Radboud University Nijmegen
Faculty of Science
Kerckhoffs Institute

Master of Science Thesis

GPU-based Password Cracking

On the Security of Password Hashing Schemes regarding Advances in Graphics Processing Units

by

Martijn Sprengers
m.sprengers@student.ru.nl

Supervisors:
Dr. L. Batina (Radboud University Nijmegen)
Ir. S. Hegt (KPMG IT Advisory)
Ir. P. Ceelen (KPMG IT Advisory)

Radboud University Nijmegen

Thesis number: 646
Final Version

Abstract

Since users rely on passwords to authenticate themselves to computer systems, adversaries attempt to recover those passwords. To prevent such a recovery, various password hashing schemes can be used to store passwords securely. However, recent advances in the graphics processing unit (GPU) hardware challenge the way we have to look at secure password storage. GPU's have proven to be suitable for cryptographic operations and provide a significant speedup in performance compared to traditional central processing units (CPU's).

This research focuses on the security requirements and properties of prevalent password hashing schemes. Moreover, we present a proof of concept that launches an exhaustive search attack on the MD5-crypt password hashing scheme using modern GPU's. We show that it is possible to achieve a performance of 880 000 hashes per second, using different optimization techniques. Therefore our implementation, executed on a typical GPU, is more than 30 times faster than equally priced CPU hardware. With this performance increase, 'complex' passwords with a length of 8 characters are now becoming feasible to crack. In addition, we show that between 50% and 80% of the passwords in a leaked database could be recovered within 2 months of computation time on one Nvidia GeForce 295 GTX.

Preface

This master thesis is a result of the research I conducted during my graduation project from September 2010 until February 2011. This project is the final part of my master 'Computer Security' which is taught at the Kerckhoffs Institute, a collaboration between the University of Twente, the Eindhoven University of Technology and the Radboud University Nijmegen. The research was carried out at 'KPMG IT Advisory: ICT Security & Control (ISC)' in Amstelveen, the Netherlands. This business unit is specialized in advisory and audit services on information security issues.

First of all, I would like to thank Dr. L. Batina, member of the Digital Security department of the Radboud University Nijmegen, for supervising this project and putting effort in my research. Furthermore, much gratitude goes out to my supervisors at KPMG, Ir. S. Hegt and Ir. P. Ceelen. I would like to thank them for investing their time and effort, which, together with their professional skills, helped me considerably in completing this project.

In addition, I would like to thank Mr. P.H. Kamp for his comments and discussions on the design of his MD5-crypt algorithm, which provided me with more insight about the context it was created in. Furthermore, my gratitude goes out to Dr. B. de Weger and Mr. M. Stevens, who provided me with comments on the key-stretching technique and the break of the collision resistant property of the MD5 hash function.

I would also like to thank Ir. P. Kornelisse, Director at KPMG ISC, for giving me the opportunity to write this thesis at KPMG IT Advisory, M. Smeets MSc. for his discussions on GPU hardware and M. van Veen MSc., who provided me with a fast password generation algorithm. Finally, I would like to thank the rest of the ISC team for their openness, professionalism and the social work environment they offered, which made me decide to stay at KPMG after my graduation.

Martijn Sprengers, Amstelveen, February 2011

Contents

Chapter 1

Introduction

1.1 Introduction

Many software services provide an authentication system that relies on a user name and password combination. Initially, this password is generated by the user and stored in a safe location on the system. To make sure that these passwords are still safe even if the security of the location can not be guaranteed, it is common to use a cryptographic hash function to calculate the digest of the password and store this together with the users credentials. When a user authenticates himself to the system again, the digest of the plaintext password is calculated and compared to the stored digest.

Due to the cryptographic properties of the hash function, the digest of the password is not easily reversible and therefore the probability that an adversary learns partial information about the password should be proportional to the work he invests and the predictability of the password distribution. However, it is the latter property that fails for human generated passwords. Therefore, most deficiencies of password authentication systems arise from human memory limitations. Users tend to pick passwords that are easy to remember and do not contain sufficient randomness, which lead to predictable passwords. This enables an adversary to generate possible candidate passwords, calculate the digest and compare them with the stored digest. Password hashing schemes have been designed to decrease the feasibility of such an attack, e.g. by increasing the complexity of the calculations. However, many of those password hashing schemes have been designed in the mid nineties and since Moore's law is still valid, it is doubted if these schemes still provide enough security. Moreover, new hardware platforms have been designed to execute such attacks even faster. One of these platforms is the Graphics Processing Unit (GPU).

Since the late nineties GPU's have been developed and improved. They have proven to be very suitable for processing parallel tasks and calculating floating point operations. It is especially the parallel design of a GPU that makes it suitable for cryptographic functions. While the advantages of GPU's in other areas (like graphical design and game-industry) have already been recognized, the cryptographic community was not able to use them due to the lack of user-friendly programming API's and the lack of support for integer arithmetic. However, GPU producers have dealt with those shortcomings.

1.2 Related work

After Thompson et al. [70] had shown that GPU's could be used for general purpose computing too, in contrast to specific graphics applications, Cook et al. [15] showed that GPU's could be used for cryptography. However, due the lack of integer arithmetic and support of API's, no remarkable speedup was gained. After general programming API's for GPU's became available, Yang and Goodman[77]

and Harrison and Waldron[29] showed that contemporary GPU's can outperform high-performance Central Processing Units (CPU's) on symmetric cryptographic computations, yielding speedups of at most 60 times for the DES symmetric key algorithm. Moreover, GPU implementations of the AES algorithm have been extensively reviewed [28, 41, 20, 12]. Szerwinski et al.[69] and Harrison et al.[30] showed how GPU's could be used for asymmetric cryptography, while Bernstein et al.[9] showed that it is possible to reach up to 481 million modular multiplications per second on an Nvidia GTX 295, in order to break the Certicom elliptic curve cryptosystem (ECC) challenge[1] [10, 8]. In addition, cryptographic hash functions, such as MD5 [40, 32] and Blowfish [48], have been implemented on graphic cards too, yielding significant speed ups over CPU's.

1.3 Scope and contributions

This research focuses on launching exhaustive search attacks on authentication mechanisms that use password hashing schemes based on a cryptographic hash function (such as MD5). We focus on how these password schemes can be efficiently implemented on GPU's, which can initiate massive parallel execution paths at low cost, compared to a typical CPU. In particular, the password hashing scheme MD5-crypt is reviewed. Therefore, our main research question is:

How do graphics processing units effect the performance of exhaustive search attacks on password hashing schemes like MD5-crypt?

With the research question defined, we summarize the contributions as follows:

- We define the security properties of password hashing schemes based on the properties of cryptographic hash functions. The trade-off (or paradox) *'slow hashing, fast authentication'* will be emphasized.
- We identify the current and future attacker models on password hashing schemes. We determine if the recent break of a cryptographic property of MD5[72, 67] affects the security of the password hashing scheme MD5-crypt. Moreover, we try to determine to what extent the break of collision resistance properties influence the security of password hashing schemes in general.
- We show how different optimization strategies for GPU's could be used to get a maximum performance increase over CPU's.

[1] See http://www.certicom.com/index.php/the-certicom-ecc-challenge for more information.

- We publish a working proof of concept, which is the first and fastest GPU implementation to crack password hashes for MD5-crypt.
- We argue to what extent password hashing schemes should be improved in order to withstand exhaustive search attacks executed on near-future hardware.

1.4 Research methodology

To answer our research question, we use the following research methodology:

- *Literature study.* We try to identify the known optimization techniques for GPU's and cryptographic applications. Furthermore, we specify the properties of common password hashing schemes and see how they protect against known attacker models. Moreover, given the recent break of the collision resistance property of MD5, we determine whether this influences the security of MD5-crypt.
- *Theoretic and practice based models.* In order to show why GPU's yield a performance increase over CPU's, we define a theoretical model to estimate the maximum speed increase for MD5-crypt. Furthermore, we compare this model to known practice based models.
- *Proof of concept.* We develop a GPU implementation of the MD5-crypt password hashing scheme and describe how optimizations are implemented. Based on an experimental evaluation, we try to determine whether exhaustive search attacks on password hashing schemes are feasible with contemporary GPU's.

1.5 Relevance

People identifying themselves to systems and Internet services have to specify a password during their first authentication process. From then, the service provider is responsible for the storage of their credentials. While most databases and password storages systems are protected, either by corporate network, 'SAM' file (on Windows) or 'shadow' file (on UNIX), password hashing plays an important role when the confidentiality of such databases and files, together with the rest of the user's information, can not be guaranteed by the system. This can the case when the data are lost, stolen or published, e.g. by an disgruntled employee or malicious user. If the password hashing scheme is poorly designed, end-user passwords can be recovered easily. With the plain text password, malicious users can then authenticate themselves with another identity on the specific system or other services (since most end-users use the same password for multiple services).

To clarify the relevance, we quote the designer of MD5-crypt, Poul-Hennig Kamp:

> "The hashing used for protection of passwords should be strong enough to make it unfeasible to brute-force any but the most trivial passwords across all the users. The driving factor (in the design) was the execution time: I wanted it to be big enough to make key-space searches unappetizing. I still believe that MD5-crypt is safe from all but key-space searches.
>
> In my naive youth, I seriously expected that when I had pointed out that a password hashing did not need to be stable over any time longer than the individual passwords it protected, real card-carrying cryptographers would spend some time solving this practical problem once and for all with a family of configurable password scramblers. In fact they did not, as it was to them a problem they had already solved theoretically."

MD5-crypt and other password hashing schemes are designed without sustainable security requirements and are actually the product of the differential attacks on the DES encryption scheme in the early nineties. Since then, MD5-crypt was used as the standard password hashing scheme in most Unix variants, such as BSD and Linux. Moreover, corporations like Cisco have it employed in their routers and the RIPE Network Coordination Centre stores the MD5-crypt hashes, used to authenticate their users, in public. If the security of MD5-crypt fails, it will have a large impact on the confidentiality and integrity of systems and services.

1.6 External validity

The password hashing scheme that will be reviewed in this work, MD5-crypt, was designed in the early nineties and has not been updated ever since. Subsequent password hashing schemes, such as SHA-crypt[21], are based on the same design as MD5-crypt. They only differ in the way how the underlying hash functions are implemented. Therefore, if our research proves to be successful, it could easily be adapted to other password hashing schemes, making our method generic. The same generalization applies for our efficient GPU implementation of the MD5 hash function, since it is very similar in design to other commonly used hash functions.

1.7 Outline

Chapter 2 introduces the generic attacks on cryptographic systems and the feasibility of such attacks on current key sizes. Chapter 3 describes the properties and

requirements of hash functions and the MD5 hash function in particular. Then, Chapter 4 describes the properties and requirements of password hashing schemes based on those cryptographic hash functions. Further more, specific attacks and attacker models to password hashing schemes are defined. To describe our attack with GPU's on one specific password hashing scheme, Chapter 5 contains an introduction to GPU hardware and programming models. Chapter 6 will cover our approach to launch an exhaustive search attack with GPU's on MD5-crypt. To maximize the performance of our attack, our optimization strategies and efficient GPU implementation will be described in Chapter 7. Chapter 8 then contains the results of the experimental evaluation for our proof of concept. Finally, the conclusions and future work are presented in Chapter 9.

Chapter 2

Attacking cryptographic systems

This chapter describes general attacks to cryptographic systems, which include hash functions and password hashing schemes based on those hash functions. Moreover, special attack enhancements, such as bit slicing and specialized hardware, are described.

2.1 Generic attacks

This section describes the attacks that hold for all cryptographic functions, including hash functions and password hashing schemes.

2.1.1 Exhaustive search attack

The easiest to perform and most powerful attack is the exhaustive search attack. However, most secure cryptographic systems have large key spaces, which make it impossible to find the key in a feasible amount of time. To decrease the time needed for cryptanalytic attacks to find a key, either the time to apply the cryptographic function should be decreased or the available computing power should be increased. Of course combining the two is even more advantageous for such attacks. For example, consider a cryptographic hash function (e.g. MD5) with an output space of 128 bit. If all outcomes are equally likely, the search space would be 2^{128}, which is not feasible for modern hardware. The European Network of Excellence in Cryptology II (ECRYPT II), publishes an annual report on algorithms and keysizes for both symmetric and asymmetric cryptographic applications. For 2010, Table 2.1 considers the feasibility of launching an exhaustive search attack on different keysizes for specific cryptographic applications (after [64, 54]).

Duration	Symmetric	RSA	ECC
Days-hours	50	512	100
5 years	73	1024	146
10-20 years	103	2048	206
30-50 years	141	4096	282

Table 2.1: Duration of exhaustive search attacks on key sizes, in bit length, for specific cryptographic applications. (Assumptions: no quantum computers; no breakthroughs; limited budget)

The feasibility to attack current password hashing schemes is shown by Clair et al.[14]. They developed an analytical model to understand the time required to recover random passwords. Their empirical study suggests that current systems vulnerable to exhaustive search attacks will be obsolete in 2-10 years.

Exhaustive search is commonly used in combination with the following types of attack[43]:

- *Known plaintext* With this type of attack, the adversary has access to both the plaintext and the corresponding ciphertext. He tries to discover

the correlation between the two, e.g. by finding the key to encrypt the plaintext.

- *Ciphertext-only* With this type of attack, the adversary only has access to the ciphertext. His goal is to find the corresponding plaintext or key, e.g. a hashed password.

Since password hashing schemes are based on one-way hash functions, exhaustive search on those schemes is only possible in combination with ciphertext only attacks[1].

If one assumes that the probability of the exhaustive search attack succeeding on the first try is exactly equal to the probability that it would succeed on the 2nd, or n-th attempt, the law of averages then states that the best and unbiased estimate is the mid-point of the series. In here, 'best' is defined as having the smallest sum of squared deviations of the difference between the successful attempt and the half-way point[26]. So, if an exhaustive search has a complexity of 2^n in time, a successful try is expected at 2^{n-1}.

2.1.2 Birthday attack

This type of attack is based on the birthday problem in probability theory. This theory states the probability that in a set of randomly chosen people a pair of them will have the same birthday. Against most people's expectation, the probability that 2 people, out of a set of 23, having their birthday on the same date is close to 50 %. With a birthday attack, an adversary randomly generates output of a given cryptographic function until two inputs map to the same output. Let $n(p, N)$ be the smallest number of values we have to choose, such that the probability for finding a collision is at least p (with a total of N possibilities). Then $n(p, N)$ can be approximated by [60]:

$$n(p, N) \approx \sqrt{2N \ln \frac{1}{1-p}}. \tag{2.1}$$

Now let $Q(N)$ be the expected number of values an adversary has to choose before finding the first collision. This number can be approximated by [60, 6]:

$$Q(N) \approx \sqrt{\frac{\pi}{2} N}. \tag{2.2}$$

Since cryptographic hash functions map an arbitrary input to a fixed size output, collisions always occur (due to the pigeonhole principle). A good cryptographic hash function H has N output values that are all equally likely. This makes birthday attacks more efficient, since finding a collision only takes $1.25\sqrt{N}$ evaluations

[1] Actually, the original UNIX crypt function uses DES to 'encrypt' the user password, but with the zero's as the plaintext and the user password as the key.

of the hash function H (where N should be sufficiently large). For example, MD5 has an output of 128 bits, so $N = 2^{128}$. If MD5 is perfect collision resistant, a collision will then be found after approximately 2^{64} tries. However, MD5 has been widely investigated on its cryptographic strength. For example, Feng and Xie [73, 74] reported that collisions can be found in approximately 2^{21} time.

2.1.3 Time-memory trade-off attack

The time that is needed to break a cryptographic system can be reduced by using a technique called *time-memory trade-off*. The technique was introduced by Hellman [31] in 1980 and it is based on the fact that exhaustive search needs a lot of time or computing power to complete. When the same attack has to be carried out more than once, it may be beneficial to execute the exhaustive search in advance and store the results in memory. If this precomputation is done, the search can be carried out almost instantly. However, storing all results in memory is not very practical since most attacks need a unfeasible amount of memory. Therefore, memory is traded against time. Consider a cryptographic system with N possible keys. Time-memory trade-off can find a key in $N^{2/3}$ operations using only $N^{2/3}$ words of memory. However, this is a probabilistic method and therefore the success of the attack depends on the time and memory allocated for the cryptanalysis. The original technique by Hellman has some constraints on its reduction function and therefore collisions in the table storage could occur. Oechslin [50] proposed a new time-memory trade-off technique (called rainbow tables) with multiple reduction functions, which significantly reduce the number of collisions and so reduce the number of calculations.

Time-memory trade-off is especially effective against the Windows Lan Manager password hashing scheme [50] and the traditional Unix hashing scheme, which is based on DES [44]. However, password hashing schemes that use a sufficiently large salt offer good protection against time-memory trade-off attacks. Since a salt differs for every user, multiple rainbow tables have to be created. If a password hashing scheme uses a n-bit salt, 2^n rainbow tables have to be created, which is unfeasible for large n.

Another way to prevent time-memory trade-off attacks against a password hashing scheme is to use a variable k number of hash iterations. The number k is then stored together with the salt (both publicly known) and every time the user authenticates himself to the system, the password hashing scheme is called with k iterations of the hash function. This way, time-memory trade-off becomes unfeasible since the adversary should precompute the rainbow tables for all possible k.

2.2 Performance enhancements

The previously described attacks can be enhanced by other techniques and hardware, which will be described this section.

2.2.1 Bit slicing

While the origin of the bit slicing technique dates from the early seventies, it became popular again (albeit in another form) by Biham's 1997 paper [11], in which he describes how bit slicing can increase the performance of cryptographic applications like DES. Bitslicing, as described by Biham, views an n-bit processor as a computer with n one-bit processors. The algorithm is then broken down to AND, OR, NOT, and XOR gates, and these gates are implemented as machine instructions. For example, if we consider a 64-bit machine, this results in the (cryptographic) function being executed 64 times in parallel. The bit-sliced DES algorithm yields a significant speedup over the original implementation, which supports exhaustive search attacks.

In more recent times, Kasper and Schwabe[33] showed that a bit sliced implementation of AES encryption is up to 25% faster than previous implementations, while simultaneously offering protection against timing attacks. This makes it valuable to consider when performing exhaustive search attacks on MD5 and MD5-crypt. Unfortunately, no literature on how to implement a bit sliced version of MD5 of MD5-crypt could be found. The authors of 'John the Ripper' [19] made a proof of concept for a bitsliced implementation of MD5, but no notable performance increase was gained. The authors state that bitsliced implementations for MD5 and SHA-1 are possible, but they are only more efficient than traditional ones if near-future CPU's have wider registers (over 128 bit), larger L1 caches, and higher instruction issue rates. However, current general-purpose CPUs that satisfy these criteria happen to support parallel operations on 32-bit elements within the 128-bit vector registers (like the SSE instruction set), which is both more straightforward and more efficient than a pure bitsliced version of these hash functions.

2.2.2 Special purpose hardware

In contrast to generic purpose machines, special purpose hardware is solely build for the application of one (cryptographic) function. For example Kedem and Ishihara[35] used a Single Instruction Multiple Data (SIMD) machine to 'crack' the traditional Unix hashing scheme. With the introduction of new architectures, new possibilities arise with regard to the exhaustive search attacks on cryptographic systems. The following architectures are most common:

- *Cell* The cell processor is a microprocessor architecture jointly developed by Sony, IBM, and Toshiba. It was specially developed for the Sony Playstation 3, since the architecture combines a general purpose processor of modest performance with streamlined co-processing elements, which greatly accelerate multimedia and vector processing applications. It turned out that this architecture could be used for finding MD5-collisions as well. The HashClash project used 200 Cell processors to find such collisions in limited time, which enabled the creation of rogue SSL certificates. [67, 66]

- *FPGA* A Field-Programmable Gate Array is an integrated circuit designed to be configured by a customer after manufacturing. They contain programmable logic components (logic blocks), and a hierarchy of reconfigurable interconnects that allow those components to communicate. Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGA's, the logic blocks also include memory elements. Since cryptographic functions can easily be simplified to logic operators, FPGA's can be used for implementing them[22]. For example, Mentens et al. [44] showed that the traditional Unix password hashing scheme can be 'cracked' within an hour on a FPGA if all possible salts are precomputed.

- *GPU* GPU's are an upcoming platform for applications of cryptographic functions, since more graphics cards support integer arithmetic and general programming application programming interfaces (API's). This research uses GPU's to speed up exhaustive search attacks on prevalent password hashing schemes.

Chapter 3

Hash functions

Since password hashing schemes rely on the cryptographic properties of the underlying hash functions, this chapter describes the design of hash functions and one hash function in particular, the Message Digest Algorithm 5 (MD5).

3.1 Introduction to hash functions and their cryptographic properties

A hash function is a mathematical function that maps an arbitrary sized input (domain) into a fixed sized output (range)[43]:

$$H : \mathbb{Z}_2^m \rightarrow \mathbb{Z}_2^n, \tag{3.1}$$

where H is the hash function, $\mathbb{Z}_2$ is equal to $\{0, 1\}$, m is the input size (in number of bits) and n is the output size (in number of bits). In most cases $m > n$ holds. This is the reason why hash functions are also called compression functions. Most hash functions are built for a particular purpose. To produce the hash, the bits of the input message are mixed by bitwise operations (rotations and shifts), modular additions and compression functions. These mixing techniques are then iterated in order to ensure high complexity and pseudo-randomness of the output.

The main properties of a good hash function are:

Uniformly distributed A perfect hash function should produce unique output for every unique input. However, due the fact that in most cases the domain is greater than the range and according to the pigeon hole principle[1], some different inputs will map to the same output. These situations are called *collisions*. To minimize the likelihood of collisions, the output of a good hash function should be uniformly distributed, meaning that the probability for all outputs is the same: $1/N$, where N is the size of the output space. In the case that the output size is determined by n bits, the probability of all the 2^n outputs should be $1/2^n$.

Deterministic For any given input, the hash function should produce the same hash value on any given time.

Low complexity It should be easy to compute a hash value for any given message. An efficient hash function should have a linear complexity of $O(m)$.

The idea for hash functions was originally conceived in the late 50's [37], but it is still a very active research field. Hash functions can be used for many purposes, but the most prevalent are:

- Fast table lookup. Elements of the table can be found fast if together with the element the hash value of every element is stored as an index.
- Message digest. A hash function is used to compare two large bit streams by calculating the hash value of both the streams. If the hash values are different, the input streams must be different. If the hash values are the same,

[1] The pigeonhole principle states that if n items are put into m pigeonholes with $n > m$, then at least one pigeonhole must contain more than one item.

then one could say that the input streams are the same, with probability P. The hash function is considered good if P is high.

- Encryption. Some encryption algorithms use hash functions to produce ciphertext that cannot be mapped back to the input. Hash functions that are used in this context are called mixing functions.

- Digital signatures. Instead of signing a whole document, it is more efficient to sign only the hash of the document. Upon verification of the signature, the hash of the document is used to ensure document integrity.

- Authentication. Hash-based Message Authentication is a specific construction for calculating a message authentication code (MAC) involving a cryptographic hash function in combination with a secret key. This way, end users who share the key can determine if the message has been tampered with.

- Password storage. Since hash functions deterministically map input to uniformly distributed output and the fact that they are hard to reverse, they are considered appropriate functions for password storage.

Depending on the application of the hash function generally two kinds of hash functions can be distinguished: cryptographic and non-cryptographic hash functions. Regarding the purpose of this research, we will only review the cryptographic hash functions.

Cryptographic hash functions are used in a variety of security applications, such as digital signatures, message authentication codes and other forms of authentication. If the security of such a function is broken, the parent security application may be broken too. Therefore extra security properties are required to make general hash functions suitable for cryptographic use[43].

Pre-image resistance This concept is related to that of a one-way function. Functions that lack this property are vulnerable to so-called pre-image attacks. To avoid those, two subtypes of pre-image resistance can be defined:

1. *First pre-image resistance* Given a hash function H and a hash h, it should be hard to find any message m such that $h = H(m)$.
2. *Second pre-image resistance* Given a hash function H and a message m_1 it should be hard to find another message m_2 (where $m_2 \neq m_1$) such that $H(m_1) = H(m_2)$. This property is sometimes referred to as *weak collision resistance.*

Collision resistance Given a hash function H, it should be hard to find two different messages m_1 and m_2 such that $H(m_1) = H(m_2)$. Such a pair is

called a cryptographic hash collision. This property is sometimes referred to as *strong collision resistance*.

A brute force attack can find a first or second pre-image for some hash function with an output size of n bits in approximately 2^n hash operations. In addition to this, a brute force attack to generate a collision can be mounted in $2^{\frac{n}{2}}$, due to the birthday paradox (which is be described in Chapter 2.1.2).

In the former description, *hard to find* implies that there should be no other attack feasible than an exhaustive search attack, and the exhaustive search attack should not be feasible for as long as the security of the system is considered important. This concept is related to the bit length of the key. For example, if an adversary needs thousands of expensive computers and years of execution time in order to break a key that is not used anymore at the time of recovery, then this attack is not considered as a break. A cryptographic hash function is considered broken if there exists an attack that requires less operations and time to execute than an exhaustive search approach would require.

3.2 The MD5 hash function

The Message Digest algorithm 5, or briefly MD5, is a widely used cryptographic hash function. It was first proposed by Rivest in 1992. The RFC1321 document [56] describes the hash function and its applications in cryptography. The algorithm takes as input a message of arbitrary length and produces as output a 128-bit 'fingerprint' or 'message digest' of the input. For example, consider the 8-byte input 'Computer', which can be represented by $436f6d7075746572_{16}$. It will be hashed to the following digest value (represented in 32 hexadecimals):

$$MD5(Computer) = 181900dad960beccb34f53c4e0ff4647_{16}$$

The MD5 algorithm is an extension of the MD-4 message digest algorithm. Various modifications have been made in order to make MD5 less vulnerable to successful cryptanalytic attacks. MD5 is slightly slower than MD4, but is more 'conservative' in design and security. However, the author already stated in 1992 that he could not guarantee that it is computationally infeasible to mount a pre-image attack or produce two messages with the same digest. The following subsections will describe the algorithm and its design principles.

3.2.1 Merkle-Damgard hash functions

In order to ensure that a cryptographic hash function is able to process an arbitrary-length message into a fixed-length output, the MD5 algorithm is based on the hash function construction by Merkle and Damgard [45, 17]. They showed that this can be achieved by splitting the input into a series of equal-sized blocks,

and operating on them in sequence using a one-way compression function that processes a fixed-length input into a shorter, fixed-length output. The compression function can either be specially designed for hashing or be built from a block cipher. Figure 3.1 shows an overview of the Merkle-Damgard construction. In the figure, the message m is divided into equal size message blocks $x_1||\ldots||x_n$, the one-way compression function is denoted as H_k and x_0 denotes the initial value with the same size as the message blocks $x_1 \ldots x_n$ (x_0 is implementation or algorithm specific and is represented by an initialization vector). The algorithm then starts by taking x_0 and x_1 as input to the compression function H_k and outputs an intermediate value of the same size of x_0 and x_1. Then for each message block x_i, the compression function H_k takes the result so far, combines it with the message block, and produces an intermediate result. The last message block x_n contains bits representing the length of the entire message m, optionally padded to a fixed length output.

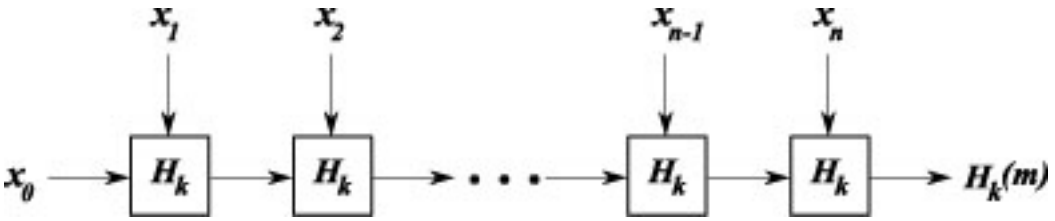

Figure 3.1: Overview of the Merkle-Damgard contruction.

Merkle and Damgard showed that their construction is sound: *if the compression function is collision-resistant, then the hash function will also be collision resistant.* An important aspect of their proof is the fact that messages should be padded with data that encodes the length of the original input message. This is called length padding or Merkle-Damgȧrd strengthening.

3.2.2 Notation

To describe the algorithm, the following notation is used:

- A *word* denotes a *32 bit* quantity.
- A *byte* denotes a *8 bit* quantity.
- MD5 works on four words of *unsigned integers*. The bits are numbered from 0 (least significant bit) up to 31 (most significant bit).
- Integers are represented in hexadecimal format (with subscript$_{16}$) or in binary format (with subscript$_2$). For example, the number 42 is represented by $2A_{16}$ and 00101010_2.

- Let $X + Y$ and $X - Y$ denote addition and subtraction (modulo 2^{32}) of two words X and Y respectively.
- Let $X \wedge Y$ denote the bitwise AND of words X, Y or bits X, Y.
- Let $X \vee Y$ denote the bitwise OR of words X, Y or bits X, Y.
- Let $X \oplus Y$ denote the bitwise XOR of words X, Y or bits X, Y.
- Let $\overline{X}$ denote the bitwise complement of the word or bit X.
- Let $X[i]$ denote the i-th bit of the word X.
- Let $X <<< n$ and $X >>> n$ denote the cyclic left and cyclic right shift of word X by n positions. For example $(01001101_2 <<< 2) = (01001101_2 >>> 6) = 00110101_2$.
- Let $X||Y$ denote the concatenation of X and Y.

3.2.3 MD5 algorithm description

To compute the MD5 message digest of an input M, with length b, the following stages should be completed (as described in [56]).

1. *Padding* To make sure that the length of the message is congruent to 448 modulo 512, the message is extended with a single '1' bit followed by a number of '0' bits. To make the length of the message an exact multiple of 512, 64 bits representing b are added at the end. In the unlikely event that b is greater than 2^{64}, then only the low-order 64 bits of b are used. For example, let the message M be the single character *'a'*, then the representation of the message in bits will be: *01100001* (*'a'* has number 97 in the ASCII-table). M contains 8 bits, which means that 440 (448-8) bits will be added. The first added bit will be a *'1'* and the 439 others will be *'0'* bits. Because the length of the message is 8, the word representing the length will be *00001000* and is appended directly after the message (The 64 bits representing the message length are appended as two 32-bit words and appended low-order word first). Altogether, the message is represented by the following 512 bits:

$$0110000110000000\ldots00000000^{54}\ldots 00000000000000000000000000001000\ldots00000000^{4}. \quad (3.2)$$

   Padding is always performed, even in the length of the message is already congruent to 448 modulo 512.

2. *Partitioning* At this point the resulting message has a length that is an exact multiple of 512 bits. Equivalently, this message has a length that is an exact multiple of 16 words. Let $M_0 \dots M_{N-1}$ denote the words of the resulting message, where N is a multiple of 16.

3. *Initialization* In this phase, four words (A_0, B_0, C_0, D_0) are initialized to the following hexadecimal values:

$$\begin{array}{lcllll} A_0 & = & 01 & 23 & 45 & 67 \\ B_0 & = & 89 & ab & cd & ef \\ C_0 & = & fe & dc & ba & 98 \\ D_0 & = & 76 & 54 & 32 & 10 \end{array}$$

4. *Processing* MD5 goes through N states IHV_i, for $0 \leq i < N$, called the intermediate hash values. Each IHV_i consists of four 32-bit words A_i, B_i, C_i, D_i. If $i = 0$ these are initialized to the four words described above:

$$IHV_0 = (A_0, B_0, C_0, D_0).$$

   For $i = 1 \dots N$, the intermediate hash value IHV_i is computed using the MD5 compression function:

$$IHV_i \ = \ \text{MD5Compress}(IHV_{i-1}, M_i). \tag{3.3}$$

   The compression function of MD5 (MD5Compress()) will be described in detail below.

5. *Output* The resulting hash value is the last intermediate hash value IHV_N, expressed as the concatenation of the sequence of bytes, each usually shown in 2 digit hexadecimal representation, given by the four words A_N, B_N, C_N, D_N.

3.2.4 The MD5 compression function

This section will describe the MD5 compression function based on the RFC1321 [56] and the work of Stevens [65]. The input for the compression function MD5Compress(IHV_{i-1},M_i) is given by an intermediate hash value $IHV_{i-1} = (A, B, C, D)$ and a 512-bit message block M_i. There are 64 steps (numbered 0 up to 63), split into four consecutive rounds of 16 steps each. Each step uses a modular addition, a left rotation, and an application of a non-linear function. Depending on the step t, an Addition Constant AC_t and a Rotation Constant RC_t are defined as follows:

$$AC_t \ = \ \left\lfloor 2^{32} |sin(t+1)| \right\rfloor, 0 \leq t < 64,$$

$$(RC_t, RC_{t+1}, RC_{t+2}, RC_{t+3}) = \begin{cases} (7, 12, 17, 22) & \text{for } t = 0, 4, 8, 12, \\ (5, 9, 14, 20) & \text{for } t = 16, 20, 24, 28, \\ (4, 11, 16, 23) & \text{for } t = 32, 36, 40, 44, \\ (6, 10, 15, 21) & \text{for } t = 48, 52, 56, 60. \end{cases}$$

The non-linear function f_t depends on the round:

$$f_t(X, Y, Z) = \begin{cases} F(X, Y, Z) = (X \wedge Y) \wedge (\overline{X} \wedge Z) & \text{for } 0 \leq t < 16, \\ G(X, Y, Z) = (Z \wedge X) \wedge (\overline{Z} \wedge Y) & \text{for } 16 \leq t < 32, \\ H(X, Y, Z) = X \oplus Y \oplus Z & \text{for } 32 \leq t < 48, \\ I(X, Y, Z) = Y \oplus (X \vee \overline{Z}) & \text{for } 48 \leq t < 64. \end{cases}$$

The message block M_i is partitioned into sixteen consecutive 32-bit words $m_0, \ldots, m_{15}$ and expanded to 64 words $(Wt)_{t=0}^{63}$ for each step using the following relations:

$$W_t = \begin{cases} m_t & \text{for } 0 \leq t < 16, \\ m_{(1+5t) mod 16} & \text{for } 16 \leq t < 32, \\ m_{(5+3t) mod 16} & \text{for } 32 \leq t < 48, \\ m_{(7t) mod 16} & \text{for } 48 \leq t < 64. \end{cases}$$

For $t = 0, \ldots, 63$, the compression function algorithm maintains four 32-bit integers $(Q_t, Q_{t-1}, Q_{t-2}, Q_{t-3})$ to store the intermediate state. These are initialized as $(Q_0, Q_{-1}, Q_{-2}, Q_{-3}) = (B, C, D, A)$ and updated as follows:

$$Q_{t+1} = Q_t + ((Q_{t-3} + f_t(Q_t, Q_{t-1}, Q_{t-2}) + W_t + AC_t) <<< RC_t) \text{ for } 0 \leq t < 64.$$

After this step the resulting state words are added to the intermediate hash value. The MD5 compression function can finally be described as:

$$\text{MD5Compress}(IHV_{i-1}, M_i) = (A + Q_{61}, B + Q_{64}, C + Q_{63}, D + Q_{62}).$$

Chapter 4

Password hashing schemes

Passwords are the most common form of user authentication in computer systems. To secure against unauthorized disclosure of users' credentials, passwords are usually protected by hashing them, e.g. by using one of the hash functions described earlier. The hashed passwords are then stored in a password table. During the log in process, the user-provided password is hashed again using the same hash function and compared with the stored hashed password to authorize user access. There are two main types of attack that can be mounted on such a authentication system:

1. *On line guessing attack* With this attack, the only way for the adversary to verify whether a password is correct, is by interacting with the login server. This kind of attack can be countered by techniques as account locking and delayed response. This research will not focus on this type of attack. An evaluation and more information about the countermeasures can be found in [52].

2. *Off line guessing attack* With this attack, it is presumed that the adversary has access to the files or database where the hashed passwords are stored. He can mount all kinds of attacks against the hashed passwords, due to the fact that they are now in his 'property'. This research will not focus on how to prevent an adversary to access password files, but it will focus on the schemes that hash and store them. Those schemes try to provide security even if an adversary has access to the password files.

This chapter focuses on the design, security properties and attacker models of prevalent password hashing schemes.

4.1 Introduction to password hashing schemes

Password hashing schemes are a set of algorithms, including one or more hashing functions, that try to protect the password information of users on a given system. Before the plaintext password is hashed, it is usually transformed and complemented by a *salt*. Because a password hashing scheme has to be a static one-way function, a salt is random data, which may be public, that is hashed together with the password in order to make it unlikely that identical passwords are hashed to the same ciphertext. Another advantage of a salt is the fact that it makes dictionary attacks more difficult, which will be described in Chapter 4.4.

A password hashing scheme is not the same as a hash function. Typically, it is build on top of a cryptographic hash function to ensure that passwords are stored securely. Most password hashing schemes are defined as follows:

$$PHS : \mathbb{Z}_2^m \times \mathbb{Z}_2^s \rightarrow \mathbb{Z}_2^n, \tag{4.1}$$

where PHS is the password hashing scheme, $\mathbb{Z}_2$ equals $\{0, 1\}$, m is the password size (in number of bits), s is the salt size (in number of bits) and n is the output size (in number of bits). Typically, $m + s < n$, because the length of the salt and the plaintext password do not exceed the output size. Password hashing schemes can restrict the input size, because if $m + s > n$, it will not add extra security. However, users should be able to remember their passwords somehow and therefore pick m small. In most schemes, the output of the function is not merely the hash of the password: it is a text string which also stores the salt and identifies the hash algorithm used. See Figure 4.1 for a graphical overview of a common password hashing scheme. This figure shows the black box overview

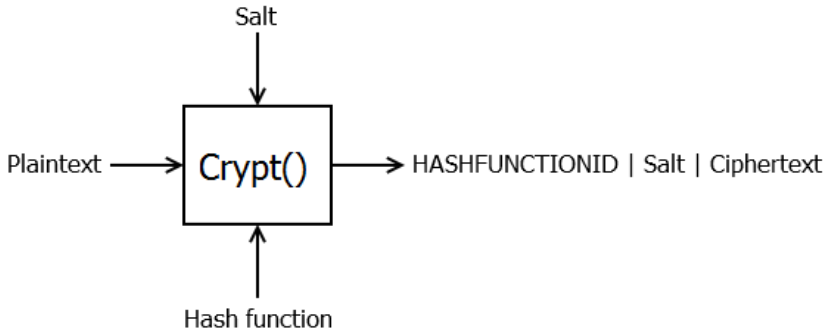

Figure 4.1: An black box overview of the UNIX password hashing scheme crypt().

of the UNIX password hashing scheme called `crypt()`. If a new user has to set his password, the system takes the plaintext password, hashes it together with the salt according to the specified hash function scheme and stores it in the `/etc/shadow` file in the following way:

```
$HASHFUNCTIONID$SALT$CIPHERTEXT
```

The contemporary `crypt()` function is compatible with the cryptographic hash functions shown in Table 4.1.

Identifier	Scheme	Hash function	Salt length (in # of characters)	Salt length (in bits)
1	MD5-crypt	MD5	8	64
2a	B-crypt	Blowfish	8	64
md5	Sun MD5	MD5	8	64
5	SHA-crypt	SHA-256	16	128
6	SHA-crypt	SHA-512	16	128

Table 4.1: Schemes shown in this table are built around the hash function they are named after.

When a user has to authenticate himself to the system again, he has to type in his password which will be hashed in the same way as when he had to set his password together with the stored salt. If the newly calculated ciphertext matches the one stored in the `/etc/shadow` file, authentication is successful.

4.2 The need for password hashing schemes

To understand the need for password hashing schemes, it it is important to distinguish between two types of off-line attacks that adversaries can mount against the storage of passwords (it is assumed that the adversary already has access to the storage files):

1. *Depth-search* With this kind of attack the adversary wants to find the password of one specific user, which is often a user with high level access, like a superuser or administrator. The adversary has to focus on this one hash. This this kind of attack can be executed with the *complete attack* strategy[71], where the adversary is willing to take all the required computational effort to find the password.

2. *Broad-search* With this kind of attack the adversary wants to find just one plaintext password out of a storage with more hashed passwords, because one plaintext password can be sufficient to compromise the security of the system. This kind of attack can be executed with the *incomplete attack* strategy[71], where the adversary tries a number of N guesses for each hashed password in the storage file, thereby finding the password only with a certain probability.

Now let p be the probability that a user has chosen a weak password and n be number of users on the targeted system. Then it is easy to see that a broad-search has a higher probability (np) of success than a depth-search (only p), given the assumption that the password strengths are identical. Table 4.2 shows some of the characteristics of real (cracked) password datasets.

Dataset	Number of passwords	Average password length	% with special chars	% only lowercase ASCII	% in dictionary
phpbb.com	184 000	7.5	2	41	29
rockyou.com	14 000 000	8.7	7	25	1

Table 4.2: Characteristics of real world password datasets.

The datasets in this table were published by skullsecurity.org[1] and contain the passwords that people use when they are not restricted by a password policy. The phpbb.com database was cracked in January 2009 and the rockyou.com database in December 2009. While the average password length for both datasets looks good (with a password length > 7), 29% of the passwords in the *phpbb* dataset could be found in an extended English dictionary[2]. A fast CPU can iterate through such a dictionary in less then a second. The statistics also show that few people use special characters in their passwords and on average more than 30 % of the people only use lowercase ASCII characters from the set $\{a, \ldots, z\}$. Regarding the average length of 8 characters, this means that only $26^8 \approx 2^{37}$ combinations are used, which which is feasible for an exhaustive search attack.

4.3 Attack strategies

In addition to the previously mentioned attack types, there is another dimension: the attack strategy. The three main strategies are defined as:

1. *Exhaustive search attack* With this sort of attack, the adversary iterates over all possible input combinations. Let M be the domain for the password hashing scheme PHS, s_1 an arbitrary salt, $m_1 \in M$ be the password the user wants to store and h_1 the hashed form of the password by applying $PHS(m_1, s_1)$. Now let M (the domain of the hashing scheme) be uniformly distributed and let $|M| = N$, such that $\forall m \in M$ the probability p_m of $PHS(m, s_1) = h_1$ is $\frac{1}{N}$. To find the password m_1 from h_1, the adversary

[1] See http://www.skullsecurity.org/wiki/index.php/Passwords for more details.
[2] An English dictionary with almost 400.000 entries, including conjungations and names.

tries all m in the domain M. If for some m, $PHS(m, s_1) = h_1$, the adversary knows then that $m_1 = m$. The probability P of a successful attack can be calculated by:

$$P = \sum_{m \in M} p_m = N * \frac{1}{N} = 1 \tag{4.2}$$

So an exhaustive search attack will always find the password. However, the time to find it is limited by N. If the input size is very large, it is not feasible to iterate over all the possibilities anymore. For example if the maximum input length is 9 characters and if all 94 printable ASCII characters are admitted, N will be:

$$N = \sum_{k=0}^{9} 94^k = 579\ 156\ 036\ 661\ 182\ 475 \approx 2^{59} \tag{4.3}$$

If one has a computer that can review 1 billion passwords per second, it will still take more than 18 years to go through the whole search space and the expected time to crack one password will be half that time. However, users tend to pick their passwords with less randomness than randomly chosen passwords.

2. *Dictionary attack* In order to remember them more easily, humans tend to pick common words as a password[51, 75]. This enables adversaries to use common dictionaries to find passwords. A typical dictionary has less entries than the number of possibilities one has to iterate with an exhaustive search attack. This makes dictionary attacks faster than exhaustive search attacks. However, under the assumption that users select uniformly distributed passwords, the probability that an adversary eventually will find the password is less than with an exhaustive search attack. Let E be the set of entries in a dictionary and $|E| = K$, where $E \subseteq M$. The probability P of a successful attack can be calculated by:

$$P = \sum_{m \in E} p_m = K * \frac{1}{N} = \frac{K}{N} \tag{4.4}$$

Typically $K << N$, which means that the probability of a successful attack is small. However, in real world situations this is not the case and a dictionary attack therefore has a higher probability of a successful attack. (See Table 4.2).

3. *Fingerprinting attack* This kind of attacks exploits the fact that people have to remember their password somehow. When constructing an easy to remember yet still hard to crack password, people use various kind of strategies. These include techniques as generating mnemonic phrases, obfuscation or the use of patterns. An example of a mnemonic phrase

could be: '**R**obert **R**odriguez **m**ade **3** **f**ilms: **M**achete, **G**rindhouse and **E**l **m**ariachi'. The derived password would then be: 'RRm3fM&G+Em', which is easy to remember but very hard to guess. Obfuscation can be achieved by replacing some of the original characters in a password by characters that look the same, like how the English word 'password' can be transformed to 'p@ssw0rd'. Examples of simple patterns that people could use to strengthen their password are: '1!qQ', '65$#' and 'rdcvgt' (characters close to each other on the keyboard). However, adversaries have become more sophisticated in how to deal with these techniques. Based on earlier cracked password databases, they derived statistics about length, character frequency and pattern usage. Based upon the retrieved statistics, specific search spaces are constructed and exhaustive search is then applied.[3]

All these attacks can be combined in order to achieve the best trade-off between input coverage and computation time.

To avoid such attacks on passwords, it is necessary that the input to the password hashing scheme is uniformly distributed. This implies that a strong password should have as much as randomness as possible. A good way to measure the strength of a password - or the randomness of information in general - is *entropy*. Shannon proposed in his paper of 1948 [63] that entropy could be used to quantify, in the sense of an expected value, the information contained in a message, usually in units such as bits. Equivalently, the Shannon entropy is a measure of the average information content one is missing when he does not know the value of the random variable, i.e. *the measure of uncertainty*. Let X be a discrete random variable with possible values $\{x_1, \ldots, x_n\}$, then the Shannon entropy $H(X)$ can be defined by[16]:

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log_b p(x_i) \tag{4.5}$$

where b is the base of the logarithm used and is $p(x_i)$ is the probability mass function of outcome x_i. For example, the entropy in a fair coin flip is exactly 1 bit because the probability $p(heads) = p(tail) = 0.5$. Then $H(X) = -p(heads)\log_2 p(heads) - p(tail)\log_2 p(tail) = 1$ bit.

The entropy for an ideal password can now be calculated by:

$$u = \log_2(n), \tag{4.6}$$

where u is the entropy in bits and n the number of possibilities. For example if the maximum size of the input space is 10 characters and if all 94 printable

[3]The authors of the article 'Password Fingerprinting' use a real-world database of passwords that was leaked to prove that their technique works on all levels of passwords including easy passwords, medium strength passwords, and a bit more difficult passwords. (See http://www.question-defense.com/2010/08/15/automated-password-cracking-use-oclhashcat-to-launch-a-fingerprint-attack)

ASCII characters are admitted, the entropy will be $\log_2(94^{10}) \approx 66$ bits. However, Shannon also conducted some social experiments to show the entropy of the English language [62]. It turned out that standard English has an entropy between 0.6 to 1.3 bits per letter. For an ideal password that is encoded with ASCII (six bits per character), one would like to have an entropy of six bits per character, which implies maximal randomness. Passwords do not always contain standard English words, but the United States National Institute of Standards and Technology (NIST)[13] calculated that a user-chosen password based on all 94 printable ASCII characters with a length of 10 characters has an entropy of 21 bits. Using modern day computing power, all 2^{21} possibilities could be searched in limited time.

Zviran and Haga[78] already showed in 1999 by means of an empirical study that users are not eager to change their habits regarding the creation of strong passwords. One of the causes, as identified by Katz et al.[34], is that users are asked to generate, memorize, and keep secret a growing number of passwords as they join new password protected services over time. Since the mid nineties this trend has been recognized as both a nuisance and a security risk. They have shown that typical users can be trained to select and remember a single secure password, but multiplying this dozens or hundreds of times is sure to push the physiological limitations of human memory. Halderman et al.[27] proposed a solution by securing all the passwords of a user by 'masterpassword', but this only solves the problem at the user-side of the authentication mechanism. At the server-side, the low entropy passwords are still stored in their original form.

4.4 Properties of a good password hashing scheme

If all users just pick high entropy passwords, there is no need for password hashing schemes, since an application of a simple cryptographic hash function like MD5 would be enough to secure their passwords (with the assumption that MD5 is pre-image resistant). However the studies by Shannon, Zviran, Katz and the NIST show that it is hard for users to create passwords with high entropy, so the storage of hashed passwords has become very important. To review the quality of such a scheme, it it necessary to define the properties of a good password hashing scheme first. In 1979, Morris et al.[47] already proposed ways to protect stored passwords. Most of their principles are still valid today.

- *Correct use of salts* As stated in Section 4.1, salts are necessary to produce a secure output of a password hashing scheme. Let H be a hash function, m_1 be a plaintext password that user 1 wants to hash and s the salt that is concatenated with password before the hash function is applied. Now consider that user 2 has a password m_2 that is the same as user 1, so $m_1 = m_2$. If we apply the hash function H without a salt, $H(m_1) = H(m_2)$.

An adversary that is in possession of the file with hashed passwords can easily see that user 1 and 2 have the same password. If we concatenate the password with the per-user different salt, the hashed passwords will not be same anymore. Let s_1 and s_2 (where $s_1 \neq s_2$) be the salts of user 1 and 2 respectively. If we now concatenate their passwords with their salts, $H(m_1||s_1) \neq H(m_2||s_2)$ even if $m_1 = m_2$ (and given that the collision probability is negligible). Another advantage of using a salt is the fact that *precomputation attacks* are not feasible anymore. With a precomputation attack the adversary computes n possible inputs (e.g. common passwords) to the hash scheme and stores all the n inputs and their hashed outputs $H(n)$ in a database. Now, if the adversary encounters a new hash he can find the plain text by a simple table lookup. Precomputation can be very useful for a dictionary attack and the decrease in the cost of contemporary mass storage has made it practical for fairly large dictionaries. Oechslin[50], as described in Chapter 2.1.3, constructed a more advanced precomputation method, by applying a time-memory trade-off in combination with hash chains. This speeds up the cracking process even more.

A good salt can prevent against both weaknesses. For example, if we add a salt of 1 bit to the input of a hash function H, the output of $H(m||1)$ and $H(m||0)$ should be totally different due the avalanche effect in the used hash function. This should solve the problem that two users with the same password have the same hashed password. However, a salt of 1 bit can still be precomputed. The original Unix password hashing scheme used triple DES in combination with a 12 bit salt. This extends the output space of a single password with 2^{12} possibilities. Mentens et al. [44] showed that even a 12 bit salt could be precomputed with a time-memory trade-off attack. Therefore, newer hashing schemes such as MD5-crypt or SHA-crypt use salts up to 8 or 16 characters respectively. Since the salt input to those schemes requires a base64 input (represented by 6 bits), this implies that the entropy is 48 and 96 bit respectively. There are no time-memory trade-off attacks on these systems reported (yet).

- *Slow hashing* Most common cryptographic hash functions are designed to quickly evaluate the one way function, in order to hash large input (See Chapter 3.1). While this could be an advantage for the systems that need to evaluate hash functions quickly, it also makes exhaustive search attacks on the input feasible. To slow down these kind of attacks, one approach is to use hash functions which do not yield efficient implementations. However, people are not likely to design such hash functions, because the usability will decrease. Instead it is possible to modify existing password hashing schemes to yield expensive hash functions, without voiding their security warranty. One approach to this is called *key-stretching*, which will be described in

Chapter 4.4.1.

- *Avoid pipelined implementations* As described by Provos et al. [55], a password hashing scheme should not lend itself to any kind of pipelined hardware implementation. It should therefore permit relatively little speed-up from any kind of computation. For example, calculating the hash of 1000 passwords with the same salt and calculating the hash of one password under 1000 salts should both cost approximately 1000 times more computation time than calculating the hash of a single password.

- *Enforcement of complex passwords* This is not really a property of a good password hashing scheme, but rather a policy of the system that uses password authentication mechanisms, which may be implemented by the password hashing scheme. To enforce the use of complex passwords, systems should enforce password policies. However, there has to be a trade-off between the strictness of the policy and the limitations of the human memory or will to remember difficult passwords. Yan et al. [75] stated that an interesting and important challenge is finding compliance enforcement mechanisms that work well with mnemonic password choice. Proactive password checkers, which verify that a password is not part of a known weak subset of the password space, might be an effective tool. But as Yan[75] has shown, what engineers expect to work and what users actually make to work are two different things. Rigorous experimental testing of interface usability is in their view a necessary ingredient for robust secure systems. If users are not content with new password policies, they just stop using the service, fall back to their simple and weak passwords or write them down.

4.4.1 Key-stretching

To slow down exhaustive search attacks on password hashing schemes, two methods have been proposed in the literature:

- In the first approach, which was proposed by Abadi et al.[3] and Manber[42], the password is concatenated with an additional salt (also known as a 'password supplement') before it is hashed. This supplement is kept secret to the user and is typically chosen in a range $1 \ldots k$, where k should be feasible for an exhaustive search. The user has to guess this supplement by repeatedly hashing his password, salt and a random supplement in the predefined range. However, this also holds for the attacker who is guessing the password by brute force. When checking a password guess an attacker will need to perform a hash for each possible supplement until the space is searched. Thus, if the password supplement space is of size k the attacker will need to perform k hashes before he can completely eliminate one guess

as a possible password, whereas the user only has to perform *at most* k hashes.

- The second approach was proposed by Kelsey et al.[36]. They derive a new password by repeatedly iterating a hash function on the original password. Assuming that there are no shortcuts, an adversary's best attack is to apply the function to each guess himself. Therefore, if the scheme is parameterized to use k iterations, an adversary will need to compute k hash functions for each guess. For example, if N is the size of the finite set of all possible passwords up to a given length, an attacker has to compute at most Nk hash functions, whereas the user only has to compute k hash functions.

The primary drawback of the first approach is that a regular user needs a method for determining if the password supplement is correct. If this approach is integrated into a login system, some extra data can be stored on the server and the server can perform the search. However, it is hard to keep such data secret and it breaks our assumption on access to data. That is why current password hashing schemes have deployed the second approach (which is also called *key stretching*). The basic concept is that a regular hash function, $H()$, is replaced with a new hash function, $H^k()$, where $H^k()$ is computed by repeatedly applying the hash function k times. For example, $H^3(x)$ is equivalent to computing $H(H(H(x)))$. If N is the set of all possible passwords the adversary wants to test and we conjecture that for all $n \in N$ the fastest way for an adversary to compute $H(n)$ is by repeated applying the function the hash function k times, then the cost of a brute force attack will increase by a linear factor k, again if no shortcuts exist.

4.5 Attacker models for hash functions and password hashing schemes

To show the strength of password hashing schemes, it is good to define where they protect against. An adversary can have multiple goals regarding the output of password hashing schemes. In this research, the following attacker models are distinguished distinguished:

1. *Plain-text password recovery* An adversary wants to find the plaintext password of one or more users of a system, in order to use the password elsewhere (for example for authentication to the user's bank account). The adversary can achieve this by iterating over all the password possibilities with a exhaustive search attack.

2. *System authentication* An adversary wants to get access to a system or service on which the user has an account (for example if an adversary has a user account on a system but wants to become local administrator).

In addition to the exhaustive search attack mentioned in the previous he now could also apply a first pre-image and second pre-image attack, since the clear text password is not necessary to successfully authenticate the adversary. Every input to the password hashing scheme that maps to the same output as the user's output will be sufficient.

3. *Weak hash generation* If an adversary has control over the salt that is generated for a typical user, he can generate a random looking salt while the user thinks this is just a random salt. Then, with the help of precomputed rainbow tables for a small set of the random looking salts, the adversary can efficiently mount a time-memory trade-off attack to find the password. This attack makes the system as safe as it is without a salt.

By choosing a good cryptographic hash function, the designer of a password hashing scheme can prove that his scheme protects against pre- and weak collision attacks. Let PHS be a password hashing scheme that is build by hash function H^k (where H^k is computed by repeatedly applying the hash function k times), p_1 the clear text password, s_1 the public salt and h_1 the application of $PHS(p_1, s_1)$. Now the following holds:

- H is first pre-image resistant $\Rightarrow$ PHS is first pre-image resistant (it is hard to find any p_1, such that $h_1 = PHS(p_1, s_1)$).
- H is weak collision resistant[4] $\Rightarrow$ PHS is weak collision resistant (it is hard to find another input p_2 (where $p_2 \neq p_1$) such that $PHS(p_1, s_1) = PHS(p_2, s_1)$).
- H is strong collision resistant $\Rightarrow$ PHS is strong collision resistant (it is hard to find two arbitrary inputs p_1 and p_2 (where $p_2 \neq p_1$) such that $PHS(p_1, s_1) = PHS(p_2, s_1)$).

At first glance, a break of the strong collision resistance property of the hash function does not have impact on the security of the password hashing scheme, since an adversary only has access to the hash outcome h_1 and a weak collision attack requires access to the input. However, if we assume the attacker model *weak hash generation*, an adversary has access to the salt. This way, the following theoretic attack to a password hashing scheme, which uses the key-stretching technique, is possible. Consider the password hashing scheme PHS, that uses the hash function H on the input password p_1 and salt s_1 in the following way:

$$PHS(p_1, s_1) = H^k(p_1||s_1) = h_1. \tag{4.7}$$

[4]The term 'weak collision resistance' is preferred over 'second pre-image resistance' because it better states the difference between first and second pre-image resistance.

In here, h_1 is the produced output hash and k is the number of applications of the hash function H. If we assume that the salt can be arbitrary in size then the adversary can produce a salt s_1 and an evil twin s_2 such that the attack described in [66] can be applied to produce the collision $H^0(p_1||s_1) = H^0(p_2||s_2)$. Since collision occurred in the 0-th iteration of the hash function, consecutive applications of H until H^k will also produce a collision and therefore $PHS(p_1, s_1) = PHS(p_2, s_2)$ holds. Moreover, if the PHS is defined in the following way:

$$PHS(p_1, s_1) = \forall_{i<k} H^i(p_1||s_1||H^{i-1}) = h_1(\text{where} H^0 = H(p_1||s_1)), \tag{4.8}$$

the attack still holds, since the collision already occurred at H^0 and the salt, password and result of H^{i-1} are always concatenated in the same way. Considering the previous, password hashing schemes that use key-stretching in the two ways described above do not provide additional security for strong collision attacks other than the application of the first hash (H^0), given that the salt can be chosen by the attacker.

However, password hashing schemes that concatenate the password p_1, the salt chosen by the adversary s_1 and the result of the last round H^{i-1} in a pseudo-random way can provide k times additional security. In this way, the complexity of a short-chosen prefix collision attack [66] on the password hashing scheme will be k times higher than such an attack on one application of the hash function, since the specially crafted salt should work for all the k applications of the hash function.

An off line exhaustive search attack on the input of a password hashing scheme is always possible, although such an attack can not guarantee a successful search because the size of the input set can be infinite (or unfeasible to iterate over). However, as shown by Section 4.2, users tend to pick passwords with far less entropy as can be achieved with a given character set. This makes the password hashing scheme more vulnerable to dictionary and brute force attacks. To deal with these kinds of attack, most password hashing schemes use the key stretching technique to slow down the hashing process. In their main loop they apply the hash function k times, every time with different input concatenated with the output of the last round (e.g. MD5-crypt, which will be described in Chapter 6). While the key stretching technique slows down exhaustive search attacks for large k, users also have to apply the hash function k times, which leads to a paradox:

- Users expect k to be low $\Rightarrow$ fast authentication.
- Users expect k to be high $\Rightarrow$ less successful exhaustive search.

Moreover, this paradox also applies for the design of hash functions. On the one hand hash functions should be computationally fast in order to hash large amounts of data, while on the other hand hash functions should be computationally slow in to order to slow down exhaustive search.

4.5.1 Attacks to the key-stretching technique

Attacks to key-stretching algorithms (as described earlier) can typically benefit from *economies of scale*. These are optimizations that are not available or useful for a typical user of the cryptographic system, but can make large scale attacks easier. The economies of scale useful for an attacker can be divided into [36, 55]:

- *Narrow-pipe attacks* This type of attack occurs when the intermediate state in the key-stretching algorithm has less bits of state then the final output. For example, if you have an intermediate state of less then $s+t$ bits (where s is the length of the original key and t is the amount of bits stretched) an adversary can guess the intermediate state instead of calculating it and the security of the system will have less than $s + t$ bits.

- *Reuse of computed values* The key-stretching algorithm should be designed in such a way that adversaries cannot reuse intermediate computations, which can speed up the key search significantly. For example, if a key-stretching algorithm uses 2^t iterations of the hash function H, it should be very unlikely that the algorithm produces intermediate values X_1 and X_2 along with i and j $(i, j \leq 2^t)$ such that $H^i(X_1) = H^j(X_2)$, where $X_1 \neq X_2$ or $i \neq j$.

- *Special purpose hardware* As will be described in the next chapters, some key search attacks may be carried out by massively parallel computers that can be built to try all possible keys cheaply. The choice of the key-stretching algorithm should take into account such attacks and hence make such attacks more expensive to carry out for special purpose hardware without making it significantly more expensive for legitimate users that have a general purpose processor.

Chapter 5

Outline of the GPU hardware

This chapter describes the internal working of a graphics processing unit and explains why it is more useful for parallel computing than a normal central processing unit (CPU). Since the late nineties Graphics Processing Units (GPU) have been developed and improved. They have proven to be very suitable for processing parallel tasks and calculating floating point operations. It is especially the parallel design of a GPU that makes it suitable for cryptographic functions. While the advantages of GPU's in other areas (like graphical design and game-industry) have already been recognized, the cryptographic community was not able to use them due to the lack of user-friendly programming API's and the lack of support for integer arithmetic. However, GPU producers have dealt with those shortcomings. Goodman and Yang[77] have shown that contemporary GPU's can outperform high-performance Central Processing Units (CPU's) on cryptographic computations, yielding speedups of at most 60 times for the AES and DES symmetric key algorithms. Szerwinski and Güneysu showed how GPU's could be used for asymmetric cryptography [69], while Bernstein et al. showed that it is possible to reach up to 481 million modular multiplications per second on a NVIDIA GTX 295 [9], in order to break the Certicom elliptic curve cryptographic system (ECC) challenge with the use of GPU's [10]. This knowledge is very useful, since a lot of cryptographic functions were designed to be implemented on a traditional CPU. The shift of focus to GPU's provides new insights on how we should deal with (implementations of) cryptography in the future.

5.1 Hardware outline

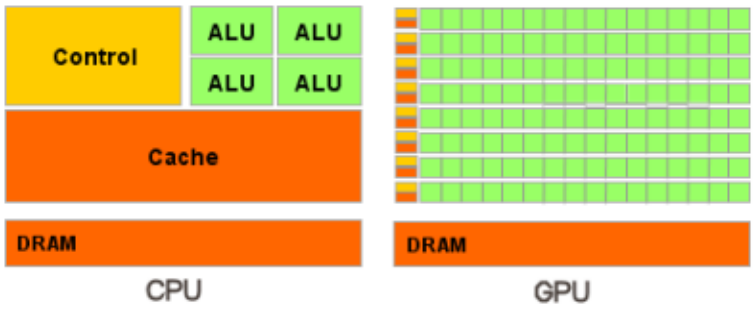

Figure 5.1: Overview of a typical CPU and GPU. Due to its design, the GPU is specialized for intensive, highly parallel computation [1].

A typical GPU, as shown in Figure 5.1, consists of the same elements as a normal CPU:

- *Arithmetic logic unit (ALU)* The digital circuit that performs arithmetic, logical and shift operations. In a GPU, this unit is also called 'stream processor' or 'thread processor'.
- *Control Unit* Dispatches instructions and keeps track of the locations in memory. In a GPU, also a thread dispatcher resides on the control unit, which controls the creation and execution of threads.
- *DRAM* Main memory, which can be refreshed dynamically.
- *Cache* A smaller, faster memory which stores copies of the data from the most frequently used main memory locations.

In addition, a GPU has multiprocessors. Every multiprocessor contains a number of thread processors and a shared memory, which is accessible by all the thread processors in the multiprocessor.

Because of the fact that a GPU is specialized for intensive, highly parallel computation, exactly what graphics rendering is about, it is designed in such a way that more transistors are devoted to data processing rather than data caching and flow control, as schematically illustrated by Figure 5.1. Moreover, a GPU is suited to address problems that can be expressed as data-parallel computations with high arithmetic intensity (the ratio of arithmetic operations to memory operations). This means that a GPU can execute the same program for each data element in parallel, which implies lower requirements for sophisticated flow control and higher arithmetic intensity (the memory access latency can now be 'hidden' with calculations by the execution of other threads instead of data caches). The processing of parallel data maps data elements to parallel processing

threads. Many applications that process large data sets can use a data-parallel programming model to speed up the computations. In 3D rendering, large sets of pixels and vertices are mapped to parallel threads. But also many algorithms outside the field of image rendering and processing are accelerated by data-parallel processing, for example applications of a cryptographic hash function on a large dataset can be done in parallel.

5.1.1 Comparing GPU and CPU design properties

The increase in CPU frequency is limited by physical restrictions and high power consumption. Their performance is often raised by increasing the number of cores: some processors may contain up to six cores. However, further growth is not expected to be coming soon [68]. As stated by Garland et al.[25], the emphasize is shifting from latency-oriented CPU's, which focus on minimizing the latency of a single sequential task, towards throughput-oriented GPU's, which focus on maximizing total throughput. Latency-oriented CPU's are designed for common applications, based on the *Multiple-Instruction Multiple-Data* (MIMD) architecture, as proposed by Flynn [23]. This means that each core works independently of the others, executing various instructions for multiple processes.

For throughput-oriented GPU's, the requirements for the graphical computational problems led to a specific parallel architecture in which multiprocessors can execute hundreds of threads concurrently. This architecture is called *Single-Instruction Multiple-Threads* (SIMT). It is based on the Single-Instruction Multiple-Data model (SIMD)[23], which describes multiple processing elements that perform the same operation on multiple data simultaneously. Accordingly, such an architecture exploits data level parallelism.

The SIMT architecture

The SIMT architecture describes a multiprocessor that can create, manage, schedule and execute threads in groups. A group of N parallel threads is called a *warp*. Every thread has its own instruction address counter and register state. Individual threads in the same warp start together at the same program address, but can execute and branch independently.

A warp executes one common instruction at a time, so full efficiency is realized when all N threads of a warp agree on their execution path. If a thread in a warp diverges via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path. When all paths are completed, the threads converge back to the original execution path. Branch divergence occurs only within a warp. Different warps on different multiprocessors execute independently regardless of whether they are executing common or disjoint code paths.

To maximize utilization, a GPU relies thus on *thread-level parallelism*. Utilization is therefore directly linked to the number of warps residing on a multiprocessor. At every instruction issue time, a warp scheduler selects a warp that is ready to execute its next instruction and issues the instruction to the active threads of the warp. The number of clock cycles it takes for a warp to be ready to execute its next instruction is called *latency* (which actually depends on the number of clock cycles it takes to issue a memory request). Full utilization is then achieved when the warp scheduler always has an instruction to issue for some warp at every clock cycle during that latency period. This is called *latency hiding*. The number of instructions required to hide a latency of L clock cycles depends on the throughput of these instructions. For example, if we assume that a multiprocessor issues one instruction per warp over 4 clock cycles and maximum throughput for all instructions is achieved, then the number of instructions to hide the latency should be $L/4$.

The SIMT architecture is related to SIMD vector organizations in the sense that a single instruction controls multiple processing elements. The major difference between the SIMD and SIMT architectures is the fact that SIMD vector organizations expose the width (and thus the number of threads that can run concurrently on one processor) to the software. SIMT instructions on the other hand specify the execution and branching behavior of a single thread. This enables programmers to write *thread-level* parallel code for independent threads as well as *data-parallel* code for coordinated threads. Actually, the programmer can even ignore the SIMT behavior; however, substantial performance improvements can be realized by taking care that the code seldom requires threads in a warp to diverge.

Threads and resources

Apart from the difference in architectural design, there exists also a major difference in the organization of threads. On a typical CPU only one or two threads can run concurrently per core, whereas on a typical GPU up to 1024 threads can run concurrently per multiprocessor (and most GPU's have several multiprocessors). For example, the Nvidia GTX 295 has 60 multiprocessors, which can yield more than 30.000 concurrent or active threads. In comparison, an Intel i7 9xx processor has only 4 cores, which can run 8 concurrent threads in total (if 'Hyper-Threading' is enabled). In addition, switching from one thread to another takes hundreds of clock cycles on a typical CPU (threads are then called 'heavyweight threads'), whereas a GPU can switch several threads per clock cycle (threads are then called 'lightweight threads'). This is due the fact that for a CPU the operating system must swap threads on and off the CPU execution channels to provide multithreading capability. Thread switching is therefore slow and expensive. On a GPU, thousands of threads are queued in warps of N threads each.

If the GPU must wait on one warp of threads, it simply begins executing work on another warp. Because separate registers are allocated to all active threads, no swapping of registers or state need to be done when switching between GPU threads. Resources stay allocated to each thread until its execution is completed.

Memory access

The CPU and GPU both have their own RAM. On a CPU, the RAM is generally equally accessible to all code (if the code accesses memory that is located within the boundaries enforced by the operating system). On a GPU, the RAM divided virtually and physically into different types, each of which has a special purpose and fulfills different needs. Due to the fact that different GPU's have different types of memory, we will not go in to detail here.

Computing capabilities

In this research we use the GPU hardware from the Nvidia Corporation. The properties and performance differ per Nvidia card, but all hardware is based on the SIMT principle. Nvidia has classified their graphic cards in order to distinguish the properties and features of each device. This classification is called *compute capability*, and determines the set of instructions supported by the device, the maximum number of threads per block and the number of registers per multiprocessor. Higher compute capability versions are supersets of lower (i.e. earlier) versions, and so they are backward compatible.

5.1.2 Speed comparison between CPU and GPU

The computing power of a hardware unit is easily measured by the number of floating point operations it can handle per second. That is why most speed comparisons of hardware use the unit GFLOPS, which stands for Giga Floating Point Operations per Second. Figure 5.2 shows the theoretical limit of computing power for contemporary hardware. Recent GPU's can handle seventeen times more floating point operations per second than the recent CPU's. And from the derivative of the GPU lines it is clear that the increase in speed has not yet converged. However, these numbers are gathered from the vendors that sell the hardware. The actual values are lower. In practice such performance will never be reached because many algorithms are limited by bandwidth, meaning that the memory subsystem cannot provide the data as fast as the arithmetic core could process it. But for a comparison in speed, these numbers will suffice.

To calculate the actual GFLOP unit for CPU's the following formula can be used: (Number of cores) * (Number of single precision floating point instructions per clock cycle) * (Clock frequency in GigaHertz). As an example consider the Intel i7 920 which has 4 cores, can calculate 4 instructions per clock cycle and

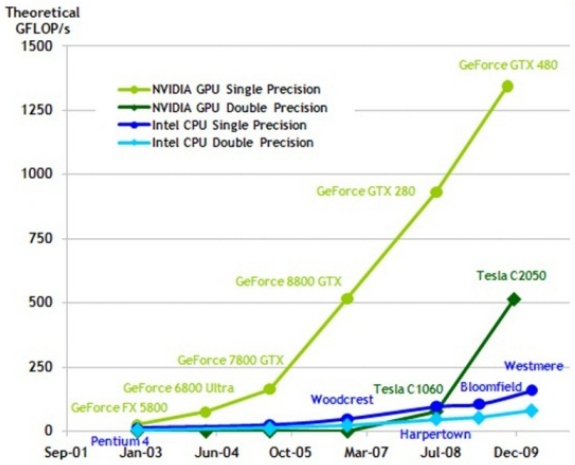

Figure 5.2: Theoretical computing power of recent hardware [1].

has a clock frequency of 2.66 GHz. So this processor can handle $4 * 4 * 2.66 \approx 42$ GFLOPS. For GPU's it is a bit harder to calculate the actual GFLOP unit. This is because (Nvidia) graphic cards use three clocks: a memory, processor (or shader) and core (or graphics) clock. To measure the actual peak performance, the shader clock is the most important because it defines the speed at which the program fragments (shaders) that perform calculations on individual elements (and produce information related to that element) can run. As an example consider the Nvidia GeForce GTX 295 graphics card which has 2 GPU units with 240 multiprocessors each and its shader clock is set to 1.242 GHz. In its turn, every multiprocessor has 8 cores. The CUDA Programming Guide [1] states that 'it takes 4 clocks for one multiprocessor to handle one warp', which means that it takes 4 clocks for 8 cores to handle one instruction for all the 32 threads in a warp. In other words, each core can handle one instruction per clock cycle. This means that the GeForce GTX 295 can handle $480 * 1.242 \approx 596$ GLOPS. This value is much lower than stated in Figure 5.2, because a GPU can use an extended instruction set for floating point operations: multiply and add is a single instruction and it can also perform one extra multiply (which is called the dual-issued single precision floating point multiplication) per clock cycle, setting the total to $3 * 596 \approx 1800$ GFLOPS. However, cryptographic functions like MD5 or AES do computations on 32-bit integers, not on floating points. To compare the speed between CPU's and GPU's with regard to cryptography it is necessary to define GIPS: Giga Instructions Per Second. Instructions that require floating points, such as *multiply and add* and *the dual-issued single precision floating point multiplication*, cannot be used on integers and therefore the number of instructions per second (IPS) is smaller than the number of floating point operations per second (FLOPS).

	Compute Capability 1.x	Compute Capability 2.0
32-bit integer add, logical operation	8	32
32-bit integer shift, compare	8	16
32-bit integer multiply	Multiple Instructions	16
24-bit integer multiply	8	Multiple Instructions
Type conversions	8	16

Table 5.1: Throughput of native arithmetic instructions (operations per clock cycle per multiprocessor)[1].

Table 5.1 shows the throughput of the arithmetic instructions (in operations per clock cycle per multiprocessor) that are natively supported in hardware for (Nvidia) devices of various compute capabilities. If we assume that 24-bit integer multiply is sufficient for all calculations, we can still state that one multiprocessor can handle 8 arithmetic instructions per clock cycle. So this means that the GeForce GTX 295 (with compute capability 1.3 and 60 multiprocessors) can handle $60 * 8 * 1.242 \approx 596$ GIPS. For a typical CPU the number of instructions per second is more difficult to determine. In practice, this number depends on the algorithm, instruction set used, memory bandwidth and bus size. For example, an Intel Core i7 980EE can handle about 45 instructions per clock cycle at 3.3 GHz[1], which leads to a total of $3.3 * 45 \approx 148$ GIPS.

5.2 GPU application programming interfaces

Developers that want to produce code that exploits the power of GPU's for generic programming could use one of the available application programming interfaces (API's). The most currently used API's for GPU programming are:

- *Compute Unified Device Architecture (CUDA)* This parallel computing architecture, developed by Nvidia and released begin 2007, provides both a low level and high level API. It only works on CUDA enabled graphic cards from Nvidia and is basically an extension of the C programming language. The CUDA drivers are needed to run an executable compiled with CUDA. The main advantages of this architecture are:
  - code can read from arbitrary memory addresses,
  - fast shared memory, which acts as an user-managed cache, can be used by threads residing on the same multiprocessor,

[1]Calculated with the Dhrystone benchmark.

  - full support for integer and bitwise operations.

  The main disadvantage of CUDA is the fact that it is only available for Nvidia graphic cards that support it.

- *FireStream* This parallel computing architecture is basically ATI's counterpart of CUDA which has its own software development kit (SDK), called 'Stream'. This SDK includes a hardware optimized version of the 'Brook' language developed by Stanford University, which is itself a variant of the C language, optimized for parallel computing. It has basically the same (dis)advantages as Nvidia's CUDA.

- *Open Computing Language (OpenCL)* Because both FireStream and CUDA are proprietary, the Khronos Group launched an open standard for parallel programming on heterogeneous systems. This computing language does not only restrict the programmer to use the power of GPU's, but it also addresses the power of every (parallel) computing device found in personal computers, servers and handheld devices. The first stable release dates from June 2010, so its is fairly new (by time of writing this thesis). The main advantages of this API are:

  - every computing device can be addressed and used for heterogeneous generic programming,
  - it is an open standard.

  However, OpenCL is a very recent technology and therefore the main disadvantage is the lack of support and integration.

The architecture used in this research is CUDA. The available cluster of GPU's is from Nvidia, which limits this research to either OpenCL or CUDA. Since OpenCL is a fairly new development language with relatively low use cases and support and CUDA has proven to be stable for some years, we have chosen the latter. In addition, CUDA has better reference guides and examples to learn from which results in a steeper learning curve. This makes it easier for CUDA to get the best performance out of Nvidia cards than OpenCL. Moreover, Nvidia has released several tools that make the programming easier. It has for example a profiler, which helps to get an overview of your CUDA programs, and an occupancy calculator, which determines the best configuration settings for a specific graphic card.

5.3 The CUDA programming model

The CUDA programming model enforces cooperation between hardware, software and different tools to compile, link and run CUDA code. This section will

describe the compiler model, the mapping from the hardware to the software (the execution model) and how the memory on the GPU can be used. Moreover, the CUDA programming model also contains a collection of compilers and tools to optimize CUDA code.

5.3.1 Compiler model

The CUDA programming model supports two ways of writing programs. This research will only focus on the CUDA C variant, which is a minimal set of extensions to the C language[2]. A source file can contain a mixture of host code (i.e. code that executes on the CPU) and device code (i.e. code that executes on the GPU). The *nvcc* compiler first separates the host and device code. Whereas the host code (plain C for example) is then compiled and linked by the programmer's favorite compiler, the device code is compiled by nvcc to an assembly form (the *PTX* code). Then this PTX code can either be converted to binary form, which is called a *cubin* object, or loaded by the application at runtime and get compiled via the *just-in-time compilation* mechanism.

5.3.2 Execution model

Because both the GPU hardware and the CUDA framework are designed by Nvidia, these two interact very well with each other. The most important aspect to understand how CUDA works is the mapping between the API and the hardware. The soft- and hardware entities in the programming model are shown in Figure 5.3. Every entity in the figure will be described below:

- *Device* The device is defined as the GPU hardware which runs the code.
- *Host* The host is defined as the CPU hardware which controls the GPU.
- *Kernel* A kernel is defined as a function that runs on the device. As mentioned earlier, the CUDA architecture is based on the SIMT principle. This means that every thread processor executes the same function in parallel. Only one kernel is executed at the time and therefore many threads can execute that kernel simultaneously.
- *Thread and Tread Processor* A thread runs a kernel which is executed by the thread processor. Every thread has a unique id.
- *Thread Block and Multiprocessor* Every thread block contains a predefined number of threads. A thread block is then executed by one multiprocessor. However, several concurrent thread blocks can reside on one multiprocessor, limited by multiprocessor resources such as shared memory and number

[2]The other way to write CUDA programs is via the CUDA driver API.

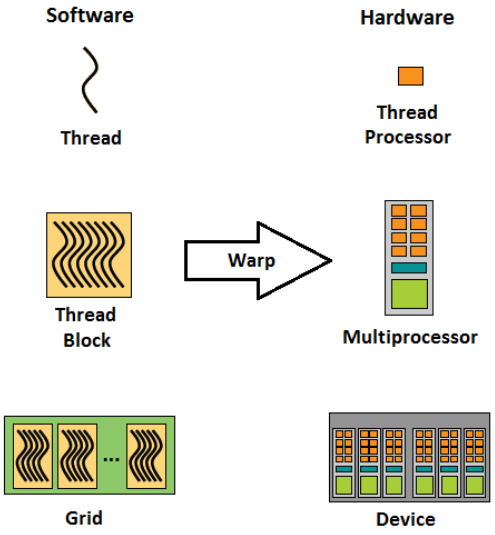

Figure 5.3: The CUDA execution model [1].

of available registers. Threads within a thread block can cooperate and synchronize, while threads in different blocks cannot.

- *Grid and Device* Every grid contains a predefined number of thread blocks, so a kernel is launched as a grid of thread blocks.

The number of threads in a thread block and the number of thread blocks in a grid can be determined at compile time or runtime. This allows programs to transparently scale to different GPU's and hardware configurations. The hardware is free to schedule thread blocks on any processor as a long as it fits in the warp size. When a multiprocessor is given one or more thread blocks to execute, it partitions them into warps that get scheduled by a warp scheduler for execution. The way a block is partitioned into warps is always the same. Each warp contains threads of consecutive, increasing thread id's with the first warp containing thread 0. The configuration of a kernel (in number of grids, thread blocks and threads) significantly influences the execution speed. It is important to divide the threads evenly over every warp. We will discuss optimal kernel configuration in Chapter 7.2.3.

Figure 5.4 gives an overview of how threads, thread blocks and grids can be defined. Over time, every kernel is executed depending on its own configuration. This configuration is described by the number of threads per block, the number of blocks per grid, the amount of static memory needed and the algorithm specific parameters.

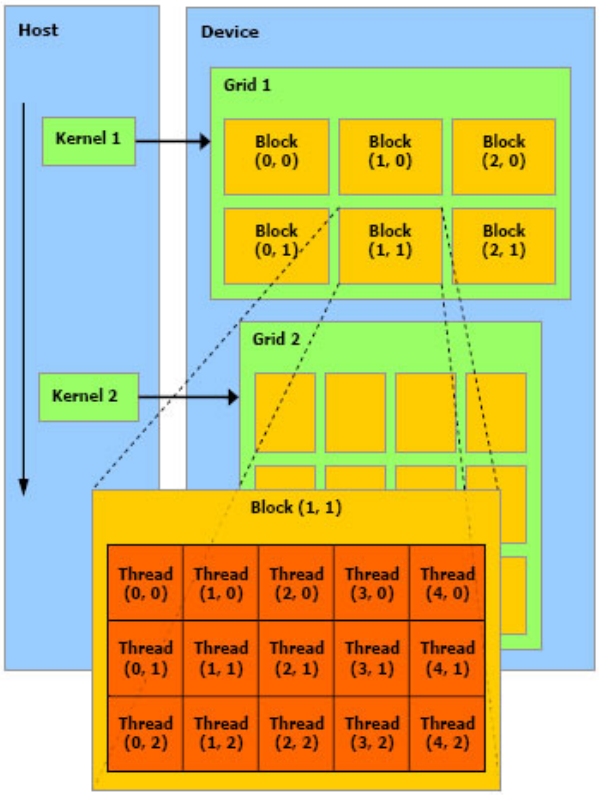

Figure 5.4: An example of a kernel configuration [1].

5.3.3 Memory Outline

Figure 5.5 gives an overview of the available memory on a typical Nvidia CUDA enabled GPU.

The following different classes of physical memory can be distinguished:

- *Device Memory* This is the largest memory, with sizes up to 1.5 GB, but also the memory with the highest latency with regard to the individual thread processors. It can be compared with the DRAM of a normal CPU. The device memory is virtually divided into the following parts:
  - *Global memory* Global memory resides in device memory and is accessed via 32-, 64-, or 128-byte transactions. These memory transactions must be aligned. This means that only the first 32-, 64-, or 128-byte segments of the global memory that are aligned to their size (their first address is a multiple of their size) can be read or written by memory transactions. When a warp executes an instruction that accesses global memory, it coalesces the memory accesses of the threads within the warp into one or more of these memory transactions depending

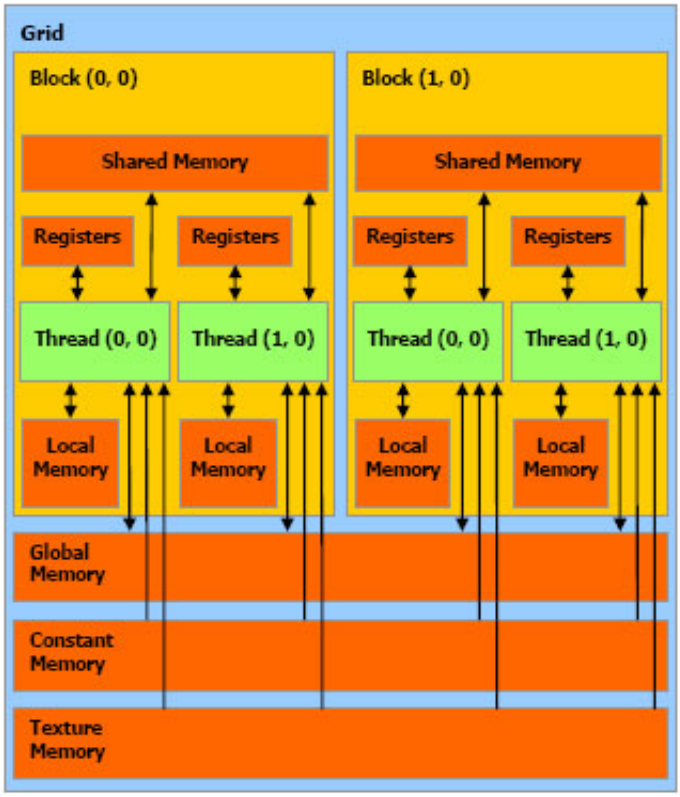

Figure 5.5: Overview of the CUDA memory model [1].

on the size of the word accessed by each thread and the distribution of the memory addresses across the threads. The more transactions are necessary, the more unused words are transferred compared to the words accessed by the individual threads, which reduces the instruction throughput. For example, if a thread wants to access 4 bytes, but a 32-byte memory transaction is generated, the throughput is divided by a factor 8.

– *Local memory* The memory outline in Figure 5.5 shows that the local memory is a distinctive unit close to the individual thread processor. The local memory space resides in device memory however, so local memory accesses have the same high latency and low bandwidth as global memory accesses. In contrast to global memory, this type of memory is organized in such a way that consecutive 32-bit words are accessed by consecutive tread id's and the access is therefore fully coalesced as long as the threads in a warp originate from the same relative address. Local memory can not be allocated: variables are automatically placed in local memory by the compiler if they not fit in

faster memory or registers (also called *register spilling*). The compiler uses various criteria for this: arrays that are not indexed with constant quantities, arrays that consume too much register space or kernels that requested more registers than available.

- *Constant memory* Every device has a constant memory space. For most devices the size of this memory is 64 KB. While this memory resides on the device memory, it is cached on-chip. This means that a read from constant memory only costs one read from the cache, which is significantly faster than global memory: for all threads that belong to a half warp (either the first or the second half of a warp), a read from the constant cache is as fast as a read from register as long as all threads read the same address. Reading from different addresses by threads in the same half warp are serialized, which means that the cost (expressed in memory latency) to access the constant memory scales linearly with the number of different addresses read by the threads in a half warp.
- *Texture memory* Texture memory has basically the same properties as constant memory except that it is optimized for 2D spacial locality, which means that threads of the same warp that read texture addresses that are close together will achieve best performance. While the use of texture memory can significantly improve the output of graphic CUDA programs, we will not go in to detail because password cracking does not use graphics processing.

The global, constant and texture memory can be set by the host and therefore allow a developer to use them directly. Local memory can only be set by individual threads and should therefore be taken care of in advance (e.g. by placing large arrays in the shared memory).

- *Registers* In Figure 5.5 is shown that every thread has it own set of registers. This is true in the sense that every multiprocessor has a number of registers which should be shared by all his thread processors. A typical multiprocessor (like the ones on a Nvidia GTX295) has 8192 32-bit registers. Basically, every access to a register by the thread processor is immediate, which means that it takes zero extra clock cycles per instruction. Delays may occur by read-after-write dependencies and register bank conflicts. A read-after-write dependency occurs when the instruction's input operands are not yet available because some of the operands are written by previous instructions whose execution has not completed yet. The latency is then equal to the execution time of the previous instruction and the warp scheduler must schedule instructions for different warps during that time. The execution time varies depending on the instruction, but the latency

is on average 24 clock cycles. A *bank* is an equally-sized set of memory modules, which can be accessed simultaneously by different threads. *Bank conflicts* occur when two addresses of a memory request fall in the same memory bank and access to those banks is then serialized by the hardware. The compiler and hardware thread scheduler will schedule instructions as optimally as possible to avoid register memory bank conflicts.

- *Shared Memory* Every multiprocessor has its own shared memory that is shared between all threads in a block (even the ones that are not in the active warp). Because shared memory resides on-chip, it is much faster than local and global memory. In fact, uncached shared memory latency is roughly 100 times lower than global memory latency, provided that there are no bank conflicts between the threads. Bank conflicts in shared memory will be described more detailed in Chapter 7.2.2. To achieve low latency, the available shared memory is divided into banks. Due to this partition, any memory read or write request made of n addresses that fall in n distinct memory banks can therefore be serviced simultaneously, increasing the overall bandwidth by n times the bandwidth of a single module. Shared memory is not initialized automatically and should therefore be assigned before compile time. Because every thread in a block can access the shared memory and the kernel configuration is not always known at compile time, the programmer has to align the memory and make sure that every thread accesses its own piece of shared memory.

To summarize, Table 5.2 shows the features of a Nvidia's GPU device memory.

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	No	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	No	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

Table 5.2: Features of Nvidia's GPU device memory (with compute capability 1.3)[1].

For an overview of the Nvidia 295 GTX specifications, we refer to Appendix A.1

Chapter 6

Cracking MD5-crypt with GPU hardware

In the previous chapters we gave a definition of password hashing schemes, defined their security properties and described several attack scenario's. Furthermore, we described the internal working of the graphics processing unit. With this knowledge, we are now able to define an exhaustive search attack with GPU's on a specific password hashing scheme. This chapter will cover our approach to launch an attack on MD5-crypt.

6.1 Considerations

Since this research focuses on one attack case only, some assumptions have to be defined. Our implementation is based on the following considerations.

- *Attacker model* We assume that the adversary definitely has to find the plaintext password given the ciphertext (ciphertext only attack) and therefore should perform an exhaustive search over all possible inputs. Furthermore, the attacker does not have control over the salt and because the salt is 48 bit in size, we consider time-memory trade-off not valuable enough to exploit.

- *Hardware* The exhaustive search is performed by Nvidia GPU's, since we have chosen CUDA to be our specific programming interface. However, the optimizations that described could be used for different architectures as well.

- *Password generation* To make sure the performance of the implementation can be compared with real password recovery tools, unique passwords need to be generated. The performance is than measured in unique password hashes that can be checked per second. Since some of the optimizations depend on the password length, we assume the password length to be less than 16 bytes. [1]

- *Law of averages* We cannot use the law of averages with our exhaustive search since we use a linear password generation algorithm (thread 0 checks 'aaaa', thread 1 checks 'baaa', etc.) and since we assume that the password input space is not uniformly random distributed[26].

6.2 Design of MD5-crypt

The scheme that will be reviewed in this research is the MD5-crypt scheme. MD5-crypt is the standard password hashing scheme for FreeBSD operating systems, supported by most Linux distributions (in the GNU C library) and implemented in Cisco routers. It was designed by Poul-Henning Kamp in 1994.

Figure 6.1 shows the schematic overview of the MD5-crypt implementation. The figure only shows the most relevant parts and abstracts from initializations. Basically, MD5-crypt applies the MD5-compression function 1002 times:

- In the first application, the concatenation of the password, salt and password again is hashed.

[1] 16 bytes is about twice the average password length of real world datasets (see Table 4.2).

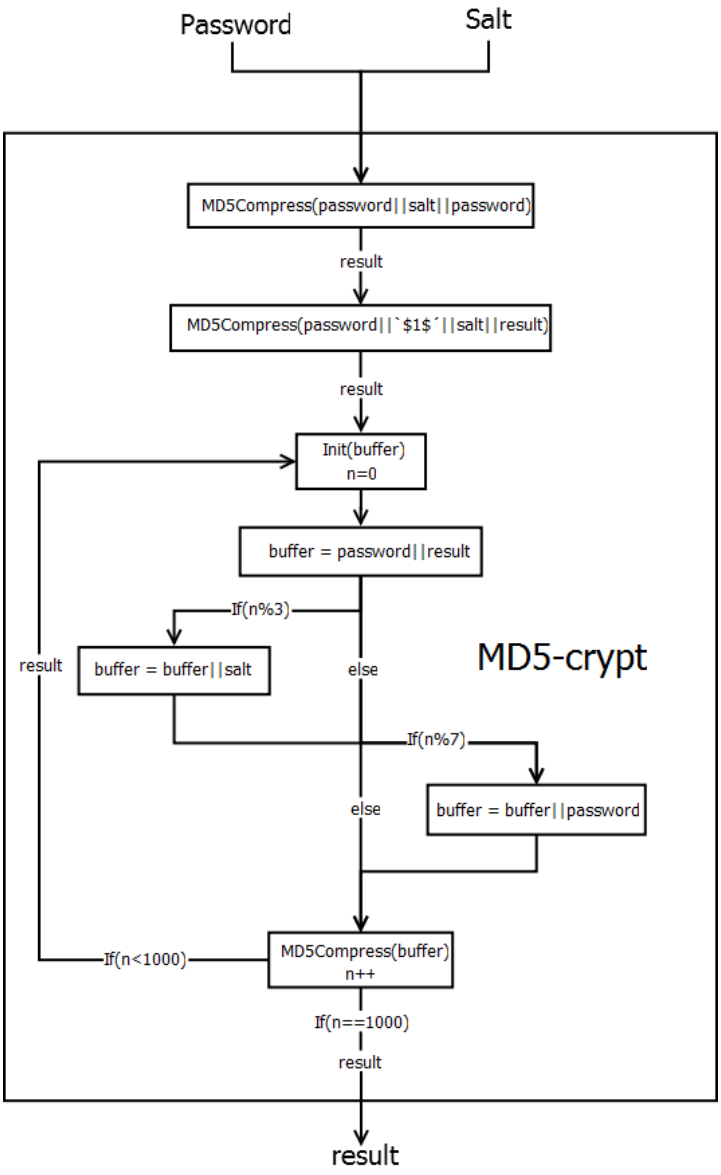

Figure 6.1: Schematic overview of MD5-crypt.

- In the second application, the concatenation of the password, magic string ('1'), salt and the result of the first application is hashed.

- In the next thousand applications, the concatenation of the password, salt and result of the previous application is hashed based on the round number n.

Although Poul-Henning Kamp did not write a design document, he put comments throughout the source code and was asked to review the security of the

scheme in an article about open source software by Jeff Norris [49]. In this article Poul-Henning Kamp states the following:

> "Fortunately, I have no reason to believe that any problem exists with either the algorithm or the implementation, and given that MD5 is pretty strong, it's unlikely that any will ever be found."

He is correct when he argues that MD5 is still pretty strong for the purpose of password encryption, since MD5 only suffers from collision attacks and no feasible pre-image attacks have been found yet. There is a theoretic attack by Sasaki and Aoki [58] that generates a pseudo pre-image of MD5 with a complexity of $2^{116.9}$ and generates a full pre-image of MD5 with a complexity of $2^{123.4}$. These numbers are still not feasible for contemporary and near future hardware, and even if they were, the key stretching technique makes the attack even n times harder.

As we have seen in Chapter 4.5, collisions can theoretically influence password hashing schemes. Wang et al. [59] and Leurent [39] have shown that for the APOP protocol strong collision attacks can be used to recover the plaintext password. The APOP protocol uses one application of MD5(P, C), where P is the password and C a random nonce. If an adversary has access to C, he can launch a collision attack. This attack assumed the following conditions:

1. There exists no message difference in the last part of messages.
2. Many collisions can be generated in practical time.

MD5-crypt uses the key-stretching technique in such a way that it becomes very hard for an adversary to exploit the break of the strong collision resistant property of MD5. The attack from [59] could work theoretically if the salt and password are concatenated in the same way with every iteration of the hash function and the salt could be arbitrarily chosen by the adversary. Then an evil salt could be created such that two different inputs collide under the salt and its evil twin [18]. Since MD5-crypt concatenates the password, salt and result of the previous iteration in a pseudo-random way (which breaks the first condition) and since MD5-crypt uses 1000 iterations (which breaks the second condition), we have no belief that collision attacks affect the security of MD5-crypt.

6.3 Enabling exhaustive search attacks on GPU's

6.3.1 Parallelization of MD5-crypt

To check whether the MD5-crypt algorithm can be parallelized, it is necessary to verify if the MD5 hash function itself can be parallelized. In the MD5-Compression function, each 512-bit input block is digested in 4 phases. Each phase consists of 16 basic steps, for a total of 64 basic steps. Then, each step

updates one word of a 4-word accumulated digest, using the entire intermediate digest as well as block data and constants (the Markle-Damgard principle). So in general, each basic step depends on the output of the prior step, defeating simple parallelization of the steps. The same concept holds for MD5-crypt, in the main loop (where the MD5-Compression functions is applied a 1000 times), the output of the last round serves as input for the next round, again defeating parallelization of the rounds.
While it is not possible to parallelize the MD5-crypt algorithm itself, an exhaustive search attack to find a matching password for a given hash can be parallelized easily. In fact, exhaustive searches are *embarrassingly parallel*[24], since the parallel processing units of the underlying hardware do not have to interact or cooperate with each other. Every processing unit tries a set of possibilities and compares them with a target. The only cooperation that is needed, is the division of the search space and a general stop message that terminates the search when one processing unit has found a match. If we map this approach to MD5-crypt, every processing unit needs to calculate the MD5-crypt output hashes of all the candidate passwords in the input set he is given and match them with a target hash.

6.3.2 Algorithm optimizations

This section describes the optimizations that can be done in the algorithm itself in order to speed up the execution of one MD5-crypt round. The main part of the MD5-crypt password hashing scheme consists of the application of the MD5-compression function a thousand times. So every optimization in the MD5-compression function will yield a linear speed up of 1000 times in the MD5-crypt function. Based on the assumption of the password length, two major optimizations of the MD5 hash function can be used.

The first optimization is based on the fact that passwords with sizes lesser than 16 bytes result in only one application of the MD5-compression function. As described in Chapter 3.2.3, the MD5 hash function operates on a b-bit message M which is padded to make the length of the message a multiple of 512. For every 512-bit block of the message M, the MD5-compression function is called. However, if M is only 512 bit long, the MD5-compression function is called only once. So if the input to the MD5 hash functions in MD5-crypt is only 448 bit long a linear speed up can be achieved. To calculate the maximum length of the password that still allows this optimization, we have to find the call to the MD5-compression function in the MD5-crypt algorithm that uses the largest input. Depending on the round in the algorithm, the password P, salt S and the result of the last application of the MD5-function R are concatenated in such a way that it uses at most $2 * |P| + |S| + |R|$ bytes. Because the salt and the result of last round are always 8 and 16 byte respectively, the following should hold for

$|P|$:

$$2 * |P| + 24 < \frac{448}{8} \Rightarrow |P| < 16. \tag{6.1}$$

The second optimization is based on the fact that the length of passwords is typically significantly shorter than 16 bytes. As described in Section 3.2.4, the MD5-compression function operates on a 512-bit message block M_i, which is partitioned into sixteen consecutive 32-bit words $m_0, \ldots, m_{15}$, and then expanded to 64 words $(Wt)_{t=0}^{63}$ for each of the 64 steps in the function. Because of the assumption of the password length, the length of the input for the MD5-compression function can be calculated in advance. This means that not all of the input words $m_0, \ldots, m_{15}$ are initialized to a value. Regarding Equation 6.1, the number of input words N_{null} that are zero can then be calculated as:

$$N_{null} = \left\lfloor \frac{56 - (2 * |P| + 24)}{4} \right\rfloor + 1 = \left\lfloor \frac{16 - |P|}{2} \right\rfloor + 1. \tag{6.2}$$

The 4 in the denominator comes from the fact that 4 bytes make up 1 word and the addition with 1 comes from the fact that the last word is not needed to represent the length of the input (the input length is smaller than 2^{32}).

6.4 Theoretical analysis of the maximum performance

In this section three ways to estimate the maximum performance of MD5-crypt are described. First, the theoretic upper bound is estimated by counting the number of instructions needed to complete one application of MD5-crypt. Then based on the number of clock cycles per second of a given device, it is possible to estimate the performance. While this approach is applicable for all kinds of devices, the second analysis will yield a more practical and specific approach, especially for CUDA enabled GPU's. The performance is now estimated by a prediction model. The last approach is based on our own implementation and will be described in detail later. To compare the different approaches, the performance is measured in *hashes per second*, i.e. the number of candidate passwords that can be checked on a given device within one second. Furthermore, the models are based on a variable password length with a maximum of 15 bytes.

6.4.1 Simple arithmetic model

To estimate the performance of MD5-crypt on a given architecture, we first define a simple model that is based on the number of arithmetic instructions needed to complete one round of the password hashing scheme. Since it is not very hard to estimate the instruction throughput of a hardware platform, this model can be used to compare the performance of the hashing scheme on different architectures and determines the maxim speedup that can be achieved. To define this model, all arithmetic and logic operations are taken into account.

To determine the number of arithmetic instructions needed to complete one round of MD5-crypt, it is necessary to determine the number of instructions for MD5 first. Recall from Chapter 3.2.4 that the MD5-compression function does all the arithmetic work. It consists of 64 rounds in which the following function is called:

$$Q_{t+1} = Q_t + ((Q_{t-3} + f_t(Q_t, Q_{t-1}, Q_{t-2}) + W_t + AC_t) <<< RC_t) \text{ for } 0 \leq t < 64.$$

In the equation, f_t depends on the round number and is

$$f_t(X,Y,Z) = \begin{cases} F(X,Y,Z) = (X \wedge Y) \oplus (\overline{X} \wedge Z) & \text{for } 0 \leq t < 16, \\ G(X,Y,Z) = (Z \wedge X) \oplus (\overline{Z} \wedge Y) & \text{for } 16 \leq t < 32, \\ H(X,Y,Z) = X \oplus Y \oplus Z & \text{for } 32 \leq t < 48, \\ I(X,Y,Z) = Y \oplus (X \vee \overline{Z}) & \text{for } 48 \leq t < 64. \end{cases}$$

To summarize, in every round Q requires 3 additions, one cyclic rotation and applies one subfunction. The cyclic rotation requires 2 shifts and 1 addition. Table 6.1 shows the number of logic operations for the subfunctions.

Subfunction	Logic Operations	One application of Q	16 applications of Q
F	4	11	176
G	4	11	176
H	2	9	144
I	3	10	160
Total after 64 rounds			656

Table 6.1: Instruction count of the elementary MD5 functions.

Every subfunction is called 16 times, and together with the arithmetic operations of Q and the cyclic rotation, the total number of native arithmetic operations for one application of the MD5-compression function will be 656 instructions. Since MD5-crypt basically applies the MD5-compression function 1002 times, it means that one application of MD5-crypt costs 657312 native arithmetic instructions. Note that this calculation only takes the native arithmetic instructions into account and not the memory accesses (for example the operations to correctly set the 64-byte input for the MD5-function). With this information, and the information about how to calculate the integer operations per second for a given architecture (as explained in Chapter 5.1.2), a performance comparison between different architectures can be made. Table 6.2 shows the performance of MD5 and MD5-crypt on different architectures. Note that we do not take compiler optimizations into account, so both architectures need to do the same amount of arithmetic instructions to execute the algorithms. The table shows us that better equipment does not always implies better performance. For example,

Architecture name	Avarage price in EUR	Peak performance in GIPS	Theoretical speed MD5 (hash/sec.)	Theoretical speed MD5-crypt (hash/sec.)
Nvidia GTX295	290	596	908M	906K
Nvidia GTX580	450	790	1204M	1202K
Nvidia S1070	6000	1382	2107M	2100K
Intel i7 965EE	450	76	116M	115K
Intel i7 980EE	900	147	224M	223K

Table 6.2: Performance comparison between different architectures.

the Nvidia Tesla S1070 has an average price of 6000 Eur, but produces only a little over twice the performance over the GTX295, while the average cost is 15 times higher[2]. This due to the design of the S1070: it is built out of four separate Tesla T10's and intended for memory intensive implementations. Therefore it has a total of 4 GB of global memory, which is expensive.

6.4.2 Performance prediction model

Since the previous model only takes the number of operations into account and neglects the memory accesses, the model will not be very accurate for a performance estimation of real implementations. In addition, the CUDA execution model is complicated and performance can not be measured by simply calculating the number of clock cycles and memory accesses. It also depends on the compute capability of the device and the way the architecture maps the software implementation on the hardware. A more accurate way to estimate the performance of real CUDA implementations was proposed by Kothapalli et al. [38]. Their model takes into account the various facets of the GPU architecture like scheduling, memory hierarchy and pipelining. It is based on the arithmetic intensity and memory accesses per kernel, which is executed by each thread. Let the number of clock cycles required for arithmetic computation in a thread be denoted as N_{comp} and let the number of clock cycles required for memory accesses (both shared and global memory) be denoted as N_{mem}. Now let $C_T(K)$ be the number of clock cycles required to execute kernel K on thread T. If the hardware is capable of hiding all the memory latency (which depends on the configuration and the nature of the problem one wants to solve), the MAX model can be used:

$$C_T(K) = \max(N_{\text{comp}}, N_{\text{mem}}) \tag{6.3}$$

[2]For overview peak performance and specifications of Nvidia GPU's, see: http://en.wikipedia.org/wiki/Comparison_of_Nvidia_graphics_processing_units

If the hardware cannot hide all the latencies, the worst case scenario is used to determine $C_T(K)$ (all memory accesses are serialized), which is called the SUM model:

$$C_T(K) = N_{\text{comp}} + N_{\text{mem}} \tag{6.4}$$

Depending on the model, the number of clock cycles required for the kernel K to execute, denoted as $C(K)$, can be defined as:

$$C(K) = N_B(K) * N_w(K) * N_t(K) * C_T(K) * \frac{1}{N_C * D}. \tag{6.5}$$

In this equation, $N_B(K)$ is the number of blocks assigned to each multiprocessor, $N_w(K)$ is the number of warps that fit in each block, $N_t(K)$ is the number of threads in a warp, $C_T(K)$ is the maximum number of cycles required by any thread T executing kernel K (where $C_T(K) = \max_T C(T)$), N_C is number of processors in a multiprocessor and D is the depth of the pipeline, i.e. a multiprocessor can execute D threads in parallel. The total time in seconds $T(K)$ to execute a kernel is then defined as:

$$T(K) = \frac{C(K)}{R}. \tag{6.6}$$

All the previously defined variables are depending on the kind of problem the GPU has to solve. The MAX model can be applied for arithmetic intensive implementations while the SUM model can be used best for memory intensive implementations.

Case study: MD5-crypt

To estimate the performance of MD5-crypt with the practical based model, we have to determine number of threads that a we need to count the number of hashes per second. If we assume that every unique thread T can review one password candidate, it is possible to determine the number of clock cycles $C(K)$ a kernel K requires to execute one round of MD5-crypt. Together with a predetermined number of password candidates N, it is possible to estimate the number of hashes per second with this model. For the sake of simplicity, let N be a multiple of 32 (the number of threads in a warp), for example 320 000. If we define a block size of 256 threads and we have a device with 60 multiprocessors (like the Nvidia GTX295), the number of blocks per multiprocessor $N_B(K) = \lceil \frac{320000}{60*256} \rceil$. Now it is necessary to determine the number of clock cycles required for arithmetic computation N_{comp} and the number of clock cycles required for memory accesses N_{mem} done in one thread that executes one instance of MD5-crypt for one candidate password. N_{comp} is the same as the amount in Section 6.4.1, which is 657312. To determine N_{mem}, we just count all the memory accesses in the default MD5-crypt implementation. However, this value is dependent on the

password- and salt length. If we assume that both have a length of 8 bytes, the total number of memory accesses will be 110394 for one thread that executes one instance of MD5-crypt for one candidate password. Further more, on a Nvidia GTX 295 the number of cores per multiprocessors N_C is 8, the maximum number of threads in a warp $N_t(K)$ is 32 and the depth of the pipeline D is 1. Finally, because the block size is 256, the number of warps that fit in each block $N_w(K)$ is 8. With the previous information available, it is now possible to estimate the kernel execution time for 320000 candidate passwords, by combining Equation 6.5 and 6.6. Figure 6.4.2 shows the performance estimates for the SUM and MAX model, both for accessing shared memory (which takes on average 4 clock cycles per access [1]) and global memory (which takes on average 500 clock cycles per access [1]). Furthermore, the performance estimate for the theoretical model is shown. While the difference between the shared memory and the global memory approach seems large (900K versus 11K hashes per second respectively), in practice the warp scheduler will try to hide the latency by swapping threads that are waiting on a memory request and therefore the practical values will be higher. The degree of hiding depends on the configuration and the implementation itself. The difference between the values of the MAX model with shared memory and the theoretical model is caused by the configuration settings in the practical model. If the workload is not divided amongst all multiprocessors equally, the performance will decrease.

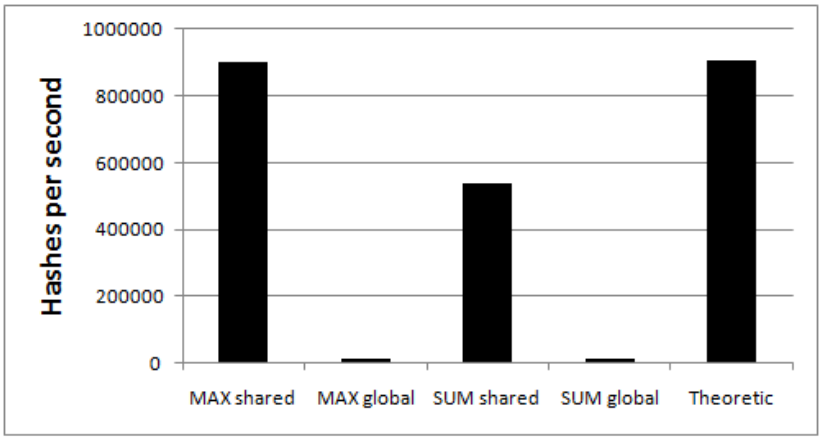

Figure 6.2: Performances achieved for MD5-crypt on a Nvidia GTX295, calculated by the practical based model.

We have shown the theoretical performance increase for MD5-crypt executed on GPU hardware. Furthermore we have shown a practical based estimated of the performance when including the CUDA memory model. It shows us that it is of utmost importance to ensure that all the variables should be stored in shared memory. In the next chapters we will describe our proof of concept, which is

an implementation of MD5-crypt on Nvidia CUDA enabled GPU's, including a description on how to optimize exhaustive searches for password hashing schemes on GPU's.

Chapter 7

Optimization of our implementation on a CUDA enabled GPU

Now we have estimated the maximum theoretical and practical performance of our approach, this chapter will treat the specific optimizations that can be used in order to increase the performance on a CUDA enabled device. It will also cover the implementation choices we have made.

7.1 Implementation details

This section contains the actual implementation in pseudocode. Only the most relevant parts are reviewed here. For an overview of all code, we refer to `http://www.martijnsprengers.eu/phKrack/`. The implementation itself differs from the original implementation by Poul-Henning Kamp because all calculations are done on integers instead of bytes and no conversion to base 64 takes place, since it is enough to compare the 128 bit output of the final MD5-compression function with the 128 bit hash of the target password. Furthermore, the algorithm specific optimizations from Chapter 6.3.2 are now incorporated. GPU and CUDA specific code is left out.

The algorithm can be described by the following steps:

1. Decode the base 64 target hash to four 32-bit words (CPU).
2. Calculate the size of the search space, based on the character set and maximum password length (CPU).
3. Initialize n GPU kernels with the salt, target hash, configuration parameters (such as grid size and number of threads per block) and the i-th (where $i < n$) part of the search space (CPU).
4. Depending on the virtual thread number, generate an unique password candidate. Since thread creation does not render large overhead, every thread checks one password candidate.(GPU)
5. Hash the candidate password is with MD5-crypt, which is described by Algorithm 1, 2 and 3 (GPU).

Algorithm 1 MD5-crypt pseudo code, Context 1.

```
1: unsigned integer buffer[16]
2: unsigned integer final[4]
3: set(buffer,password||salt||password)
4: set(buffer,0x80)                          ▷ Add the binary 1
5: set(buffer,msglen << 3)                   ▷ Add the message bit length
6: MD5Compress(final,buffer)                 ▷ Call MD5 and store result in final
```

Algorithm 1 shows how the first context is built. It sets up $buffer$, which contains the concatenation of the password, salt and password. Then, $buffer$ is structured as described by Chapter 3.2.3 and the MD5-compression function is called. The result is stored in the $final$ array. This result is then fed into context two (shown in Algorithm 2), where a new $buffer$ is built. This time it contains the concatenation of the password, the string '1' and the salt. Furthermore n bytes of the result of the first context are added, where n is the length of the

password. Then, depending on n, a null byte or the first byte of the password is added. Finally, $buffer$ is padded and partitioned before the MD5-compression function is called. Again, the result of this context is stored in the $final$ array and fed to the third context (Algorithm 3). This context contains a loop with 1000 iterations. Every iteration, the result of the last round ($final$) and the password are stored in $buffer$. Then, depending on the iteration number, the password and salt are added, and finally the MD5-compression function is called, which stores the result in $final$ again. At the end, the result of the last iteration contains the actual password hash, which is then matched against the target hash. If both match, a flag is set and the algorithm can stop searching. This concludes the description of the most important part of the MD5-crypt algorithm.

Algorithm 2 MD5-crypt pseudo code, Context 2.

```
1:  set(buffer,password||$1$||salt)
2:  set(buffer,final[len(password)])        ▷ Add len(password) bytes of final
3:  for i = passwordLength, i, i >>= 1 do
4:      if i & 1 then                                          ▷ Case i is odd
5:          set(buffer,\0)                                     ▷ Add a null byte
6:      else                                                   ▷ Case i is even
7:          set(buffer,pw[0])                  ▷ Add the first byte of the password
8:      end if
9:  end for
10: set(buffer,0x80)                                           ▷ Add the binary 1
11: set(buffer,msglen << 3)                          ▷ Add the message bit length
12: MD5Compress(final,buffer)                ▷ Call MD5 and store result in final
```

7.2 Optimizations

This section describes the types of optimizations that can be used within the CUDA framework. Some types result in a significant performance increase while others even decrease the overall performance. The optimizations described in this section are applied to our implementation. In general, the programmer has to determine if this specific type of optimization affects his own implementation. The discussed optimizations are mentioned in [1, 2].

7.2.1 Maximizing parallelization

To maximize the utilization it is important that the parallelism is efficiently mapped to the different levels of the architecture. Three main levels are distinguished.

Algorithm 3 MD5-crypt pseudo code, main loop.

```
1: msglen = 0
2: for i = 0, i < 1000, i++ do
3:     msglen = 0
4:     for j = 0, j < 16, j++ do
5:         buffer[j] = 0
6:     end for
7:     if i & 1 then                                   ▷ Case i is odd
8:         set(buffer,password)
9:         msglen += len(password)
10:    else                                            ▷ Case i is even
11:        set(buffer,final)                           ▷ Add the result of last round
12:        msglen += 16
13:    end if
14:    if i % 3 then                                   ▷ Case i ∤ 3
15:        set(buffer,salt)
16:        msglen += len(salt)
17:    end if
18:    if i % 7 then                                   ▷ Case i ∤ 7
19:        set(buffer,password)
20:        msglen += len(password)
21:    end if
22:    if i & 1 then                                   ▷ Case i is odd
23:        set(buffer,final)                           ▷ Add the result of last round
24:        msglen += 16
25:    else                                            ▷ Case i is even
26:        set(buffer,password)
27:        msglen += len(password)
28:    end if
29:    set(buffer,0x80)                                ▷ Add the binary 1
30:    set(buffer,msglen << 3)                         ▷ Add the message bit length
31:    MD5Compress(final,buffer)                       ▷ Call MD5 and store result in final
32: end for
33: if target == final then                            ▷ Match with target
34:    setFlag(inputPassword)
35: end if
```

Application level

According to Amdahl's law [4], the maximum speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program. It is thus desirable that most of the program can be executed in parallel. Amdahl's law states that if P is the part of a program that can be made parallel (i.e. benefit from parallelization), and $(1 - P)$ is the part that cannot be parallelized (remains sequential), then the maximum speedup that can be achieved by using N processors is calculated by:

$$S = \frac{1}{(1 - P) + \frac{P}{N}}. \tag{7.1}$$

As an example, consider Figure 7.1. It shows the maximum performance increase for a multiprocessor with 240 cores (like a Nvidia GTX295) depending on which percentage of the program can be made parallel. The figure clearly shows that the speedup is only feasible if at least 80% of the program can be parallelized.

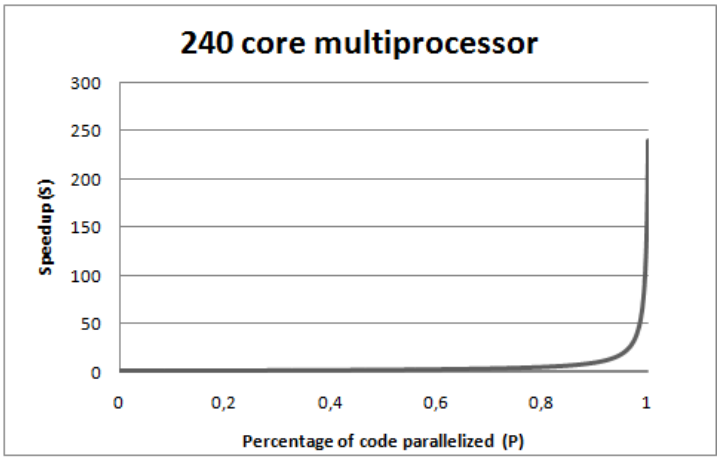

Figure 7.1: Example of Amdahl's law for a 240 core multiprocessor.

A great part of parallel programming consists of attempting to reduce the component $(1 - P)$ to the smallest possible value. Iterating over a given input space and generating passwords according to a password hashing scheme is even *embarrassingly parallel*, which means that the sequential fraction of the program is almost zero and consequently the maximum speedup can be achieved. Password crackers achieve this because of their nature: every password can be checked independently (individual threads do not need to cooperate) and all the threads have the same execution path (only the data part is different). The only thing that can not be parallelized in our implementation is the initialization and configuration of the GPU. This includes the division of the input space in case of a distributed system. However, this is only be done once and the calculation of

the actual hashes is still fully parallelized on the GPU. In addition, there are no intermediate communications between the GPU and the host CPU, which are dependent on the bus speed and slow down the execution speed.

Device level

The compute capability of the device determines the number of kernels that can reside concurrently on a device. This is an important aspect, because the more kernels that reside on a device, the more the hardware is kept busy. For example, if one kernel needs to access data on the global memory, the device can run another kernel while waiting for the memory access to complete. However, on devices with compute capability 1.x, only one kernel can reside on the device concurrently. This implies that the kernel should be launched with sufficient thread blocks, i.e. with at least as many thread blocks as there are multiprocessors on the device. Chapter 7.2.3 will treat more configuration optimizations that can be applied to our implementation.

Multiprocessor level

The application should use the *thread-level parallelism* principle at an even lower level. It is therefore important to keep the individual multiprocessors as busy as possible. This can be achieved by *latency hiding*. Whenever the warp scheduler issues an instruction, it selects a warp which is ready to execute its next instruction and then issues that instruction to the active threads in the warp. The number of clock cycles it takes for a warp to be ready for the execution of its next instruction is called *latency*. Maximum utilization is therefore directly dependent on the number of warps that can reside on one multiprocessor. If all warp schedulers always have an instruction to issue for some warp at every clock cycle during that latency period, the latency can be completely *hidden*. As described in Chapter 5.1.1, the number of instructions needed to hide a latency of L clock cycles can be determined by taking $\frac{L}{4}$, since a multiprocessor in a device with compute capability 1.x can issue one instruction per warp over 4 clock cycles [1].

A warp is not always ready to execute its next instruction. This is mainly caused by the input variables not being available yet. Since our final optimized implementation does not require calls to global memory, latency is caused by either register dependencies, shared memory accesses and constant cache misses. If all input operants are registers, for example in the MD5-compression function, the latency is caused by previous instructions whose execution is not completed yet.

7.2.2 Memory Optimizations

While hash functions have high arithmetic intensity in their calculations, the password hashing schemes based on them frequently access the memory (for example to add the password and salt to a buffer) and therefore it is mostly the memory that determines the bottleneck in the execution speed. This section describes all memory optimizations that can be done in order to improve performance. The result of each improvement will be presented in Chapter 8.

Global memory

Our implementation does not use large data structures and therefore we do not need to store data in global memory manually. However, if our data structures do not fit in shared memory and the compiler could not place them in the registers, those structures will be placed in local memory, which actually resides in global memory. Local memory is so named because its scope is local to the thread, but the physical location of this memory is off-chip and has the same properties as global memory. Only the compiler decides what variables will be placed in local memory, based on how much space the data structures need and based on how much registers are *spilled*. (See Section 5.3.3). Because global memory latency is very high compared to shared memory or register latency, the most important performance consideration is to make sure global memory accesses are coalesced. With our implementation this is already the case, since variables in local memory are organized as such that consecutive words are accessed by consecutive thread id's. Accesses are therefore fully coalesced if all threads in a warp access the same relative address. This is the case in our implementation because all threads use the same index in the arrays every time. In our first, non-optimized implementation, the following data structures are stored in per thread local memory (and thus having the same properties of global memory):

```
unsigned char  buffer  [64];
unsigned int   final   [16];
unsigned char  password[N];
unsigned char  salt    [8];
unsigned char  charset [M];
unsigned int   target  [4];
```

`buffer` is used to store the 64-byte input for the MD5-compression function, `final` is used to store the output of the MD5-compression function, `password` is the N-byte password that has to be checked, `salt` is the 8-byte salt that is the same for all threads, `charset` is the M-byte character set which is used to build a candidate password based on the current thread id and `target` is the representation of the target hash in 4 unsigned integers.

As a consequence, the kernel has to access the local memory every time it uses one of the above variables. The latency of these accesses is about 400 to 600 clock cycles (depending on the compute capability and the memory bus speed of the device). The number of warps that are needed to keep the warp schedulers busy during the latency period can be determined by the ratio between the number of instructions with on-chip memory and the number of instructions with off-chip memory. If this ratio is low, more warps resident per multiprocessor (and thus more threads per block) are required to hide the latency.

Shared memory

Although the warp scheduler uses *latency hiding* to cover the global memory latency, the access time of shared memory is almost as fast as register access time and use of shared memory is therefore preferred over global memory. Because it is on-chip, uncached shared memory latency is roughly 100x lower than global memory latency. However, all virtual threads that run on one multiprocessor have to share the available shared memory and so concurrent threads can access other thread's memory addresses. Therefore, the developer has to manage the memory accesses himself.

Individual threads can access their shared memory addresses concurrently with the use of equally sized 32-bit wide memory modules, called *banks*. Because banks can be accessed simultaneously, any memory load or store of n addresses that spans n distinct memory banks can be serviced simultaneously, yielding an effective bandwidth that is n times as high as the bandwidth of a single bank. On a device with compute capability 1.x, the shared memory has 16 banks that are organized in such a way that successive 32-bit words are assigned to successive banks, i.e. interleaved. A memory request for a complete warp is split into two memory requests, one for each half-warp, that are issued independently. Each bank has a bandwidth of 32 bits per clock cycle and since there are 16 banks on a device and the warp size is 32, accessing all the banks for the threads in a full warp has a bandwidth of 32 bits per two clock cycles.

Bank conflicts arise when multiple threads from the same half-warp access the same bank. The warp is then serialized, and every thread that accesses that bank is then executed sequentially. No bank conflicts can arise between threads that belong to the first- and second half warp respectively. The hardware splits a memory request that has bank conflicts into as many separate conflict-free requests as necessary, decreasing the effective bandwidth by a factor equal to the number of separate memory requests. There is one exception: when all threads in a half warp address the same shared memory location, the value is *broadcasted*. To minimize bank conflicts, it is important to understand how memory addresses map to memory banks and how memory requests should be optimally scheduled in a specific implementation. For example, consider the following listing:

```
__shared__ char shared[32];
           char *data = shared[threadId];
```

In here, `shared[32]` is an array of bytes in the shared memory and every thread has to access its own byte by its `threadId`. However, this example will generate a 4-way bank conflict, because only 4 bytes fit in one 32-bit bank. The warp scheduler will now serialize the accesses, degrading performance of the effective bandwidth by a factor 4. The bank conflict in this example can be solved by multiplying the indexes with a factor 4:

```
char *data = shared[threadId*4];
```

Now every thread of the half-warp accesses its corresponding word. To use the speed of the shared memory for this research, the buffer where the input is stored, before it is passed to the MD5-compression function, is declared like this:

```
__shared__ unsigned int shared[THREADS_PER_BLOCK][16];
           unsigned int *buffer = shared[threadId];
```

Every thread in a block has now access to its own part of the shared memory. However, as explained by the previous example, this implementation will lead to a 16-way bank conflict (which is very inefficient): thread 0 accesses shared[0][0] (address 0) which is mapped to bank 0, thread 1 accesses shared[1][0] (address 16) which mapped to bank 0,..., thread 15 accesses shared[15][0] (address 240) which is also mapped to bank 0 ($\equiv 240 \bmod 16$). This bank conflict will cause all 16 threads of a half warp to access the first bank, which will lead to 16 serialized requests. However, this conflict can be solved entirely by declaring one extra element per thread (which is called a *stride*):

```
__shared__ unsigned int shared[THREADS_PER_BLOCK][16+1];
```

Now the addresses per thread have to be shifted with one word too:

```
unsigned int *buffer = shared[threadId]+1;
```

Now due to the modular arithmetic of the device, it will calculate the correct corresponding bank for all the threads in a half-warp: thread 0 accesses shared[0][1] (address 1) which is mapped to bank 1, thread 1 accesses shared[1][1] (address 18) which is mapped to bank 2 ($\equiv 18 \bmod 16$),..., thread 15 accesses shared[15][1] (address 256) which is mapped to bank 0 ($\equiv 256 \bmod 16$). Now all threads can access their respective banks simultaneously. However, this solution does increase the total amount of shared memory needed to run the kernel and shared memory is sparse. As a consequence, less threads per block can be configured to run concurrently on a multiprocessor, which will yield an overall loss in occupancy.

To calculate the total amount of shared memory S_{block} in bytes that is allocated for a block, the following can be used:

$$S_{block} = \text{ceil}(S_k, G_S), \tag{7.2}$$

where S_k is the amount of shared memory used by the kernel in bytes and G_S is the shared memory allocation granularity, which is equal to 512 for devices of compute capability 1.x and 128 for devices of compute capability 2.x. S_k can be derived by taking the product of the shared memory needed to run one thread and the number threads in a block.

If we want to decrease the local memory usage to zero and remove all the bank conflicts, our implementation needs the following data structures available in shared memory (assumed that the password length is n bytes):

```
__shared__ unsigned int   buffer   [THREADS_PER_BLOCK][k+1];
__shared__ unsigned int   final    [THREADS_PER_BLOCK][4+1];
__shared__ unsigned char  passwords[THREADS_PER_BLOCK][n+1];
```

In here `buffer[THREADS_PER_BLOCK][k+1]` contains the input to the MD5-compression function, k is the number of words needed to store the input calculated by $(16 - N_{null})$ (see Equation 6.2), `final[THREADS_PER_BLOCK][4+1]` is used to store the output of the MD5-compression function and `passwords[THREADS_PER_BLOCK][n+1]` contains the candidate passwords that threads have to check. So the total shared memory used by the kernel in bytes can now be calculated by:

$$S_{block} = \text{ceil}(\text{THREADS\_PER\_BLOCK} * (4k + n + 25), G_S). \tag{7.3}$$

On a device with compute capability 1.x, the maximum shared memory per multiprocessor is set to 16384 bytes. For example, if we take 256 threads per block and a password length of 8 bytes, the total shared memory needed would become $\text{ceil}(256 * (4 * 10 + 8 + 25), 512) = 18944$ bytes (which is more than the 16384 bytes available, making the kernel refuse to launch). If we take 224 threads per block, the kernel uses $\text{ceil}(224 * (4 * 10 + 8 + 25), 512) = 16384$ which is exactly the maximum possible amount.

Table 7.1 shows the effect of the stride on the number of warp serializes in our implementation when the kernel is ran once with 224 threads per block. The about 9000 warp serializes that remain after the shared memory bank conflicts are solved, have several causes: uncontrollable register bank conflicts, constant cache misses and read-after-write dependencies. However, compared to the 66 million warp serializes that exist with the bank conflicts, this number is not significant.

Registers

Theoretically, because registers have a zero clock cycle latency, it is of utmost importance to keep all the program variables in the registers whenever possible.

Stride	Number of warp serializes
No stride	66 000 000
Stride 1	9 389

Table 7.1: Effects of stride on the number of warp serializes.

Every multiprocessor stores the execution context, such as program counters and variables, of all threads in the registers for the entire lifetime of the warp. These threads are called *active threads* and have to share the available registers. Every multiprocessor has a maximum number of registers that can be used, which implies that the number of warps in a block that reside and can be processed on a given multiprocessor depends on the amount of registers (and shared memory) used by the kernel. The total number of warps in a block also depends on the compute capability of the device. It is calculated as follows[1]:

$$W_{block} = \text{ceil}(\frac{T_b}{W_{size}}, 1), \tag{7.4}$$

where T_b is the number of threads per block, W_{size} is the warp size which is equal to 32 for devices with compute capability 1.x and $\text{ceil}(x, y)$ is equal to x rounded up to the nearest multiple of y. Now, the total number of registers R_{block} allocated for a block can be calculated as follows[1]:

$$R_{block} = \text{ceil}(\text{ceil}(W_{block}, G_w) \times W_{size} \times R_k, G_T), \tag{7.5}$$

where G_w is the warp allocation granularity which is equal to 2 for devices with compute capability 1.x, R_k is the number of registers used by the kernel and G_T is the thread allocation granularity (which is equal to 256 for devices with compute capability 1.0 and 1.1, and 512 for devices with compute capability 1.2 and 1.3).

Register latency is caused by either read-after-write dependencies and register memory bank conflicts. This type of latency approximately costs 24 cycles, but can be completely hidden if the number of active threads per block is well chosen. A device with compute capability 1.x has 8 thread processors per multiprocessor and thus needs $8 * 24 = 192$ active threads (which is equal to 6 warps) to hide this latency.

The programmer has no control over which variables are placed in registers and which are spilled to local memory. The only thing a programmer can do, is to restrict the compiler in the number of registers that can be used by a kernel. The compiler and hardware thread scheduler will then schedule instructions as optimally as possible to avoid register memory bank conflicts. They achieve the best results when the number of threads per block is a multiple of 64. Other than following this rule, an application has no direct control over these bank conflicts.

Constant memory

The constant memory space originally resides in the device (global) memory and is cached in the constant cache. Because the constant memory is cached on-chip, a read from the constant memory is as fast as a read from the registers. On a cache miss, a read from the constant memory is as fast as a read from the device memory. For devices with compute capability 1.x, a request for constant memory is split into two request, one for all threads in a half-warp, which are issued independently. This request is then split into as many separate requests as there are different memory addresses in the initial request. So accesses to different addresses are thus serialized and decreasing throughput by a factor equal to the number of separate memory addresses.

In our implementation, the following variables stay the same throughout the execution of the whole kernel and can therefore be placed in the constant memory:

```
__constant__ unsigned char  salt    [8];
__constant__ unsigned char  charset [M];
__constant__ unsigned int   target  [4];
```

From these 3 variables, the threads of a warp always access the same addresses for `salt` and `target` and those requests are thus as fast as register requests. However, every thread generates a unique password based on its thread id and so multiple threads in a warp access different memory addresses of `charset` when generating the password. However, this is only done once per thread and therefore it has no significant impact on the performance.

7.2.3 Execution Configuration Optimizations

One of the major keys to performance is to keep the hardware as busy as possible. This implies that the workload should be equally shared between all the multiprocessors of a device. If the work is poorly balanced across the multiprocessors, they will deliver suboptimal performance. It is therefore important to optimize the execution configuration for a given kernel. The key concept that helps to achieve optimal performance is *occupancy*.

Occupancy is specified as the metric that determines how effectively the hardware is kept busy by looking at the active warps on a multiprocessor. This metric originates from the fact that thread instructions are executed sequentially and therefore the only way to hide latencies and keep the hardware busy when the current warp is paused or waiting for input, is to execute other warps that are available on the multiprocessor. Occupancy is therefore defined as $\frac{W_a}{W_{max}}$, where W_a is the number of active warps per multiprocessor and W_{max} is the maximum number of possible active warps per multiprocessor (which are 24, 32 and 48 for devices with compute capability 1.2, 1.3 and 2.x respectively). Occupancy

is kernel (and thus application) dependent, which means that some kernels do achieve higher performance with lower occupancy. However, low occupancy always interferes with the ability to hide memory latency, which result in a decrease of performance. On the other hand, higher occupancy does not always implies higher performance, but it may help to cover the latencies and achieve a better distribution of the workload.

The main factor to determine occupancy is thus the number of active warps per multiprocessor, which in its turn is determined by the number of thread blocks per multiprocessor and the number of threads per block. Recall from Chapter 5.3.2 that a entire blocks are executed on one multiprocessor and that the warp size is 32 for all compute capabilities. The number of active warps per multiprocessor can then be calculated as $W_a = \frac{T_a}{32}$, where T_a is the number of active threads per multiprocessor. Now to determine the optimal number threads per block T_b we have to take into account the following restrictions, which we will discuss below:

- Physical configuration limits of the GPU.
- Maximum number of available registers per multiprocessor.
- Maximum number of available shared memory per multiprocessor.

Apart from these restrictions, we will also discuss some guidelines to determine the optimal number of threads per block.

Physical configuration limits of the GPU

Every GPU has its own physical properties, described by the compute capability group the GPU belongs. The most important properties are:

- Threads per warp W_{size} (warpsize). The maximum allowed value of the number of threads that can be executed concurrently, which is equal to 32 for all compute capabilities.
- Warps per multiprocessor W_{max}. The maximum allowed value of the number of warps that can reside on one multiprocessor.
- Threads per multiprocessor. The maximum allowed value of the number of threads that can reside on one multiprocessor. This number is determined by the product of the threads per warp and the warps per multiprocessor.
- Thread blocks per multiprocessor TB_{max}. The maximum allowed value of the number of thread blocks per multiprocessor, which is equal to 8 for all compute capabilities.

Based on the previous, the number of active threads per multiprocessor T_a can be calculated as:

$$T_a = T_b * \min\left(TB_{max}, \frac{W_{max} * W_{size}}{T_b}\right) \tag{7.6}$$

Now maximum occupancy can be achieved if the active threads per multiprocessor is equal to the product of warps per multiprocessor and threads per warp:

$$T_b * \min\left(TB_{max}, \frac{W_{max} * W_{size}}{T_b}\right) = W_{size} * W_{max} \tag{7.7}$$

For example, if we do not consider the register and shared memory usage for our implementation and we use a device with compute capability 1.3, we will get the following formula for the calculation of the optimal number of threads per block T_b (and thus maximum occupancy):

$$T_b * \min\left(8, \frac{32 * 32}{T_b}\right) = 1024 \tag{7.8}$$

Figure 7.2 plots Equation 7.8 for several values of T_b and shows the influence of several block sizes on the occupancy of our kernel implementation. As becomes clear from the figure, block sizes of 128, 256, 512 and 1024 result in the maximum occupancy. However, we did not yet take the effects of shared memory and register size into consideration.

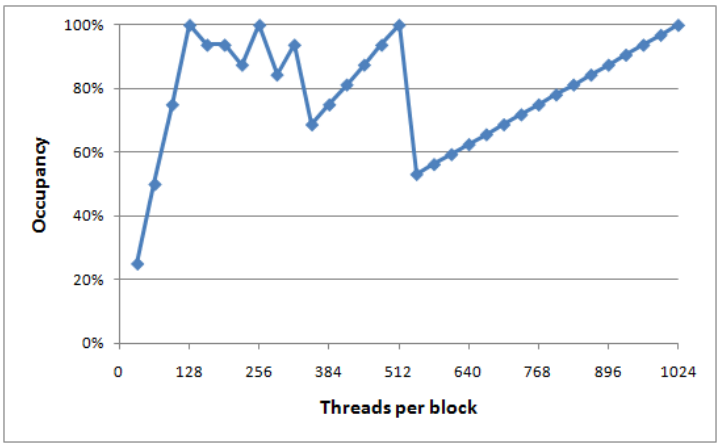

Figure 7.2: General influence of number of threads per block on the occupancy of a device with compute capability 1.3.

Maximum number of available registers per multiprocessor

Equations 7.4 and 7.5 show how the total number of registers R_{block} allocated for a thread block can be calculated. After inspection of our intermediate ptx

code, it turned out that our implementation uses 17 registers per thread. With this information we can calculate the maximum number of threads per block and find the maximum occupancy our implementation can achieve, given that we use a device with compute capability 1.3 which has 16384 32-bit registers available per multiprocessor. Now the maximum number of threads per block T_b can be calculated as:

$$T_b = \frac{R_{block}}{R_k \times W_{size}} \times W_{size} = \frac{16384}{17 \times 32} \times 32 = 960$$

Because of the ceil(x, y) function we have to round down $\frac{16384}{17 \times 32}$ to 30 and the final result then becomes 960 threads per block. With this number of threads per block, the number of active threads per multiprocessor W_a is then $960/32 = 30$ and the achieved occupancy is then $30/32 * 100\% \approx 94\%$.

Maximum number of available shared memory per multiprocessor

Equation 7.3 shows how the amount of shared memory for a given kernel can be calculated. If we use a device with compute capability 1.3, we have 16384 bytes shared memory available per multiprocessor. In the previous section we already showed that a maximum of 224 threads per block can be configured, with the assumption that the password length is 8 bytes. It turns out that the available shared memory is the bottleneck with a maximum of only 224 threads per block, where based on the physical limits and the number of available registers, a maximum number of 1024 and 960 threads per block can be configured respectively. This means that maximum occupancy for our implementation can be achieved with 224 threads per block, as shown by Figure 7.3. From the figure it becomes clear that with 224 threads per block (marked with a square), the maximum occupancy is 22%. Note that this figure looks different from Figure 7.2, because the bottleneck is now determined by the available shared memory and not by the physical configuration limits.

General guidelines

The physical configuration limits, number of registers used and the available shared memory are not the only influences on the optimal configuration setting. From several other design documents [57, 38], reference guides [1, 2] and our own experience, the following guidelines should be taken into consideration to determine the number of threads per block:

- Since the warp size for all compute capabilities is 32 threads, the number of threads per block should be a multiple of 32. This ensures that no incomplete warps are executed, which do not fully use the computing power of a multiprocessor.

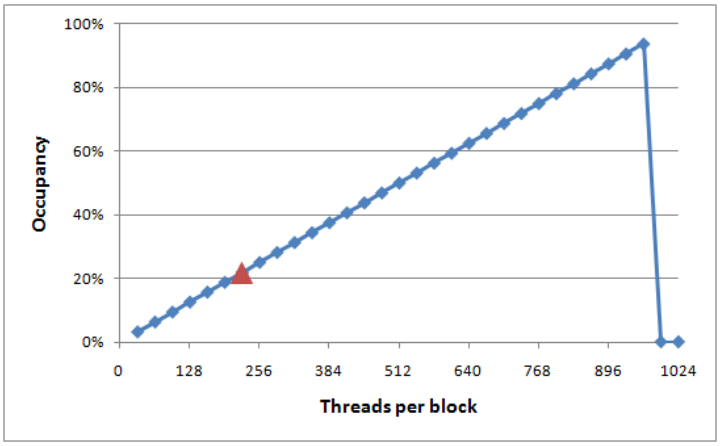

Figure 7.3: Maximum achieved occupancy for our implementation ran on a device with compute capability 1.3.

- To hide the latency for register memory bank conflicts, the optimal number of threads should be a multiple of 64.
- Since the latency of register read-after-write dependencies is approximately 24 cycles, the number of threads per block should be higher then $8 \times 24 = 192$ and $32 \times 24 = 768$ for compute capabilities 1.3 (8 thread processors per multiprocessor) and 2.x (32 thread processors per multiprocessor) respectively.
- The performance increases when the gridsize is a multiple of the number of available multiprocessors since the thread blocks are then equally divided amongst the multiprocessors.

Together with the information extracted from Figure 7.2, the most optimal number of threads per block should therefore be 256, 512 or 1024, but since our implementation is restricted by available shared memory our most optimal value is 224 threads per block, which equals to 22 % occupancy.

7.2.4 Instruction Optimizations

GPU's have an instruction set that fits their purposes for graphics calculation. However, not all instructions that execute fast on a CPU will do so on a typical GPU. Therefore, awareness of how instructions are executed often permits low-level optimizations, especially in code that is run frequently (*hot-spots*), such as the 1000 iteration for-loop in our implementation. When all high-level optimizations have been done, it may be advantageous to look at low-level instruction optimizations. The following low-level optimizations may affect our implementation:

- *Modular arithmetic* This kind arithmetic is expensive on GPU's. In the hotspot in our implementation the if-statements `if(i % 3)` and `if(i % 7)` are used. Unfortunately, we cannot rewrite these statements because 3 and 7 are not powers of 2. If they were, $(i\&(n-1))$ (where n is a power of 2) could be used, which is less expensive. The same holds for division, but since we use division only once per thread (in the password generation phase), it is not significantly beneficial to rewrite this statement. And if it was, we could still not use it since the denominator is not a power of 2 (else `(i / n)` could be written as $(i >> log_2(n))$).

- *Type conversions* This kind of arithmetic is expensive too, for example functions operating on `char` whose operands need to be converted to an `int`. In the original implementation two data structures were used to store the input to the MD5-compression function:

  ```
  unsigned char  buffer[64];
  unsigned int   buffer[16];
  ```

  The 64 `char` array was converted to a 16 `int` array, which then served as input for the MD5-compression function. However, since the elements of the array are stored sequentially in the GPU memory, it is not necessary to converge between `char` and `int`. In our current implementation only the `int buffer[16]` array is enough, because the addresses of the 16 `int` elements can be accessed byte-wise with this function:

  ```
  void setThisRef(unsigned int *a,unsigned int b, unsigned char c){
    unsigned char* ptr = (unsigned char*) a;
    *(ptr + b) = c;
  }
  ```

  While this solution does not decrease the number of instructions (because a cast from `int` to `char` is still needed), it needs less calls to shared memory (because we only need one data structure to store the input), which slow down the execution.

7.2.5 Control Flow Optimizations

Any flow control instruction (if, switch, do, for, while) can significantly affect the instruction throughput if threads of the same warp follow different execution paths, which is called *warp divergence*. If this is the case, the different execution paths must be serialized, increasing the total number of instructions executed for this warp. When all the different execution paths have completed, the threads converge back to the same execution path. This only happens when

threads have data-dependent branches, for example based on their thread-id. Algorithm-dependent (e.g. based counters or constants) branches do not cause warp divergence.

With an exhaustive search attack, threads do not have to cooperate and do not have data-dependent branches. This means that all threads in a warp can follow the same execution path, so warp divergence does not occur. Even in the main loop of our implementation, every thread in a warp executes the same instruction at the same time throughout all the 1000 iterations. Our implementation only has a data-dependent branch on the password length, so if we make sure that threads in the same warp check passwords with the same length, warp divergence does not exist.

Although all threads have the same execution path, there is still an optimization which can be used. Branching is not a native arithmetic instruction and thus multiple clock cycles are required to calculate the outcome of an `if` statement. This can be prevented by using branch prediction. In our case we have two options to achieve this:

- *Loop unrolling* By default, the compiler unrolls small loops with a known trip count. The `#pragma unroll` statement can be used to control unrolling of larger loops, even if the trip count is not known. In our case, it seems that the compiler ignores this optimization for the for loop with 1000 iterations. Even if we used the `#pragma unroll 1000` statement, the compiler decided to keep the 1000 iterations, disabling a performance increase. If we apply a loop unroll on only the first 5 iterations, the number of instructions in the ptx file doubles and since a CUDA kernel is restricted to a maximum of 2 million instructions, we think the compiler uses the total number of instructions to determine if the kernel is able to launch.

- *Loop prediction* It may be advantageous to precalculate all the given instances of our for loop, since the 1000 iterations have a certain period. Algorithm 3 shows that the input structure to the MD5-compression function is different for some iterations, based on the iteration number i. However, not all the 1000 iterations are different. For example, $i = 1$ and $i = 5$ have the same input structure: `password+salt+password+final`, where `final` is the result of last round. If we can precalculate the period, we can remove all the `if` statements. To determine the period, we need to find the least common multiple of $[2, 3, 7]$, which is 42. In total, there are 9 different input structures, which repeat themselves after 42 iterations. We can now rewrite the for loop such that all the `if` statements are removed. However, loop prediction increases the number of lower level instructions and we have to make sure that we do not exceed the maximum number of instructions per kernel, which is physically determined to 2 million for all Nvidia devices.

Chapter 8

Experimental evaluation

We have described how GPU's can speedup the performance of MD5-crypt and differentiated the optimizations. To see whether our implementation is successful, this chapter will contain the setup and outcomes of our experiments. The following three kinds of results are distinguished:

- We describe how the specific optimizations, as defined in Chapter 7, effect the performance on CUDA enabled GPU's.
- We compare the performance of known CPU implementations of MD5-crypt to ours. Moreover, we compare the performance increase against other achievements in the field (such as GPU implementations for AES and ECC).
- We measure the effects of our performance increase on contemporary password databases and password policies.

8.1 Experiment setup

8.1.1 Performance metric

To measure the performance of our implementation and to compare it against implementations on other hardware platforms, we use the real hashes per second metric. Let n be the size of the search space, then this metric is measured by counting the number of clock cycles it takes for all n threads to:

- Generate a candidate password.
- Calculate a MD5-crypt hash.
- Compare the hash against the target hash.

Then, the total number of clock cycles c is divided by the number of clock cycles a given device can execute per second C to get the execution time in seconds s. Finally, the number of hashes per second is calculated as $\frac{n}{s}$.

8.1.2 Available hardware

Although our implementation can execute on multiple GPU's, we have ran our tests only on one Nvidia GeForce 295 GTX. To compare the implementation with CPU hardware, we used an equally priced Intel i7 920 processor. The specifications of the hardware on which we have performed the experiments can be found in Appendix A.1.

8.2 Optimizations effects

To measure the impact of the optimizations described in Chapter 6 and 7 on the performance of our GPU implementation, we have set up the following experiments:

- Measure the influence of different optimizations on the performance of our implementation.
- Measure the influence of different configuration parameters on the performance of our implementation.

8.2.1 Algorithm optimizations

After we have defined a baseline (point 1), four optimizations are compared to this baseline (point 2 to 5). Then, the optimizations are combined (point 6 and 7) and again compared to the baseline.

1. *Baseline.* This is the implementation which does not use any optimization at all. Moreover, all the variables are stored in the local (i.e. global) memory and the allocation of the memory is done automatically.

2. *Constant memory.* The variables that do not change during the execution, such as target hash, salt and character set, are now stored in the constant memory. The other variables are still stored in local memory.

3. *Shared memory (bank conflicts).* The changing variables, such as the input buffer and the resulting hashes, are now stored in the fast shared memory. However, as described in Chapter 7.2.2, bank conflicts and thus warp serializes occur, which slow down the execution. The non-changing variables are still stored in the local memory.

4. *Shared memory (no bank conflicts).* The allocation of the shared memory is now done is such a way that no bank conflicts occur. With a strided access pattern every thread can access its own bank such that warp serializes are kept to a minimum, which increases the performance (at the cost of a little increase in total shared memory used).

5. *Optimized MD5.* Based on the description in Chapter 6.3.2, the MD5-compression function is statically optimized. Depending on the password length, only the necessary calculations are performed.

6. *Shared and constant memory (no bank conflicts).* Now both the changing and non-changing variables are kept in shared and constant memory respectively.

7. *Optimized MD5 with shared and constant memory.* In addition to point 6, the optimized version of the MD5-compression function is used as well.

To make a fair comparison, the configuration parameters are kept equal. The number of thread blocks (gridsize) is set to '1200' (since this is a multiple of '60', which is the number of multiprocessors on a Nvidia GTX 295) and the number of threads per block (blocksize) is set to '160' (since this is the maximum number for all optimizations to have enough shared memory available). In Figure 8.1 the performance increase per optimization is shown in black, combined optimizations are shown in gray. The figure shows us that the shared memory (without bank conflicts) and MD5-compression function optimization achieve speedups of 3 and 2 times the baseline respectively. However, when we combine both optimizations, the speedup is only a little over 3 times the baseline. This can be explained by the fact that the MD5-compression function optimization reduces the number of memory calls. While this optimization achieves a significant performance increase when all variables are stored in local memory, only little performance

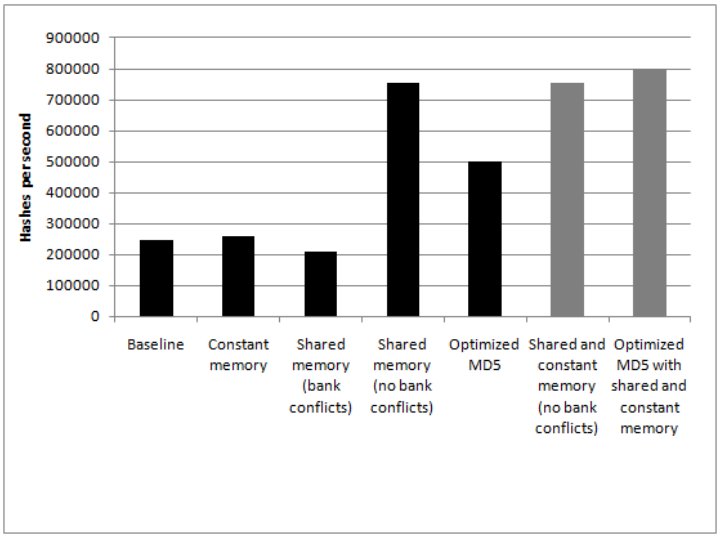

Figure 8.1: Performance increase per optimization, executed on one Nvidia GeForce GTX 295.

increase is gained when all variables are stored in shared memory (since this type of memory has low latency). Furthermore, the figure also shows that there is very little performance increase when the non-changing variables are stored in constant memory and the other variables are stored in shared memory compared to storing all the variables in shared memory (point 6 and 4 respectively). This can be explained by the fact that the compiler stores frequently used variables in the constant memory by itself and keeps a copy in local memory too (when a constant cache miss arises). Finally, the figure shows that storing variables in the shared memory while not solving the bank conflicts even degrades the performance compared to the baseline. Therefore, bank conflicts (and thus warp serializes) should be avoided.

8.2.2 Configuration optimizations

Chapter 7.2.3 shows us that it is important to keep the hardware as busy possible by maximizing the occupancy. To measure the influence of the number of threads per block (blocksize) on the performance of our most optimal implementation, we varied the number of threads per block while keeping the gridsize constant. Figure 8.2 shows the influence of the blocksize on the performance while the gridsize is set to '1200'. With the most optimal implementation and the threads per block set to 224, we are able to launch an exhaustive search with our maximum performance of approximately 880 000 hashes per second. Blocksizes up to 229 could be used as well, but did not achieve more performance than 224 threads per block. With blocksizes higher than 229, the kernel refused to run since there was not enough

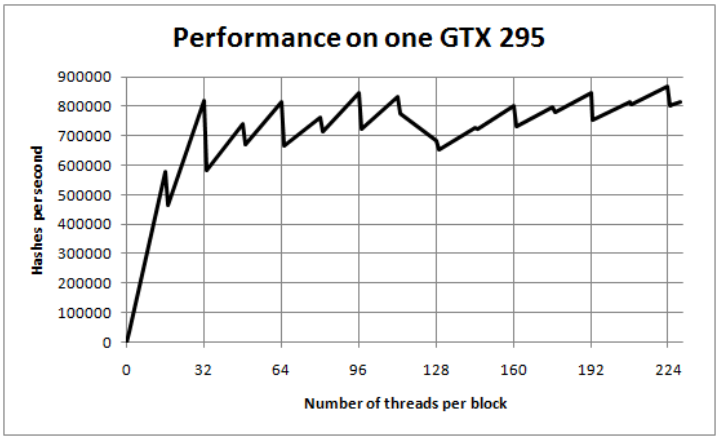

Figure 8.2: Influence of the number threads per block on the performance of our most optimal implementation, executed on one Nvidia GeForce GTX 295.

shared memory available. The graph shows a characteristic behavior: depending on the number of threads per block configured, we get stair-like graphs. Multiples of the warp size (32) and half warp size (16) result in more optimal performance. However, depending on the number of registers per thread and the amount of shared memory used, other configurations are possible and lead to smaller steps in between.

8.2.3 Comparison of theoretical limitations versus practical limitations

In the theoretical estimate of the performance we did not incorporate the candidate password generation phase and the comparison with the target hash. Since our password generation algorithm (which can be found in Appendix A.2.1) uses modular arithmetic and integer division, this influences the final performance. However, every thread only generates one password, so the time this takes may be ignored compared to the execution time of the rest of the algorithm. Figure 8.3 shows our most optimal implementation, but without the optimization of the MD5 algorithm, compared against the theoretic and practical based models (as described in Chapter 6.4.1 and 6.4.2). From the figure it becomes clear that the performance of our implementation is between the performance estimates of the SUM and MAX model. This can be explained by the fact that the warp scheduler swaps threads while they are waiting for a (shared) memory call to complete and thus higher performance can be achieved than estimated with the SUM model. Since MD5-crypt is memory intensive, memory calls can not be ignored and therefore our implementation will never reach the same level of performance as the MAX or theoretic model.

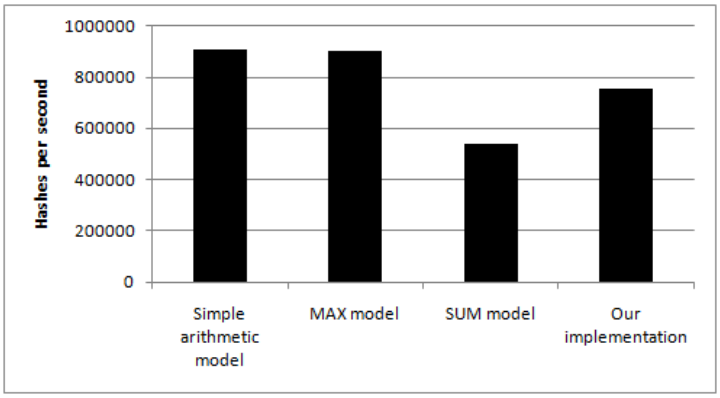

Figure 8.3: Performance of our implementation compared against the theoretic models. The SUM and MAX model displayed in the graph assume that all variables can be placed in the shared memory.

8.3 Comparison with other implementations and hardware

8.3.1 Comparison against CPU implementations

In order to compare our implementation against implementations on other hardware platforms, it is necessary to consider the costs of the hardware. We have ran our implementation on two equally priced devices: the Nvidia GeForce 295 GTX and the Intel i7 920 (both with an average price of 290 euro). However, we neglected the fact that a GPU cannot run without a host CPU. In addition, a CPU is not able to address all its resources since it has to run an operating system as well.

At the time of writing this thesis, no other MD5-crypt GPU implementations were available for testing. Therefore, we compared our implementation to the fastest MD5-crypt CPU implementation known to us, which is incorporated in the password cracker 'John the Ripper'[19]. However, John the Ripper does not have support for parallel MD5-crypt yet. As of June 2010, only parallel CPU implementations of DES and bcrypt are supported[1]. In order to fully use all four cores of the Intel i7 920, we used the OpenMP library to parallelize our CPU implementation. Figure 8.4 shows the performance of MD5-crypt implementations on GPU and CPU hardware. Our GPU implementation (about 880 000 hashes per second) achieves a speedup of 28 times over our CPU implementation (about 32 000 hashes per second) and a speedup of 104 times over the John the Ripper CPU implementation (about 8500 hashes per second). Compared to the John the Ripper implementation, we achieve a speedup of two orders of magnitude.

[1]See: http://openwall.info/wiki/john/parallelization

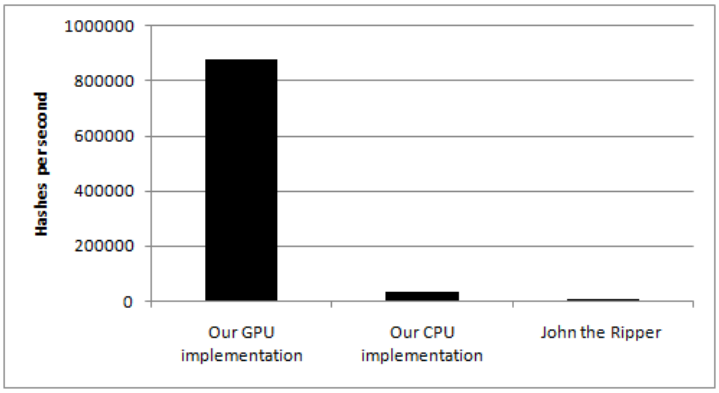

Figure 8.4: Performance comparison of different implementations on different architectures.

8.3.2 Comparison against other cryptographic implementations

GPU's have proven to be suitable to speedup the implementations of symmetric [29, 28] and asymmetric [30, 69] cryptographic functions, such as AES [12], RSA[30] and ECC[9, 8]. Table 8.1 shows the different speed ups GPU's can achieve over CPU's. Note that not all experiments were carried out on equally priced hardware, which influences the final outcome. Moreover, variations in hardware, relative newness of GPU and CPU chips, CPU implementations, and CPU technologies (e.g. the use of the SSE instruction set) will all lead to changes in the GPU / CPU run-time ratio. However, the table does show that GPU's can be efficiently used to perform cryptographic operations, or at least can act as a cryptographic 'co-processor'.

Work	Cryptographic type	Algorithm	Speed up GPU over CPU
[9, 8]	Asymmetric	ECC	4-5
[41]	Symmetric	AES	5-20
[29]	Symmetric	AES	4-10
[30]	Asymmetric	RSA	4
This work	Hashing	MD5-crypt	28

Table 8.1: Speed up GPU over CPU for different cryptographic applications.

The major difference between the speedup for hashing and the other cryptographic types comes from the fact that hash functions are solely build out of native arithmetic instructions (such as shifts, additions and logical operators) that have high throughput on a typical GPU. RSA and ECC, in contrast, are build out of multiplications and modular arithmetic, which have lower throughput than native arithmetic instructions. In addition, password hashing allows for

a maximal parallelization whereas with AES not all modes of encryption can be used for parallelization (e.g. Cipher-block Chaining or Cipher Feedback mode).

8.4 Consequences for practical use of password hashing schemes

We have shown that in the field of exhaustive searches on the password hashing scheme MD5-crypt, GPU's can significantly speed up the cracking process. To determine if this speedup is significant enough to attack systems that use such password hashing schemes, we have to see how our implementation performs on real world datasets. As example databases, we took the ones described in Chapter 4.2. To see whether the passwords are crackable in a feasible amount of time, we defined four password 'classes':

1. Passwords consisting of only lowercase ASCII characters (26 in total)
2. Passwords consisting of lowercase and numeric ASCII characters (36 in total)
3. Passwords consisting of lowercase, numeric and uppercase ASCII characters (62 in total)
4. Passwords consisting of lowercase, numeric, uppercase and special ASCII characters (94 in total)

Table 8.2 shows how the four classes are represented in the two datasets.

Dataset	% in class 1	% in class 2	% in class 3	% in class 4
phpbb.com	41	47	10	2
rockyou.com	26	59	8	7

Table 8.2: Characteristics of real world password datasets.

We can now determine which percentage of the databases could be exhaustively searched in a feasible amount of time. If we assume that all password combinations are equally likely, we can calculate the maximum length k of the passwords in each class (with c the number of base characters in the respective class) based on a given level of entropy h:

$$k = \left\lfloor log_c(2^h) \right\rfloor \tag{8.1}$$

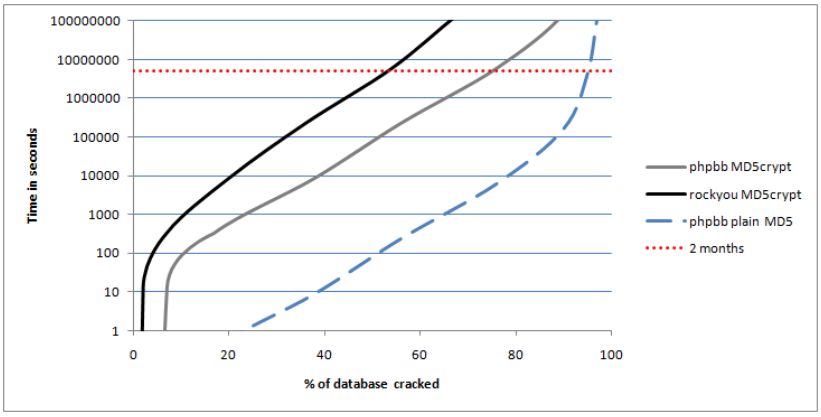

Figure 8.5: Crack times for real world datasets on one Nvidia GTX 295 with a performance of 880000 hashes per second.

Figure 8.5 then shows the amount of time it takes for one Nvidia GTX 295 to crack a percentage of the database, given that all the passwords were hashed with MD5-crypt. The figure shows us that within two months (which is the standard password expiration period in most corporate environments) of cracking time, respectively 77% and 54% of the passwords in the phpbb and rockyou database could be cracked. In addition, the figure shows that 98% of the phpbb database could be cracked if no password hashing scheme is used and the passwords are hashed with only one application of the MD5 function. Considering the fact that one cracked password can influence the security of an entire system, we consider the identified percentages as unacceptable. Moreover, given Moore's Law, even more passwords can be effectively searched in the near future. Table 8.4 shows the search times for the four password classes and some password lengths, given the performance of 880 000 hashes per second.

Length	26 characters	36 characters	62 characters	94 characters
4	0,5 Seconds	2 Seconds	16 Seconds	2 Minutes
5	13 Seconds	1 Minute	17 Minutes	2 Hours
6	5 Minutes	41 Minutes	18 Hours	10 Days
7	2 Hours	1 Days	46 Days	3 Years
8	2 Days	37 Days	8 Years	264 Years
9	71 Days	4 Years	488 Years	20647 Years
10	5 Years	132 Years	30243 Years	2480775 Years

Table 8.3: Exhaustive search times of some password lengths on one GTX 295 with a performance of 880 000 hashes per second.

To decrease the percentage of passwords that can be recovered with exhaustive searches on prevalent graphics hardware, the following methods could be used:

- Increase the entropy of the user password by enforcing a password policy.
- Increase the complexity of the password hashing scheme in such a way that one user is able to hash his password, but exhaustive searches take too much time to complete for multiple candidate passwords. This can be achieved by increasing the number of calls to the underlying hash function (e.g. by increasing the number of iterations in the key-stretching technique).

The first method increases the search space exponentially while the second method only linearly increases the time needed to iterate over the search space. Therefore, it is better to enforce users to increase the entropy in their passwords. Even so, if all users would pick passwords with high entropy, the use of password hashing schemes would be superfluous. However, if we want to increase the complexity of password hashing schemes in order to withstand exhaustive search attacks on current hardware, we could use the following rule of thumb: *One application of the password hashing scheme should not execute in less than 10 milliseconds on specialized hardware.* If we apply this rule to MD5-crypt, the number of iterations should be increased with four orders of magnitude (from 1000 to 10 000 0000 iterations).

To see whether a specific password policy is still valid given contemporary and near-future hardware, we present a model to calculate the probability P of a successful exhaustive search attack on one password p, given the password policy (determined by number of characters in the password class c and minimum password length k), the exhaustive search speed (in hashes per dollar per second) S, the value of the assets that should be protected A and the password expiry period (in seconds) E. The model can then be described as:

$$P(p) = \frac{S \times A \times E}{c^k} \tag{8.2}$$

This model is only valid under the following assumptions:

- The adversary has no knowledge about the password distribution, i.e. all candidate passwords should be randomly generated.
- All users have unique random salts.
- Users did not pick passwords that can be found (partially) in a dictionary.

Figure 8.6 shows the probability of a successful exhaustive search attack for some lengths in the most complex password class (where c is 94 and $k > 6$). With an exhaustive search speed of 2200 hashes per dollar per second and the expiry

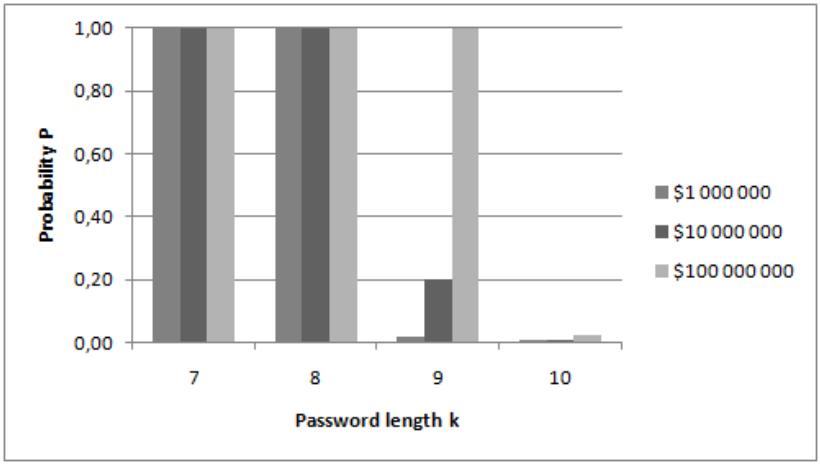

Figure 8.6: Probability of a successful exhaustive search attack for some lengths in the most complex password class.

period set to 2 months, well funded adversaries (with $A > \$10,000,000$) are able to crack all passwords with a length less than 10 characters.

This concludes the description of our experimental results. We have shown that GPU's can significantly speedup cryptographic applications such as password hashing, provided that the full power of a typical GPU can be addressed. Moreover, we have shown that either the design of password hashing schemes should be adapted to contemporary standards or users should increase the entropy in their passwords.

Chapter 9

Conclusions and future work

9.1 Conclusions

In this research, we have identified both the security properties and attacker models of prevalent password hashing schemes. In particular, we have shown that GPU's could be used for launching exhaustive search attacks on password hashing schemes that rely on the key strength of user-chosen passwords. We have shown that such attacks could be parallelized because of their embarrassingly parallel nature, which enabled us to optimize the implementation of one password hashing scheme, MD5-crypt. We have shown that our implementation approaches the theoretical speed limit on a CUDA enabled GPU. Moreover, our implementation achieves a speedup of two orders of magnitude over the best known existing CPU implementation.

We described how a general programming framework for GPU's, CUDA, could be used in combination with the specific hardware properties of a (Nvidia) GPU, such as fast shared memory and intelligent thread schedulers. Our implementation achieves a performance around 880 000 password hashes per second, whereas an equally priced CPU only achieves a performance around 30 000 password hashes per second. With this performance increase, 'complex' passwords with a length of 8 characters are now becoming feasible to crack. Moreover, we showed that between 50 % and 80 % of the passwords in a leaked database could be recovered within 2 months of computation time on one Nvidia GeForce 295 GTX. To help an organisation assess its risks related to the password policy, we proposed a model that, given the value of the protected assets, the performance of a specific hardware platform, the password policy imposed and the password expiration period, will estimate the probability of a succesfull exhaustive search attack on a password hahing scheme. To decrease the probability of success for such an attack, the complexity of prevalent password hashing schemes should be increased by at least four orders of magnitude.

In addition to the existing attacks on password hashing schemes, we identified a theoretical attack that is based on a strong collision attack on the underlying hash function (see Chapter 4.5). Therefore, one extra property should be defined for password hashing schemes that use the key-stretching technique to 'strengthen' the password. Based on the collision attacks, those schemes should hash the salt, password and result of last round in a pseudo random way for every iteration in the key-stretching phase. While the collision resistance property of the MD5 hash function has been broken, password hashing schemes based on this function, according to our findings, can still be used securely. However, the use of new hash functions, such as SHA-3, should be considered in the near future.

9.2 Discussion

Since password hashing schemes rely on cryptographic hash functions, it makes it hard to find the pre-image of a given password hash. However, most hash functions are built to execute in a very fast way, making them suitable for calculating hashes of large input domains. It is exactly this property that makes password hashing schemes vulnerable to exhaustive search attacks. For normal cryptographic functions that are based on keys (like AES) this is not a problem since the search space is of order 2^n where n is typically larger than 128. However, password hashing schemes take the user-chosen password as a 'key', which leads to a much smaller search space (order 2^n, where n is smaller than 64). To seize this problem, we considered three countermeasures:

- Either increase the complexity of the underlying cryptographic hash function or increase the number of iterations in the key-stretching technique, which both increase the complexity of the password hashing scheme linearly.
- Increase the entropy in user-chosen passwords exponentially, e.g. by implementing a password policy in the password hashing scheme that decides if the password matches a predefined entropy level.
- Focus on two-factor or other authentication mechanisms, such as zero-knowledge password proofs[7].

The first countermeasure only temporarily solves the problem since, according to Moore's law[46], hardware speed increases linearly too. Their should be a trade-off between the usability and security, as there exists a paradox that authentication needs to be fast but exhaustive searches need to be slow. Moreover, every time the number of iterations or hash function is updated, the new scheme is not backward compatible with the old passwords.

The second countermeasure seems more promising, since imposing a password policy does make users choose stronger passwords. However, because it is shown that users can not remember long passwords, they will either write them down or use their password for multiple services, which both defeat security to some extent.

The third countermeasure seems to be the most promising, but shifting in this direction requires willingness from the users and service providers in both a technical and economical way.

To put everything in perspective, passwords are only one way to ensure some level of security. If an adversary really wants a user's password, it is pretty certain that he can and will get it at some level of effort. Therefore, password hashing schemes only play the role of making some other, more illegal, way the easiest one.

9.3 Future work

This research mainly focused on the optimization of launching an exhaustive search attack on one password hashing scheme with Nvidia GPU's and the CUDA framework. As future work, it would be interesting to implement and optimize other password hashing schemes, such as SHA-crypt, bcrypt, Windows NTLM and Oracle's proprietary scheme. In addition, it would be valuable to see how Nvidia GPU's perform compared to chip sets and frameworks by other manufacturers (e.g. ATI cards that use the FireStream framework).

Our current implementation is scalable over multiple GPU's, but every GPU gets an equal share of the search space, while not every GPU has the same specifications. A load balancer for GPU's in a distributed environment is something worth looking at. In addition, it would be interesting to explore how the computing power of multiple GPU's could be addressed in a larger distributed environment, with frameworks such as BOINC[5]. Moreover, we think that this is one step towards the goal to solve complex problems in heterogeneous environments, consisting of a mix of CPU's, GPU's, mobile devices and other hardware platforms. This phenomenon is called Jungle Computing [61]. While OpenCL already provides something like a framework for this purpose, it is still a very young research field [76].

This research has focused on password hashing schemes and attacks to their security properties. While we have shown that exhaustive searches for most password keyspaces are possible, it would be interesting to see if other properties can be attacked. An example of such an attack is to use the break of the collision resistant property of MD5 to exploit the MD5-crypt scheme, given that the adversary has full control over the salt and that the salt may be arbitrary in length.

Another way to optimize implementations of cryptographic applications on GPU's is the bit slicing technique. If future graphic cards support higher register sizes, bitsliced implementations would become an option, which could increase the performance of GPU's.

Finally, as mentioned in the discussion section, it would be interesting to do a feasibility study to see whether contemporary authentication systems can be improved to manage two-factor authentication. Moreover, other studies should investigate whether or not users and service providers are willing to change their mentality regarding authentication, such that authentication does not depend on a username/password combination anymore. Identity service providers use highly qualified authentication mechanisms. An example of such a provider is the Dutch Digidentity[1], which provides authentication mechanisms for services based on verified and uniform digital identities.

[1] See: http://www.digidentity.eu/

Bibliography

[1] Compute Unified Device Architecture Programming Guide. Technical report, Nvidia Corporation, August 2010. [cited at p. 36, 40, 41, 44, 45, 46, 48, 58, 63, 66, 71, 75, 108, 112]

[2] CUDA Best practices guide. Technical report, Nvidia Corporation, August 2010. [cited at p. 63, 75]

[3] M. Abadi, T. M. A. Lomas, and R. Needham. Strengthening passwords. *SRC Technical Note*, 33, 1997. [cited at p. 29]

[4] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967. [cited at p. 65]

[5] D. P. Anderson. BOINC: A system for public-resource computing and storage. In *proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10. IEEE Computer Society, 2004. [cited at p. 94]

[6] M. Bellare and T. Kohno. Hash function balance and its impact on birthday attacks. In *Advances in Cryptology-Eurocrypt 2004*, pages 401–418. Springer, 2004. [cited at p. 9]

[7] S. M. Bellovin and M. Merritt. Encrypted key exchange: Password-based protocols secureagainst dictionary attacks. *Proceedings of the IEEE symposium on Security and Privacy*, 1992. [cited at p. 93]

[8] D. Bernstein, H. C. Chen, C. M. Cheng, T. Lange, R. Niederhagen, P. Schwabe, and B. Y. Yang. ECC2K-130 on NVIDIA GPUs. *Progress in Cryptology-INDOCRYPT 2010*, pages 328–346, 2010. [cited at p. 3, 85]

[9] D. J. Bernstein, H. C. Chen, M. S. Chen, C. M. Cheng, C. H. Hsiao, T. Lange, Z. C. Lin, and B. Y. Yang. The billion-mulmod-per-second PC. *SHARCS Workshop*, 2009. [cited at p. 3, 35, 85]

[10] D. J. Bernstein, T. R. Chen, C. M. Cheng, T. Lange, and B. Y. Yang. ECM on graphics cards. *Advances in Cryptology-EUROCRYPT 2009*, 28, 2009. [cited at p. 3, 35]

[11] E. Biham. A fast new DES implementation in software. In *Fast Software Encryption: 4th International Workshop, FSE'97, Haifa, Israel, January 1997. Proceedings*. Springer, January 1997. [cited at p. 11]

[12] J. W. Bos, D. A. Osvik, and D. Stefan. Fast Implementations of AES on Various Platforms. Technical report, Cryptology ePrint Archive, Report 2009/501, November 2009. http://eprint. iacr. org, 2009. [cited at p. 3, 85]

[13] W. E. Burr, D. F. Dodson, and W. T. Polk. Electronic authentication guideline. *NIST Special Publication*, 800:63, 2004. [cited at p. 27]

[14] L. Clair, L. Johansen, W. Enck, M. Pirretti, P. Traynor, P. McDaniel, and T. Jaeger. Password exhaustion: predicting the end of password usefulness. *Information Systems Security*, pages 37–55, 2006. [cited at p. 8]

[15] D. L. Cook, J. Ioannidis, A. D. Keromytis, and J. Luck. CryptoGraphics: Secret key cryptography using graphics cards. *Topics in Cryptology–CT-RSA 2005*, pages 334–350, 2005. [cited at p. 2]

[16] T. M. Cover, J. A. Thomas, and J. Wiley. *Elements of information theory*, volume 1. Wiley Online Library, 1991. [cited at p. 26]

[17] I. Damgård. A design principle for hash functions. *Advances in Cryptology*, CRYPTO89:416–427, 1990. [cited at p. 16]

[18] B. Den Boer and A. Bosselaers. Collisions for the compression function of MD5. In *Advances in CryptologyEurocrypt93*, pages 293–304. Springer, 1994. [cited at p. 52]

[19] S. Designer. John the Ripper password cracker, January 2011. [cited at p. 11, 84]

[20] A. Di Biagio, A. Barenghi, G. Agosta, and G. Pelosi. Design of a parallel AES for graphics hardware using the CUDA framework. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009. [cited at p. 3]

[21] Ulrich Drepper. Unix crypt using SHA-256 and SHA-512. Technical report, Akkadia, 2008. [cited at p. 5]

[22] A. J. Elbirt, W. Yip, B. Chetwynd, and C. Paar. An FPGA implementation and performance evaluation of the AES block cipher candidate algorithm finalists. In *The Third AES Candidate Conference, printed by the National Institute of Standards and Technology, Gaithersburg, MD*, pages 13–27. Citeseer, 2000. [cited at p. 12]

[23] M. J. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 2009. [cited at p. 37]

[24] I. Foster. *Designing and building parallel programs: concepts and tools for parallel software engineering*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995. [cited at p. 53]

[25] M. Garland and D. B. Kirk. Understanding throughput-oriented architectures. *Communications of the ACM*, 53(11):58–66, 2010. [cited at p. 37]

[26] C. M. Grinstead and J. L. Snell. *Introduction to probability*. Amer Mathematical Society, 1997. [cited at p. 9, 50]

[27] J. A. Halderman, B. Waters, and E. W. Felten. A convenient method for securely managing passwords. *Proceedings of the ACM*, 14th international conference on World Wide Web:471–479, 2005. [cited at p. 27]

[28] O. Harrison and J. Waldron. AES encryption implementation and analysis on commodity graphics processing units. *Cryptographic Hardware and Embedded Systems-CHES 2007*, pages 209–226, 2007. [cited at p. 3, 85]

[29] O. Harrison and J. Waldron. Practical symmetric key cryptography on modern graphics hardware. In *Proceedings of the 17th conference on Security symposium*, pages 195–209. USENIX Association, 2008. [cited at p. 3, 85]

[30] Owen Harrison and John Waldron. Efficient acceleration of asymmetric cryptography on graphics hardware. In Preneel [53], pages 350–367. [cited at p. 3, 85]

[31] M. Hellman. A cryptanalytic time-memory trade-off. *Information Theory, IEEE Transactions on*, 26(4):401–406, 2002. [cited at p. 10]

[32] G. Hu, J. Ma, and B. Huang. High Throughput Implementation of MD5 Algorithm on GPU. In *Ubiquitous Information Technologies & Applications, 2009. ICUT'09. Proceedings of the 4th International Conference on*, pages 1–5. IEEE, 2010. [cited at p. 3]

[33] Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant AES-GCM. In *Cryptographic Hardware and Embedded Systems – CHES 2009*, volume 5747 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2009. Document ID: cc3a43763e7c5016ddc9cfd5d06f8218, http://eprint.iacr.org/2009/129. [cited at p. 11]

[34] J. Katz, R. Ostrovsky, and M. Yung. Efficient password-authenticated key exchange using human-memorable passwords. *Advances in CryptologyEurocrypt 2001*, pages 475–494, 2001. [cited at p. 27]

[35] G. Kedem and Y. Ishihara. Brute force attack on UNIX passwords with SIMD computer. In *Proceedings of the 8th conference on USENIX Security Symposium-Volume 8*, page 8. USENIX Association, 1999. [cited at p. 11]

[36] J. Kelsey, B. Schneier, C. Hall, and D. Wagner. Secure applications of low-entropy keys. *Information Security*, pages 121–134, 1998. [cited at p. 30, 33]

[37] D. E. Knuth. The art of computer programming, Vol. 3, 1973. [cited at p. 14]

[38] K. Kothapalli, R. Mukherjee, MS Rehman, S. Patidar, PJ Narayanan, and K. Srinathan. A performance prediction model for the cuda gpgpu platform. In *High Performance Computing (HiPC), 2009 International Conference on*, pages 463–472. IEEE, 2010. [cited at p. 56, 75]

[39] G. Leurent. Message freedom in MD4 and MD5 collisions: Application to APOP. In *Fast Software Encryption*, pages 309–328. Springer, 2007. [cited at p. 52]

[40] C. Li, H. Wu, S. Chen, X. Li, and D. Guo. Efficient implementation for MD5-RC4 encryption using GPU with CUDA. In *Anti-counterfeiting, Security, and Identification in Communication, 2009. ASID 2009. 3rd International Conference on*, pages 167–170. IEEE, 2009. [cited at p. 3]

[41] S. A. Manavski. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In *Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on*, pages 65–68. IEEE, 2008. [cited at p. 3, 85]

[42] U. Manber. A simple scheme to make passwords based on one-way functions much harder to crack. *Computers & Security*, 15(2):171–176, 1996. [cited at p. 29]

[43] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography*. CRC, 1997. [cited at p. 8, 14, 15]

[44] N. Mentens, L. Batina, I. Verbauwhede, and Bart Preneel. Time-Memory Trade-Off Attack on FPGA Platforms: UNIX Password Cracking. *Reconfigurable Computing: Architectures and Applications*, pages 323–334, 2006. [cited at p. 10, 12, 28]

[45] R. Merkle. A certified digital signature. In *Advances in CryptologyCRYPTO89 Proceedings*, pages 218–238. Springer, 1990. [cited at p. 16]

[46] G. E. Moore et al. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998. [cited at p. 93]

[47] R. Morris and K. Thompson. Password security: A case history. *Communications of the ACM*, 22(11):594–597, 1979. [cited at p. 27]

[48] R. Mukherjee, M. S. Rehman, K. Kothapalli, PJ Narayanan, and K. Srinathan. Presenting new Speed records and constant time encryption on the GPU. [cited at p. 3]

[49] J. S. Norris and P. H. Kamp. Mission-critical development with open source software: Lessons learned. *Ieee Software*, pages 42–49, 2004. [cited at p. 52]

[50] P. Oechslin. Making a faster cryptanalytic time-memory trade-off. *Advances in Cryptology-CRYPTO 2003*, pages 617–630, 2003. [cited at p. 10, 28]

[51] L. O'Gorman. Comparing passwords, tokens, and biometrics for user authentication. *Proceedings of the IEEE*, 91(12):2021–2040, 2005. [cited at p. 25]

[52] B. Pinkas and T. Sander. Securing passwords against dictionary attacks. *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 161–170, 2002. [cited at p. 21]

[53] Bart Preneel, editor. *Progress in Cryptology - AFRICACRYPT 2009, Second International Conference on Cryptology in Africa, Gammarth, Tunisia, June 21-25, 2009. Proceedings*, volume 5580 of *Lecture Notes in Computer Science*. Springer, 2009. [cited at p. 97]

[54] Bart Preneel. Perspectives on lightweight cryptography. Inscrypt 2010, October 2010. [cited at p. 8]

[55] N. Provos and D. Mazieres. A future-adaptable password scheme. In *Proceedings of the Annual USENIX Technical Conference*. Citeseer, 1999. [cited at p. 29, 33]

[56] R. Rivest. RFC1321: The MD5 message-digest algorithm. *RFC Editor United States*, 1992. [cited at p. 16, 18, 19]

[57] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82. ACM, 2008. [cited at p. 75]

[58] Y. Sasaki and K. Aoki. Finding preimages in full MD5 faster than exhaustive search. *Advances in Cryptology-EUROCRYPT 2009*, pages 134–152, 2009. [cited at p. 52]

[59] Y. Sasaki, L. Wang, K. Ohta, and N. Kunihiro. Security of md5 challenge and response: Extension of apop password recovery attack. In *Proceedings of the 2008 The Cryptopgraphers' Track at the RSA conference on Topics in cryptology*, pages 1–18. Springer-Verlag, 2008. [cited at p. 52]

[60] B. Schneier. *Applied cryptography: protocols, algorithms, and source code in C*. A1bazaar, 2007. [cited at p. 9]

[61] F. J. Seinstra, J. Maassen, R. V. Nieuwpoort, N. Drost, T. Kessel, B. Werkhoven, J. Urbani, C. Jacobs, T. Kielmann, and H. E. Bal. Jungle Computing: Distributed Supercomputing beyond Clusters, Grids, and Clouds. *Grids, Clouds and Virtualization*, pages 167–197, 2011. [cited at p. 94]

[62] C. E. Shannon. Prediction and entropy of printed English. *Bell System Technical Journal*, 30(1):50–64, 1951. [cited at p. 27]

[63] C. E. Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55, 2001. [cited at p. 26]

[64] N. Smart. ECRYPT II yearly report on algorithms and keysizes (2009-2010). Technical report, Technical report, ECRYPT, 2010, 2010. [cited at p. 8]

[65] M. Stevens. On Collisions for MD5. Master's thesis, Eindhoven University of Technology, June 2007. [cited at p. 19]

[66] M. Stevens, A. Lenstra, and B. De Weger. Chosen-prefix collisions for MD5 and colliding X. 509 certificates for different identities. *Advances in Cryptology-EUROCRYPT 2007*, pages 1–22, 2007. [cited at p. 12, 32]

[67] M. Stevens, A. Sotirov, J. Appelbaum, A. Lenstra, D. Molnar, D. Osvik, and B. De Weger. Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. *Advances in Cryptology-CRYPTO 2009*, pages 55–69, 2009. [cited at p. 3, 12]

[68] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs Journal*, 30(3):202–210, 2005. [cited at p. 37]

[69] R. Szerwinski and T. Güneysu. Exploiting the power of GPUs for asymmetric cryptography. *Cryptographic Hardware and Embedded Systems–CHES 2008*, pages 79–99, 2008. [cited at p. 3, 35, 85]

[70] C. J. Thompson, S. Hahn, and M. Oskin. Using modern graphics architectures for general-purpose computing: A framework and analysis. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 306–317. IEEE Computer Society Press, 2002. [cited at p. 2]

[71] E. Verheul. Selecting secure passwords. *Topics in Cryptology–CT-RSA 2007*, pages 49–66, 2006. [cited at p. 23]

[72] X. Wang and H. Yu. How to break MD5 and other hash functions. *Advances in Cryptology–EUROCRYPT 2005*, pages 19–35, 2005. [cited at p. 3]

[73] T. Xie and D. Feng. How To Find Weak Input Differences For MD5 Collision Attacks. 2010. [cited at p. 10]

[74] T. Xie and Dengguo Feng. Construct md5 collisions using just a single block of message. Cryptology ePrint Archive, Report 2010/643, 2010. http://eprint.iacr.org/. [cited at p. 10]

[75] J. Yan, A. Blackwell, R. Anderson, and A. Grant. Password memorability and security: Empirical results. *Security & Privacy, IEEE*, 2(5):25–31, 2004. [cited at p. 25, 29]

[76] C. T. Yang, C. L. Huang, and C. F. Lin. Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters. *Computer Physics Communications*, 2010. [cited at p. 94]

[77] J. Yang and J. Goodman. Symmetric key cryptography on modern graphics hardware. *Advances in Cryptology–ASIACRYPT 2007*, pages 249–264, 2008. [cited at p. 2, 35]

[78] M. Zviran and W. J. Haga. Password security: an empirical study. *Journal of Management Information Systems*, 15(4):185, 1999. [cited at p. 27]

Appendices

Appendix A

Appendix A

A.1 Specifications test machine

CPU	Intel Core i7 920 (s1366,2.66GHz)
Memory	6GB DDR3 1066MHz in triple channel mode
Motherboard	ASUS P6T7 WS SuperComp (place for four GPU's)
Harddisk	3x 500GB S-ATAII
GPU	Nvidia GeForce GTX 295 (2x) and GeForce 9800 GT (1x)
Power supply	1500 watt
Operating System	Microsoft Windows server 2008 x64

A.1.1 Specifications Intel Core i7 920

The Intel Core i7 920 processor we used for this research has the following specifications.

CPU Essentials:	
Number of cores	4
Number of Threads	8
Clock Speed (MHz)	2670
Max Turbo Frequency (MHz)	2930
Smart Cache (MB)	8
Instruction Set	64-bit
Instruction Set Extensions	SSE4.2

CPU Memory specs.:	
Max Memory Size (GB)	24
Memory Types	DDR3-800/1066
Number of Memory Channels	3
Max Memory Bandwidth (GB/s)	25.6

A.1.2 Specifications Nvidia GeForce GTX 295

The Nvidia GeForce GTX 295 cards we used for this research have the following specifications.

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

The Nvidia GeForce 9800 GT card we used for this research has the following specifications.

GPU Engine Specs:	
Processor Cores	112
Graphics Clock (MHz)	600
Processor Clock (MHz)	1500
Texture Fill Rate (billion/sec)	33.6

Memory Specs:	
Memory Clock (MHz)	900
Standard Memory Config	512MB GDDR3
Memory Interface Width	256-bit (448-bit x 2)
Memory Bandwidth (GB/sec)	57.6

A.2 Code overview

All the code produced in this research can be found at http://www.martijnsprengers.eu/phKrack/. The following implementations are available:

- *GPUCrypt*. This is the main algorithm that uses the power of CUDA enabled GPU's to launch an exhaustive search on a MD5-crypt target hash, given a salt, maximum password length, base character set and GPU specific configuration parameters.
- *CPUCrypt*. This is the same algorithm as GPUcrypt, but now adapted to be used on a typical multiprocessor CPU. The parallelization is achieved with the OpenMP library (http://www.openmp.org).

A.2.1 Password generation algorithm

In both the implementations, we used the following algorithm to generate a unique candidate password based on a unique thread number. The generation is based on the base charset, password length and the startpoint, which determines the part of the search space the current GPU should traverse.

```
char* generatePassword(int threadID, char *charset, int charsetLength,
int passwordLength, int startpoint){

//Store the input
char input[passwordLength];
char buffer[passwordLength];
memset(buffer,0,passwordLength);

int base = charsetLength;
//startpoint determines share of the search space
int n = threadID+startpoint;
int index=0;
do{
   buffer[index] = n%base;
   n /= base;
   index++;
}
while(n>0);

for(int i=0; i < passwordLength; i++)
   input[i] = charset[buffer[i]];

//Set null byte
```

```
input[passwordLength] = '\0';

return input;
}
```

List of Symbols and Abbreviations

Abbreviation	Description	Definition
AES	Advanced Encryption Standard	
ALU	Arithmetic Logic Unit	
BSD	Berkeley Software Distribution	
CBC	Cipher-Block Chaining	
CFB	Cipher Feedback mode	
CPU	Central Processing Unit	
CUDA	Compute Unified Device Architecture	
DES	Data Encryption Standard	
ECC	Elliptic Curve Cryptography	
FLOP	Floating point operations per second	
FPGA	Field-programmable Gate Array	
GPU	Graphics Processing Unit	
IPS	Instructions per second	
MD5	Message-Digest Algorithm 5	
NIST	National Institute of Standards and Technology	
OpenCL	Open Computing Language	
PHS	Password Hashing Scheme	
RAM	Random-Access Memory	
RFC	Request For Comments	
RSA	Rivest, Shamir and Adleman	
SHA	Secure Hash Algorithm	
SIMD	Single Instruction Multiple Data	
SIMT	Single Instruction Multiple Threads	

List of Figures

List of Algorithms

List of Tables