconfidential.aspx.

[29] Passcape Software. Recovering domain cached passwords, January 2009. URL http://www.passcape.com/domain_cached_passwords.

[30] D. Niggebrugge. Cacheebr, the ms cache password brute forcer, January 2009. URL http://blog.distracted.nl/2009/05/cacheebr-ms-cache-password-brute-forcer.html.

[31] R. Rivest. *RFC: 1321. The MD5 Message-Digest Algorithm.* MIT Laboratory for Computer Science and RSA Data Security, Inc, April 1992.

[32] A. Sotirov, M. Stevens, J. Appelbaum, A. Lenstra, D. Molnar, D.A. Osvik, and B. de Weger. Md5 considered harmful today, December 2008. URL http://www.win.tue.nl/hashclash/rogue-ca/.

[33] A. Main. passphrases using the md5-based unix crypt(), 2009. URL http://search.cpan.org/~zefram/Authen-Passphrase-0.006/lib/Authen/Passphrase/MD5Crypt.pm.

[34] The Recurity Lablog. Oracle 0xdeadf00d, September 2007. URL http://www.phenoelit.net/lablog/archives/2007/09/22/oracle_0xdeadf00d/index.html.

[35] P. Finnigan's. Oracle 11g password algorithm revealed, September 2007. URL http://www.petefinnigan.com/weblog/archives/00001097.htm.

[36] X. Wang, Y.L. Yin, and H. Yu. Finding collisions in the full sha-1. In *CRYPTO*, pages 17–36, 2005.

[37] A. Zonenberg. Distributed hash cracker: A cross-platform gpu-accelerated password recovery system, April 2000.

[38] A. Juels. Password expiration: Like margarine and water?, May 2008. URL `http://www.rsa.com/blog/blog_entry.aspx?id=1286`.

[39] M. Rademaker. Hommeles in gpu-land, January 2009. URL `http://tweakers.net/reviews/1499/2009-hommeles-in-gpu-land.html`.

[40] I. Golubev. New results for md5 hashes for ati gpus, January 2009. URL `http://www.golubev.com/blog/?p=9`.

[41] K. Kuah. Motion estimation with intel streaming simd extensions 4 (intel sse4). Technical report, Intel, April 2007. `http://software.intel.com/en-us/articles/motion-estimation-with-intel-streaming-simd-extensions-4-intel-sse4/`.

[42] AsusTeK. Asustek product image engtx295, January 2009. URL `http://www.asus.com/product.aspx?P_ID=vuzvpginHnMTxCcr&templete=2`.

[43] D.S. Carstens. Human and social aspects of password authentication. Technical report, Florida Institute of Technology, USA, 2009.

Glossary

CA certificate

CA is an abbreviation for certificate authority. Who's main business is sighing certificates, to assure a certain degree of authenticity. The CA certificate plays a roll in the process of signing certificates. 19

hash

A hash is a one way function which converts a variable input into a fixt length (often between 128 bits and 512 bits) output. Its output is also called a fingerprint of the input value. It is not possible to retrieve the orginal input value by making use of the output value. 6, 8–11, 17–22, 32

Linux

Unix-like computer operating systems based in the Linux kernel. Their development process is one of the most important examples of open source development. 22

salt

A salt is a (sometimes secret) string that is used as one of the inputs to hash function. A salt is used to prevent rainbow table attacks. A rainwbow table is actually a large database storing a lots of password-hash pairs. Creating a rainbow table takes a lot of time, but is speeds up the process of cracking a hash. By adding salt (which will be different for every hash) you have to create a different rainbow for every salt value. This will defeat a rainbow-table attack. 8, 18–21

List of Acronyms

Central Processing Unit (CPU)

The central processing unit is the general processing unit in every computer. 4, 6, 10–13, 15, 17, 22, 34–36

Compute Unified Device Architecture (CUDA)

CUDA is a parallel computing architecture developed by Nvidia, which allow general purpose programming on GPUs from Nvidia. 14, 34

Domain Cached Credentials (DCC)

A salted NTLM hash stored on the local machine. 10, 18, 19, 25, 26, 29, 32, 34, 36, 37, 57

Data encryption standard (DES)

DES is a block cipher selected by the National Bureau of Standards and introduced in 1976 to be used in the United States. DES is now considered to be insecure for many applications. 20

Giga Floating Point Operations Per Second (GFLOPS)

GigaFLOPS is the same as 1.000.000.000 FLOPS. FLOPS are the number of floating point operations processed per second. FLOPS is used as a measure for the speed of processing units. 12, 13, 17, 35

General-Purpose Graphics Processing Unit (GPGPU)

A GPU used for more general purposes than only graphical computations. 11, 14, 17

Graphics Processing Unit (GPU)

The GPU is a processing unit optimised and ment for graphic operations. 4, 6, 10–15, 17, 22, 24, 34–36, 38, 39, 49

Kleynveld Peat Marwick Goerdeler (KPMG)

KPMG is an international company which offers Audit, Tax and Advisory services. 4, 37, 41

Lan manager (LM)

A Network Operating System (NOS) from Microsoft developed in cooperation with 3Com in 1987 as an answer on Novell's Netware. 18, 20

Message-Digest algorithm 5 (MD5)

MD5 is a hashing algorithm developed by Professor Ronald Rivest at MIT in 1994. It is the successor of MD4, which had several weaknesses. The message digest has a length of 128 bits. 10, 18–21, 25, 26, 28, 29, 32, 36, 37, 39, 57

Message-Digest algorithm 4 (MD4)

MD4 is a hashing algorithm developed by Professor Ronald Rivest at MIT in 1990. The message digest has a length of 128 bits. 10, 18–20

Moving Picture Experts Group (MPEG)

The Moving Picture Experts Group was formed by the ISO to set standards for audio and video compression and transmission. 14

National Institute of Standards and Technology (NIST)

The NIST is a measurement standards laboratory which is a non-regulatory agency of the United States Department of Commerce. 20, 47

NT Lan manager (NTLM)

Is a Microsoft authentication protocol. It is also used to generate a hash of a user password. 10, 18–20, 25, 26, 29, 32, 36, 37, 57

Original Equipment Manufacturer (OEM)

OEM refers to the company that originally manufactured the product 18

Open Computing Language (OpenCL)

OpenCL is a cross-platform API to support general purpose programming on GPUs. 14

Security Accounts Manager (SAM)

SAM is the accounts database used by Microsoft Windows NT, Windows 2000, and later Microsoft Operating systems to store user passwords in hashed format. 18

Secure Hashing Algorithm 1 (SHA-1)

SHA1 is a hashing algorithm, which was published by the NIST. The message digest has a length of 160 bits. 18, 20, 38

Single Instruction, Multiple Data (SIMD)

SIMD is a technique used to achieve data level parallelism. 12, 34, 47

Single Instruction, Single Data (SISD)

SISD is the technique used in CPUs to process data. Data is processed in a serial way. 12, 34

Streaming SIMD Extensions version 5 (SSE5)

SSE5 is version 5 of the Streaming SIMD Extensions, which extends the x86 instruction set. SIMD is implemented in the SSE instruction set.[41] 13

Streaming SIMD Extensions version 4 (SSE4)

SSE4 is version 4 of the Streaming SIMD Extensions, which extends the x86 instruction set. SIMD is implemented in the SSE instruction set.[41] 10

Streaming SIMD Extensions version 2 (SSE2)

SSE2 is version 2 of the Streaming SIMD Extensions, which extends the x86 instruction set. SIMD is implemented in the SSE instruction set.[41] 10

A Specifications test system

A.1 Cracking server

The cracking server has the following specifications. More details about the graphic card can be found in the next section (section A.2):

System specifications:	
CPU	Intel Core i7 920 (s1366,2.66GHz)
Memory	6GB DDR3 1066MHz in triple channel mode
Motherboard	ASUS P6T7 WS SuperComp (place for four GPUs)
Harddisk	3x 500GB S-ATAII (one disk for the OS, the other two disks in RAID0 for data storage)
GPU	ASUS ENGTX295 2x
Power supply	1500 watt
Operating System	Microsoft Windows server 2008 x64

A.2 Graphics card

The Nvidia GTX295 card we used for this research has the following specifications[8]. Notice that some manufacturer changes these specifications for their own implementation. We checked this on the site of Asus[9] (Asus is the manufacturer of the card). At their site we found the same specifications as found at Nvidia's site.

GPU Engine Specs:	
Processor Cores	480 (240 x 2)
Graphics Clock (MHz)	576
Processor Clock (MHz)	1242
Texture Fill Rate (billion/sec)	92.2
Memory Specs:	
Memory Clock (MHz)	999
Standard Memory Config	1792MB (896MB x 2) GDDR3
Memory Interface Width	896-bit (448-bit x 2)
Memory Bandwidth (GB/sec)	223.8
Feature Support:	
NVIDIA SLI®-ready	4 (Quad)
NVIDIA PureVideo® Technology	PVHD
NVIDIA PhysX™-ready	Yes
NVIDIA CUDA™ Technology	Yes
Microsoft DirectX	10
OpenGL	2.1
Bus Support	PCI-E 2.0 x16
Certified for Windows Vista	Yes
Display Support:	

[8] www.nvidia.co.uk/object/product_geforce_gtx_295_uk.htm
[9] www.asus.nl/product.aspx?P_ID=gYXUhIbeFszdRPSY&templete=2

Figure 18: The ASUS GTX295 graphics card used during the research.[42]

Maximum Digital Resolution	2560x1600
Maximum VGA Resolution	2048x1536
Standard Display Connectors	Two Dual Link DVI, HDTV
Multi Monitor	Yes
HDCP	Yes
HDMI	Yes
Audio Input for HDMI	SPDIF
Standard Graphics Card Dimensions:	
Height	4.376 inches (111 mm)
Length	10.5 inches (267 mm)
Width	Dual-Slot
Thermal and Power Specs:	
Maximum GPU Tempurature (in C)	105 C
Maximum Graphics Card Power (W)	289 W
Minimum System Power Requirement (W)	680 W
Supplementary Power Connectors	6-pin and 8-pin

B Raw test results

Table 8: Raw test results of the password cracking tests.

	Elcomsoft (sec)	**Extreme GPU Bruteforcer** (sec)	**IGHASHGPU** (sec)	**BarsWF bruteforce** (sec)
Password	**MD5**			
789740	1	1	0	4
455278	1	1	0	2
651773	1	1	0	2
109642	1	1	0	3
511881	1	1	0	3
bnlwoh	1	1	0	3
fxlomv	1	2	0	3
pncsol	1	1	0	3
kqmenf	1	1	0	2
wlwuuo	1	2	0	3
ylfw21	2	3	1	3
mqy1ll	2	2	0	4
5czznz	2	3	0	5
82q31w	2	3	1	4
16tfdj	2	2	0	4
CqBAQh	5	6	6	9
wTTCiG	16	20	12	14
IyNSTP	20	24	7	11
D6NR5W	24	27	19	13
vPlS1x	12	12	18	19
⟩,%w+7	219	117	35	21
F)EVZm	47	47	169	85
3⟨DW'8	225	121	23	87
de∼'1t	73	69	54	206
$∼2D!{	282	280	8	44
85650786	1	1	0	5
63099104	1	1	0	4
74567940	1	1	0	4
46000701	1	1	0	4
54223502	1	1	0	4
eihptwjw	99	88	61	39
zbppirmj	46	40	28	51
uyxdebyu	93	82	15	79
ruakcqjm	57	51	11	10
symrtqym	60	53	60	44
gprjepsw	954	814	152	764

Table 8 – Continued

Password	**Elcomsoft** (sec)	**Extreme GPU Bruteforcer** (sec)	**IGHASHGPU** (sec)	**BarsWF bruteforce** (sec)
	MD5			
7nasv4k4	1265	1082	632	319
9tihpdae	206	175	446	523
ws9n2vfg	291	250	826	301
bzd9kulm	545	462	322	400
nfwC50P2 Pz2aNRLT jfEeUyG8 jYeffeHD QT0Z0cr0				64598
HD'I5fP$ 'hxgFM,4 uE⟨An_(y 6Nˆl[1@w %EZ5?E+∗				
0119425077	6	6	2	9
6433423789	7	7	2	9
2762342329	7	7	2	10
2760161644	5	4	1	10
0124732837	6	6	3	10
sbbebgjxry kpqfsfzziz mylpkghyuz pqcfmfdknc hlxrpydpzp				4181
nu08oowe90 Yqq6mycDOq 427MSE58Ba MA12sKeHs6 hrVwaypVTr				
oVzjq2aCb8 Yqq6mycDOq 427MSE58Ba MA12sKeHs6 hrVwaypVTr				
⟩GM-6⟨:zdr Z0U}G%I6aC }I6"1(⟨r[t cUfWy%⟩97) gIjtgQP'∼{				

Table 8 – Continued

Password	**Elcomsoft** (sec)	**Extreme GPU Bruteforcer** (sec)	**IGHASHGPU** (sec)	**BarsWF bruteforce** (sec)
	MD5			
371787564926	420	326	356	210
685225732402	180	140	134	537
763251588665	384	298	226	246
695128406731	142	111	143	368
590747117552	211	162	210	151
fxjoctruxlze				
zpejprmignwf				
gyocxxizzzzl				
pzcexkphgyys				
kffrvaqsxrgb				
r4t9vwpyuymt				
qbg3s9tdei75				
0yrfnc2ygqo7				
5c3iiptl6ugc				
i695fuzahh5z				
aFqGGQothBdR				
pKIprkGZDBMb				
8OFX2bGC7d2f				
AMANEesDJoPK				
xgI5ZkaTvYDN				
+B)V,{.!yD4J				
MfJ'.!RDUDpS				
A:EBl@C?bUwX				
]N}lto")\—)r				
9:z2{Gl'rAr4				

Table 9: Raw test results of the password cracking tests.

	Extreme GPU Bruteforcer (sec)	**IGHASHGPU** (sec)	**Extreme GPU Bruteforcer** (sec)	**IGHASHGPU** (sec)	**RainbowCrack** (sec)	**IGHASHGPU** (sec)
Password	**NTLM**		**DCC**			**Oracle 11g**
789740	1	0	1	0		0
455278	1	0	1	0		1
651773	1	0	1	0		1
109642	1	0	1	0		0
511881	1	0	1	0		0
bnlwoh	1	0	1	0		0
fxlomv	1	0	2	0		0
pncsol	1	0	1	0		1
kqmenf	1	0	1	0		0
wlwuuo	1	0	2	0		0
ylfw21	2	1	5	1		2
mqy1ll	2	0	3	0		1
5czznz	2	0	4	1		1
82q31w	2	1	4	1		2
16tfdj	1	0	2	0		1
CqBAQh	4	5	8	10		17
wTTCiG	15	11	30	21		34
IyNSTP	18	6	36	12		20
D6NR5W	21	18	41	34		56
vPlS1x	9	17	18	32		52
⟩,%w+7	89	32	177	63		102
F)EVZm	36	154	71	305		498
3⟨DW’8	92	21	183	42		69
de∼‘1t	52	49	105	97		157
$∼2D!{	211	8	422	15		25
85650786	1	0	1	0		1
63099104	1	0	1	0		1
74567940	1	0	1	0		1
46000701	1	0	1	0		0
54223502	1	0	1	0		0
eihptwjw	66	55	132	108		173
zbppirmj	30	26	61	50		81
uyxdebyu	62	13	124	26		43
ruakcqjm	38	10	77	19		30
symrtqym	40	55	80	106		172
gprjepsw	599	138	1238	271		438
7nasv4k4	796	578	1647	1137		1836
9tihpdae	129	406	267	799		1295
ws9n2vfg	184	749	380	1469		2378

Table 9 – Continued

	Extreme GPU Bruteforcer (sec)	**IGHASHGPU** (sec)	**Extreme GPU Bruteforcer** (sec)	**IGHASHGPU** (sec)	**RainbowCrack** (sec)	**IGHASHGPU** (sec)
Password	**NTLM**		**DCC**			**Oracle 11g**
bzd9kulm	340	293	703	576		933
nfwC50P2	36045		72168			
Pz2aNRLT						
jfEeUyG8						
jYeffeHD						
QT0Z0cr0						
HD'I5fP$						
'hxgFM,4						
uE⟨An_(y						
6Nˆl[1@w						
%EZ5?E+∗						
0119425077	4	2	9	3		6
6433423789	5	2	10	3		6
2762342329	5	2	10	3		5
2760161644	3	1	7	2		3
0124732837	4	3	9	5		9
sbbebgjxry	48067		96027			
kpqfsfzziz						
mylpkghyuz						
pqcfmfdknc						
hlxrpydpzp						
nu08oowe90						
Yqq6mycDOq						
427MSE58Ba						
MA12sKeHs6						
hrVwaypVTr						
oVzjq2aCb8						
Yqq6mycDOq						
427MSE58Ba						
MA12sKeHs6						
hrVwaypVTr						
⟩GM-6⟨:zdr						
Z0U}G%I6aC						
}I6"1(⟨r[t						
cUfWy%⟩97)						
gIjtgQP'∼{						
371787564926	237	324	497	630		1017
685225732402	102	121	213	235		380
763251588665	217	206	454	401		647

Table 9 – Continued

	Extreme GPU Bruteforcer (sec)	**IGHASHGPU** (sec)	**Extreme GPU Bruteforcer** (sec)	**IGHASHGPU** (sec)	**RainbowCrack** (sec)	**IGHASHGPU** (sec)
Password	**NTLM**		**DCC**			**Oracle 11g**
695128406731	80	130	169	252		408
590747117552	118	191	248	371		601
fxjoctruxlze						
zpejprmignwf						
gyocxxizzzzl						
pzcexkphgyys						
kffrvaqsxrgb						
r4t9vwpyuymt						
qbg3s9tdei75						
0yrfnc2ygqo7						
5c3iiptl6ugc						
i695fuzahh5z						
aFqGGQothBdR						
pKIprkGZDBMb						
8OFX2bGC7d2f						
AMANEesDJoPK						
xgI5ZkaTvYDN						
+B)V,{.!yD4J						
MfJ’.!RDUDpS						
A:EBl@C?bUwX						
]N}lto”)\—)r						
9:z2{Gl‘rAr4						

C Commands: cracking tools

An overview of the commands used for the different tools.

Elcomsoft

Character sets:

• `-c0`	0-9
• `-ca`	a-b
• `-ca0`	a-b, 0-9
• `-caA0`	a-b, A-B, 0-9
• `-caA0!`	a-b, A-B, 0-9, punctuation marks

Example:

```
lhc.exe -l6 -L12 -i0123 -ca md5hash.txt
```

`-l6` specifies a minimal password length of 6, `-L12` specifies maximum password length of 12 and `-i0123` defines the GPUs to use for cracking. The last parameter is the text file containing the hash to crack.

Extreme GPU Bruteforcer

Every hash has a different executable:

- `MD5.exe`
- `NTLM.exe`
- `MSCACHE.exe`

Character sets:

• `digits.ini`	0-9
• `lower.ini`	a-b
• `lowerdigits.ini`	a-b, 0-9
• `lowerupperdigits.ini`	a-b, A-B, 0-9
• `all.ini`	a-b, A-B, 0-9, punctuation marks

Inside the .ini files the character set is specified as mentioned above, and the following parameters:

• `MaxLength=12`	Maximal password length
• `PasswordsInThread=3000,3000,3000,3000`	Number of passwords to be processed by a thread. 6000 is the default. 3000 gave us the best results.
• `StreamProcessors=240,240,240,240`	Number of stream processors. Our GPUs had a total 240 steam processors each.
• `CurrentDevice=1,2,3,4`	The GPU devices to use for the cracking session.

Example:

```
MD5.exe lower.ini md5hash.txt
```

The last parameter is the text file with the hash to crack.

IGHASHGPU

Every hash has different parameters:

• `-t:md5`	MD5
• `-t:md4 -unicode`	NTLM
• `-t:dcc -usalt:uvasne`	DCC
• `-t:sha1 -asalt:uvasne`	Oracle 11g

Character sets:

• `-c:d`	0-9
• `-c:s`	a-b
• `-c:sd`	a-b, 0-9
• `-c:csd`	a-b, A-B, 0-9
• `-c:a`	a-b, A-B, 0-9, punctuation marks

Example:

```
ighashgpu.exe -max:12 -c:s -t:md5 -h:acbd18db4cc2f85cedef654fccc4a4d8
```

`-max:12` sets the maximum password length to 12 characters. The last parameter is the hash to crack.

BarsWF bruteforce

Character sets:

• `-c 0`	0-9
• `-c a`	a-b
• `-c 0a`	a-b, 0-9
• `-c aA0`	a-b, A-B, 0-9
• `-c aA0~`	a-b, A-B, 0-9, punctuation marks

Example:

```
BarsWF_CUDA_x64.exe -c a -h acbd18db4cc2f85cedef654fccc4a4d8
```

The last parameter is the hash value to crack.

D Random password generator

To generate the random passwords and the corresponding hashes needed for the research we wrote a python script.

```python
#!/usr/bin/python

import random, sys, getopt, re, hashlib, binascii, Crypto.Hash.MD4, md5_crypt

MD4 = Crypto.Hash.MD4
PASS_LENGTH = 15
AMOUNT_OF_PASSWORDS = 1
characters = []
hash_output = ""
hash_salt = ""
password = ""

# Print the help information
def usage():
    print "Generates random passwords and the corresponding hashes. Available hashes are:"
    print "MD5, MD5 crypt, NTLM, MSCACHE and ORACLE 11g\n"
    print "-a, --amount=AMOUNT     amount of passwords"
    print "-d, --digit             use digits for the password"
    print "-h, --help              display this help information"
    print "-l, --lower             use lower case letters in the password"
    print "-L, --length=LENGTH     specify the length of the password (default 15)"
    print "-o, --output=HASH       define the hash to output. Use \"all\" to output all"
    print "                            available hashes"
    print "-p, --punctuation       use punctuation marks in the password"
    print "-P, --password=PASSOWRD set a predefined password. No random passwords are used"
    print "-s, --salt=SALT         define the salt for the hash"
    print "-u, --upper             use upper case letters in the password"
    quit()

# Determine if the argument is a integer
def isInt(integer):
    if re.search("^[0-9]+$", integer):
        return True
    else:
        return False

# Capture the options and the corresponding arguments
if len(sys.argv) > 1:
    try:
        opts, args = getopt.getopt(sys.argv[1:], "L:hdpula:o:s:P:", \
            ["length=", "help", "digit", "punctuation", "upper", "lower", \
             "amount=", "output=", "salt=", "password="])
        for opt, arg in opts:
            if opt in ("-a", "--amount"):
                if (isInt(arg)):
                    AMOUNT_OF_PASSWORDS = int(arg)
                else:
                    print arg, "is no integer"
                    quit()
            elif opt in ("-L", "--length"):
                if (isInt(arg)):
                    PASS_LENGTH = int(arg)
                else:
                    print arg, "is no integer"
                    quit()
            elif opt in ("-h", "--help"):
                usage()
            elif opt in ("-d", "--digit"):
                characters += range(48,58)
            elif opt in ("-p", "--punctuation"):
                characters += range(32,48)
                characters += range(58,65)
                characters += range(91,97)
                characters += range(123,127)
            elif opt in ("-u", "--upper"):
                characters += range(65,91)
            elif opt in ("-l", "--lower"):
                characters += range(97,123)
            elif opt in ("-o", "--output"):
                hash_output = arg
            elif opt in ("-s", "--salt"):
                hash_salt = arg
            elif opt in ("-P", "--password"):
                password = arg
            else:
                usage()
    except getopt.GetoptError:
        usage()

# Set the default settings for the different kind of character sets
if len(characters) == 0:
    characters += range(32,127)

# Generate a hash and return the result
def generateHash(password, hash, salt = ""):
    if re.search("^md5$", hash, re.IGNORECASE):
        m = hashlib.md5()
        m.update(password)
        return m.hexdigest().upper()
    elif re.search("(^md5crypt$|^md5c$)", hash, re.IGNORECASE):
        return md5_crypt.md5crypt(password, salt)
        # + "\n" + md5_crypt.md5crypt(password, salt, "$6)
        #m = hashlib.md5()
        #m.update(salt+password)
        #return m.hexdigest().upper()
    elif re.search("(^oracle|^oracle11$|^oracle11g$)", hash, re.IGNORECASE):
        m = hashlib.sha1()
        m.update(password)
        salt = binascii.hexlify(salt)
        m.update(binascii.a2b_hex(salt))
        return str.upper(m.hexdigest()) + " - " + salt
    elif re.search("^NTLM$", hash, re.IGNORECASE):
        m = MD4.new()
        m.update(unicode(password).encode('utf-16-le'))
        return m.hexdigest().upper()
    elif re.search("^MSCACHE$", hash, re.IGNORECASE):
        m1 = MD4.new()
        m2 = MD4.new()
        m1.update(unicode(password).encode('utf-16-le'))
        ntlm = m1.digest()
        ntlm+=unicode(salt.lower()).encode('utf-16-le')
        m2.update(ntlm)
        return m2.hexdigest().upper()
    else:
        print "Unknown hash or not supported"
        quit()

# Generate the passwords, the corresponding hashes and print out the result
```

```python
for i in range(AMOUNT_OF_PASSWORDS):
    if password == "":  # Check if a predefined password is used
        while len(password) <> PASS_LENGTH:
            password += chr(random.choice(characters))

    # Generate the hashes
    if re.search("^all$", hash_output, re.IGNORECASE):
        print password
        print "MD5        :", generateHash(password, "md5")
        print "MD5 crypt  :", generateHash(password, "md5c", hash_salt)
        print "Oracle 11g :", generateHash(password, "oracle11g", hash_salt)
        print "NTLM       :", generateHash(password, "ntlm")
        print "MSCACHE    :", generateHash(password, "mscache", hash_salt)
        print "------------------------------------------------------------"
    elif hash_output <> "":
        hash_text = "\n" + hash_output.upper() + ((11 - len(hash_output)) * " ") + ": "
        print password + hash_text + generateHash(password, hash_output, hash_salt)
    else:
        print password
    password = ""
```