Tilo Fischer
Prof. Dr. Andreas Aßmuth

Setting up a Multi-core Computing System for Cryptanalytic Problems and Evaluation of Password Cracking Algorithms

Zusammenfassung

Praktisch jeder Internetnutzer ist dazu gezwungen, täglich mit mehreren Passwörtern umzugehen. Diese sollten möglichst lange und wahllose Zeichenketten aus einem möglichst großen Zeichenvorrat sein (Groß- und Kleinbuchstaben, Ziffern und Sonderzeichen), die nicht in Wörterbüchern zu finden sind. Es ist praktisch unmöglich, sich die benötigte Anzahl von Passwörtern zu merken, die dann auch noch nach einigen Wochen oder Monaten durch neue zu ersetzen sind. Deshalb verwenden viele Internetnutzer auch heute noch einfache Passwörter oder noch schlimmer nur ein Passwort für verschiedene Dienste.
Vor diesem Hintergrund wurde unter Verwendung einer Mehrkern-Spezial-Hardware untersucht, ob, wie und wie schnell Passwörter „geknackt“ werden können.

Abstract

Every internet user has to cope with a lot of different passwords every day. These passwords should be pseudorandom strings of a suitable length with letters chosen from a preferably large character set (upper-case and lower case characters, numerals and special characters) which do not belong to any dictionary. It is practically impossible to remember all these passwords that on top of this have to be replaced after a couple of weeks or months. That is the reason why many internet users even today use weak passwords or – even worse – just one password for multiple services. In this report we present the results of our evaluation of password cracking algorithms using a special multi-core hardware.

Introduction

Passwords are often the weakest link in a modern IT security environment. But often passwords are the only – or the most comfortable way to authenticate users or programs. The problem of authentication through a secret is that the user must know the secret. We already stressed that users usually have to remember a couple of different passwords and that it is hard to learn many passwords consisting of random characters.
To compensate this people use passwords that are easy to remember. Of course, normally passwords are not stored as (readable) plain text today. And even during authentication the plain text of the passwords is not used – instead the hash value of the password comes into play which is the output of a cryptographically-secure hash function. A hash function maps any string of arbitrary length to a fixed-length output, say 512 bits, for example. A cryptographically-secure hash function needs to be a one-way function that is collision-resistant and second-preimage-resistant (cf. Menezes 1997).

If an attacker wants to intrude a system he always uses the easiest way – “the low-hanging fruits”. These “low-hanging fruits” are often passwords. If an attacker gets the password hashes, e. g. through SQL injection, leaks from other hackers, previous hacks etc., the only obstacle is to get the plain text passwords to the corresponding hash values.

For both sides in this scenario, the attacker as well as the target, it is important to know how to crack passwords and how long hash values can resist against attacks. In this report we focus on the following issues:

- Is it useful/possible to try all passwords of a given length (so called brute force attack)?
- Is it possible to exploit the missing entropy of user generated passwords?
- What is the fastest way to crack passwords?
- What is the best hardware to crack passwords (giving special attention to the Kalray MPPA 256 platform)?

To answer these questions the first step is to look at different password cracking techniques and their efficiency. The next step is to calculate statistics of passwords and analyse them. Based on this analysis it is possible to develop different password cracking tools. In the last step the speed of password cracking will be compared on different hardware.

Basics about Multi-Core Systems

Principles of Multi-Core Processor

The main problem of single-core CPUs is, you can only run one thread at the same time. With scheduling you can run as many threads as you like but you can never run threads simultaneously. The scheduler gives every thread a short time slot and after this time the operating system (OS) stops the process and saves the state of the process. In the next step the scheduler loads and executes the next thread and so on (cf. Thies 2008, p. 1 et seqq.).
Multi-core processors consist of more than one CPU, so they can run multiple threads simultaneously. To make all of these cores work, this must be supported by the OS (so called multithreading) (cf. Thies 2008, p. 1 et seqq.).

The main difference between different multi-core processor designs is their link to the memory. The best known design is Multiple Instruction Multiple Data (MIMD). MIMD means that n processors can edit n data elements with n processes (cf. Beierlein 2011 see p. 53 et seq.).

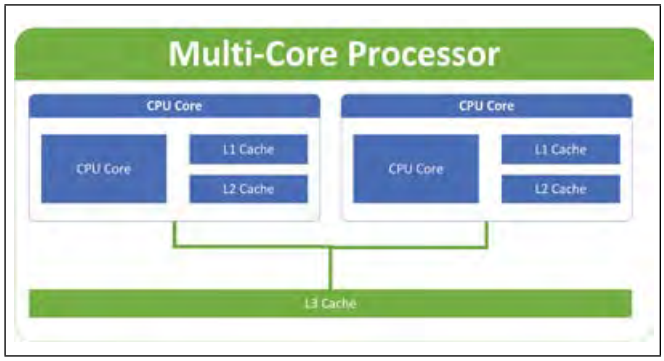

Figure 1: Multi-Core

Figure 1 shows the structure of a multi-core CPU. Every single core is connected to a dedicated L1 (level 1) and L2 (level 2) cache and a shared level 3 cache. The sizes of the caches increase from L1 to L3 but the access time decreases from L1 to L3. The cores use these caches to store data with highly frequented access
(cf. Thomas Beierlein 2011, p. 530).

GPU vs. Multi-Core

Graphics Processing Units (GPUs) are designed for real-time rendering. Every GPU consists of several processors and every processor consists of several cores (cf. figure 2). The processors are optimised for massive parallel floating point operations. In favour of the number of cores, GPUs renounce branch predictors, more cache and Instruction-Level Parallelism (ILP).

There is a major difference between the core of a CPU and the core of a GPU: On a GPU every core in one processor executes the same program with different data (so called Single Program Multiple-Data, abbreviated SPMD). All instructions on the cores in one processor run synchronously. That is called Single-Instruction Multiple Threads (SIMT). In contrast to GPUs the CPU cores are more powerful and independent of the type of their task. This implies that GPUs are much more powerful than CPUs for tasks that one can parallelise well by executing the same program with multiple data. That is due to the much higher number of cores on the GPU in relation to a multi-core processor. It must be kept in mind during the software development process that the execution time for "if else" instructions will be summed up, due to SIMT. Multi-core systems show their strength in this point. To put it in a nutshell: very complex programs run very fast on multi-core processors and program code with different threads runs very fast on CPUs (cf. Dinkla 2013, p. 136 et seqq.).

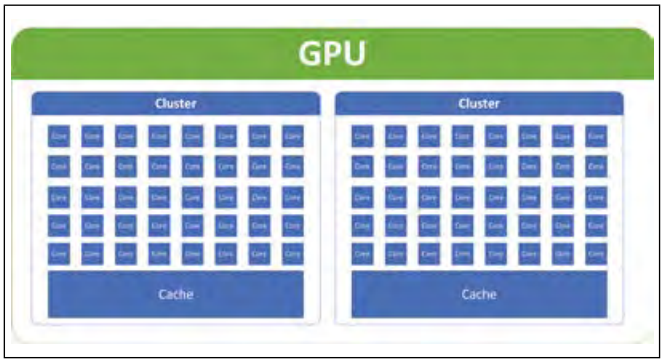

Figure 2: GPU

Kalray

Kalray is a French company which produces the "Multi-Purpose Processor Array (MPPA) MANYCORE". The corresponding developer station is called "MPPA Developer" (cf. Kalray 2012). At the Laboratory for Safe and Secure Systems (LaS^3), at the OTH Amberg-Weiden as well as at the OTH Regensburg, this hardware is used for research topics concerning scheduling, multi-core programming, safety and security.

Architecture

The before mentioned developer station consists of:

1. a MPPA-256 board,
2. a trace and debug board,

3. a motherboard with host processor (i7) and DDR3-Random-Access Memory (RAM),
4. a 500Gb HDD,
5. a special developer environment (MPPA ACCESSCORE Software Development Kit) based on the Eclipse Integrated Development Environment (IDE) and
6. a Fedora 17 host OS.

The OS and the IDE run on the i7 processor. Only the programs developed for the MPPA-256 run dedicated on the MANYCORE-processor. Figure 3 shows an overview of the architecture.

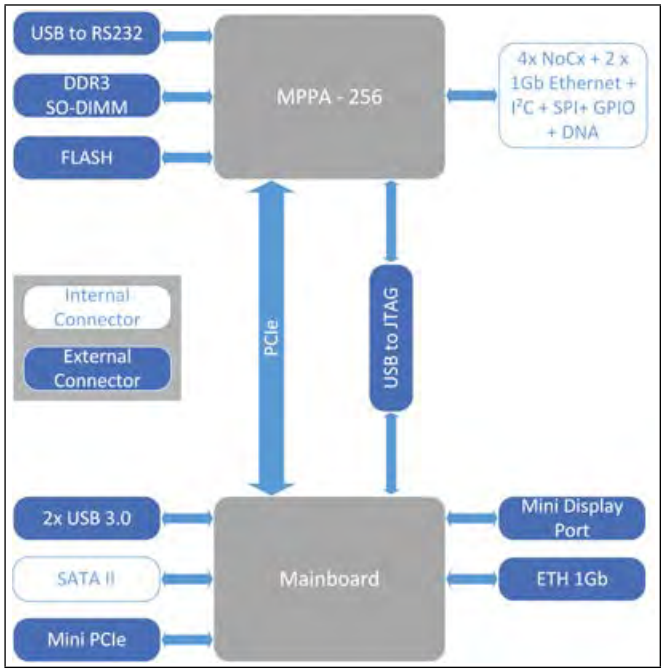

Figure 3: Kalray MPPA Developer

Two Peripheral Component Interconnect express (PCIe) GEN3 x8 interfaces combined by a x16 PCIe switch connect the MPPA-MANYCORE to the motherboard. For each PCIe x8 interface there is a dedicated quad-core processor on the MPPA-MANYCORE to handle the data transfer. Additionally, the MANYCORE is composed of 16 computing cluster with a private Floating-Point Unit (FPU) and Memory Management Unit (MMU), a smart Direct Memory Access (DMA), an own memory and a debug support unit (cf. Kalray 2013).

Every computing cluster is composed of 16 cores with the following features (cf. Kalray 2013):

1. one Branch/Control Unit,
2. two Arithmetic Logic Units,
3. one Load/Store Unit including simplified Arithmetic Logic Unit (ALU),
4. one Multiply-Accumulate (MAC) / FPU including a simplified ALU,
5. standard IEEE 754-2008 FPU with advanced Fused Multiply-Add (FMA) and dot product operators,
6. one MMU,
7. an instruction and data L1-cache and
8. 500 MHz clock rate.

The picture below (figure 4) shows the disassembled MPPA-MANYCORE:

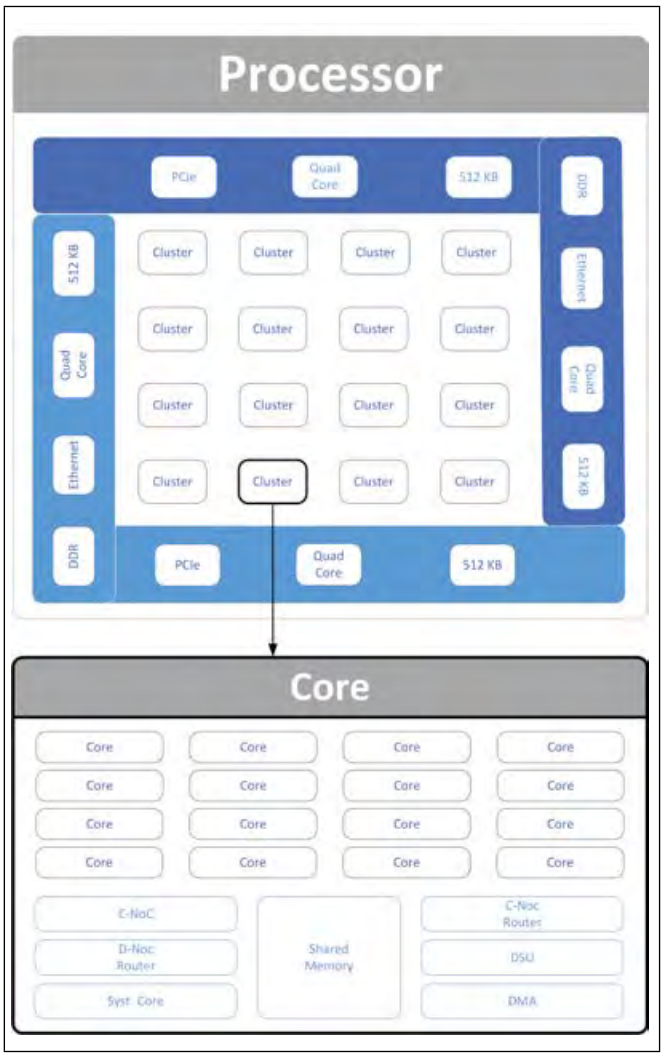

Figure 4: MPPA MANYCORE

Principles of Password Cracking

In order to learn password cracking it is necessary to understand what passwords are used for and how they are handled by computers.

Hash Function

Let X,Y be sets. A hash function maps data $x \in X$ with arbitrary size to data $y \in Y$ with a fixed size. These functions are called “one-way functions” because they are non-invertible.

$$f: X \rightarrow Y$$

Since f maps a larger set to a smaller one, a hash function is not injective. This means that there exist different preimages that are mapped to the same hash value. Such a pair of different preimages is called a collision for the

given hash function. For practical applications it must be unfeasible to find such collisions.

Password protection

To protect plaintext passwords against unauthorized access, these are always stored as hash values. To verify the user input for the password *x´*, the given input is hashed to *y´* and then compared with the saved password hash *y*.

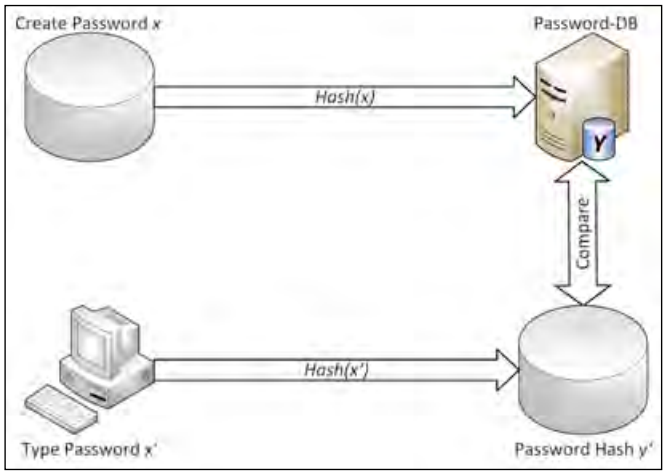

Figure 5: Save and Verify Passwords

To increase the entropy of a password a random string will be added to the password and in the next step the password hash will be generated. The random string is called salt and will be saved in plain text with the password. This increases the time for precomputation of the password hashes. However, salts are meant to make brute force attacks much harder if not unfeasible.

Rainbow tables

The default attack against hashes is to compute many hashes and compare them with the target password hash. With "rainbow tables" it is not necessary to save all hashes with all plain texts.

The best practice to save memory is to compute hash chains. A hash chain is formed by computing the hash value of the input, reducing the output with the function and taking this value to repeat the procedure.

$$input \xrightarrow{H} output \xrightarrow{R} newinput \xrightarrow{H} newoutput \xrightarrow{R} \ldots \xrightarrow{R} finaloutput$$

Calculating this chain, it is only necessary to save the *input* and the *finaloutput*. To crack passwords one has to compute many chains. If one wants to retrieve a corresponding plaintext to password x, it is necessary to compare it with all *finaloutput*. If it matches, the corresponding plaintext is used to compute the final password else the corresponding hash chain with *x* as *input* has to be built and every output must be compared with all saved values *finaloutput*. (cf. Hellmann 1980)

An optimisation of hash chains are the rainbow tables. Instead of calculating every chain with the same function *R* a set of $t - 1$ reducing functions will be used. This reduces the number of collisions in a table. With this algorithm it is only necessary to compare all *finaloutput* to detect collisions. Thus rainbow tables can be used to generate merge-free tables. This reduces the necessary storage and the computation time. (cf. Oechslin 2003)

Randomised Attacks on Passwords

The main disadvantage of rainbow tables is that they still require large amounts of data because there is no way to give distinct single values higher priorities. The needed storage grows exponentially with the maximum length of the passwords *x*. At the length of 9 characters for a password more than 1 petabyte is necessary to store the rainbow table which exceeds the capacity of state of the art computers. This is only for one salt. For every salt a new rainbow table has to be computed. Of course it is possible to compute longer chains to reduce the amount of memory but this would result in longer computation times. Since such amounts of storage are way too expensive and too much time is needed to compute these rainbow tables, a new approach is required. The following sections show some measures to reduce the memory needed. All of them exploit the low-entropy of passwords created by humans.

Dictionary Attacks

Dictionary attacks are similar to brute force attacks but do not try all possible combinations of characters. Only special words are used to crack passwords. The words are from dictionaries, plain text password lists, etc. This works because many people use the same password for different accounts and use words from dictionaries. According to Alleyne the English language contains about 1,000,000 words (cf. Alleyne 2010). To give an impression of the computing power of a state of the art computer it must be stressed that with only one computer it is possible to crack a password consisting of one of these English words and hashed with Secure Hash Algorithm 512 (cf. NIST 2015) in about 0.01s (cf. Hashcat 2013).

The downfall of this kind of attacks is that passwords which are not in the dictionary, cannot be cracked. If someone uses the same password with one different character for every account, the passwords are safe against this type of dictionary attacks. Another disadvantage is that dictionary cannot be stored as efficiently as rainbow tables. That strongly limits the maximum size of the dictionary (cf. Ziegler 2014, p. 22–25).

Markov chains

Markov Chains are used to describe a sequence of characters using a transition matrix *P*. With a list of plain text passwords it is possible to create a matrix *P* that contains the transition probability of every character from the list. Using the matrix *P* it is possible to create many passwords with a high probability.

Using Markov chains to describe passwords

The "RockYou" password list, which was used for our work, was chosen because it is the biggest password list that is publicly available (cf. Miessler 2013).

Let X be a set of passwords with cardinality $|X| = N$. If every user chose a random password of the set , it would make no sense to create a Markov chain because every symbol would occur with the probability of occurrence $1/N$ and also every Markov property would be $1/N$. Computing Markov chains only makes sense if the user does not choose actually random passwords. If Markov chains should be used for password cracking it is necessary to prove the lack of randomness for the considered passwords. We found out that based on a 8-bit character map only 214 of 256 characters are picked for passwords by the users (so only about 80 %). Therefore the passwords cannot really be random. The probability of occurrence shows that from the set of used symbols only a small proportion appears regularly. This confirms the assumption that passwords are not randomly chosen because the picked characters are not uniformly distributed.

We found two big bursts, which correspond to the numerals 0...9 and the lower case letters a...z. The lower case letters are similarly distributed like they are in English (cf. Bauer, p. 304). The reason is that the rockyou.com website is an English one with mainly English-speaking visitors. The rest of the characters are not similar to the English language.

To give a résumé: passwords are not uniformly distributed and they are not based solely on a language. Therefore, there are passwords that occur with a higher probability than others. These passwords can be described using Markov chains.

We also found out that the Markov chains differ for different password lengths. That is why it seems reasonable to compute different Markov chains for different password lengths. Finally, Markov chains differ between position transitions. But this effect decreases with the size of passwords.

Real-world attacks against cryptographic hash functions

Optimized Code for Kalray

Two different programs, both running on the Kalray Developer platform, have been developed.

Password-List

The first program computes hash values based on a pre-defined list of passwords. To crack a password it is only necessary to compare the target hash with the computed list of hashed passwords. If there is a match, it allows to calculate the position of a corresponding plain text. The concept of this tool provides one program on the host processor with different threads to read and write a file from storage, one program on "PCI0" to handle data input from the cluster, one program on "PCI1" to handle data output from the cluster and one program that will be spawned on all 16 Cluster cores.

The read thread runs on the main CPU and reads data from the storage and sends them to PCI0. The write thread also runs on the main CPU and receives all data from the program that runs on PCI1 and writes them to the storage. This construct is necessary for fast handling a huge amount of passwords.

The two threads can alternatingly read and write data and between the read/write section they can send/receive data from the cluster. For this reason it is possible to get the maximum IO-speed concerning to the HDD bottleneck. To get the maximum speed between the main CPU and the cluster, the best way is to use both PCIe interfaces. So PCI0 is used to read data from read thread and pass them on to the cluster. PCI1 is used to read data from cluster and send them to the host.

Every cluster gets only a part of every data set. The cluster main cores split these parts of the data set and send them to every core. The single cores compute the result and if all clusters have finished, the result will be returned to PCI1. Figure 6 shows the sequence diagram of the data flow:

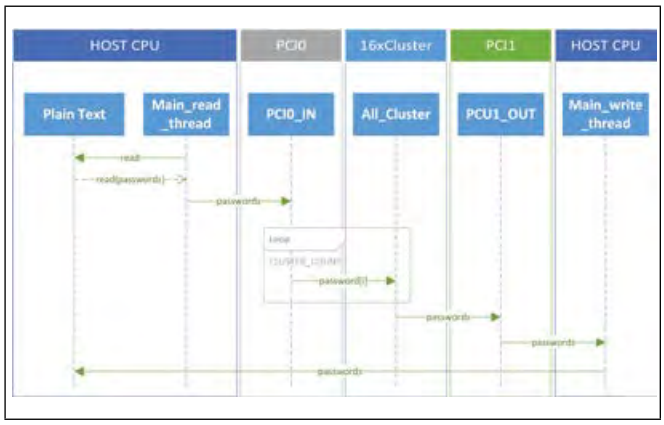

Figure 6: Sequence diagram

The architecture of the program does not depend on the password hash algorithm. There are two implementations, one for MD5 (cf. Rivest 1992) as a "best-case"[1] example, based on the short runtime. The other one uses *crypt*, the default hashing routine for Linux operating systems, with SHA512 as a real world example. *crypt* is also a "worst-case" scenario because it is very time-intensive.

The performance of those two applications is measured in

$$\frac{KiloHash}{seconds}$$

The time starts with reading the first password and stops with finishing writing. The code is written in "C" and compiled using the GNU Compiler Collection (GCC)[2] with optimization level 3 (-O3). The table below shows the average computing performance of the Kalray Developer platform.

	MD5	crypt_SHA512
kH/s	~ 1593.3	~ 2.8

The pros of the application are: The architecture of this application is not depending on the hash algorithm or the input data. It is possible to work with big password lists (bigger than the size of RAM). It is possible to pre-compute lists of hashed passwords to attack multiple passwords with the same salt.

The cons of this application are: This kind of computing password hashes works much faster on GPUs. This is due to the greater number of cores on a GPU and the fact that disadvantages like SIMT and SPMD of the GPU do not affect the result. The applications execute the same code on every core and only the input differs (correspond to SPMD) and the code does not contain jumps that are not simultaneously on every core (correspond to SIMT). Another disadvantage are the read- and write instructions on the host system which are time-intensive. The reason is that the data must be transferred to the cluster over two more OSs[3]. This is a very time-intensive process.

Markov-Cracker

This Application is based on the findings of section *"Using Markov chains to describe passwords"*.
Markov-Cracker works with Markov chains for every password length. Markov chains are used to describe the probabilities of transitions between characters and their subsequent characters. A list of start values is needed for every password length. The input data must be saved in Octave's text data format. This format allows it to easily manipulate or to create input data with Octave or MATLAB respectively.

This algorithm does not depend on the hash algorithm. The output of the application is only on the system console and shows the target hash and a corresponding plain text password.

Markov-Cracker starts with the shortest password size and loads the necessary data to the cores. The passwords are created on the fly on the cores based on the Markov chains. Due to load balancing aspects, the PCI-OS sends only one start value to every cluster. If one cluster has finished, the PCI-OS will send the next one. The start values will be distributed in descending order of probability. On the cluster every single core gets the start value from the PCI-OS and gets a unique character which is calculated with the start value and the first Markov chain. Based on these values, every core iterates the following characters according to the Markov chains. For every iteration the corresponding hash is calculated and compared to the target passwords. If one hash is equal to a target password hash, then a corresponding plain text password and the hash will be printed on the system console, else a new password will be iterated. The iteration starts with the most probable character and ends with the least probable character. This probability-depending order and load balancing ensure that every core is busy and computes the passwords with the highest probability first.

The performance of the program is again measured in

$$\frac{KiloHash}{seconds}$$

The time starts with reading the first password and stops with finishing writing. The code is completely written in "C" and compiled with the GNU Compiler Collection (GCC) with optimization level 3 (-O3).

	MD5	crypt_SHA512
kH/s	~ 132864.0	~ 2.8

In comparison to the first program, this one uses less memory. Only the size for the Markov chains and the start values are necessary. Another advantage is that the passwords only exist on the cores and it is not necessary

to transfer or save them. Finally, the most probability passwords are tested first.

A disadvantage is that Markov chains make only sense with not random, uniform distributed passwords. Also a negative aspect is that it is not possible to try specific words as a password, for example words which describe the personal background of the target user (and in this way using hints gained from social engineering).

Comparison

The main difference between Password-List and Markov-Cracker is the performance. Cracking MD5 is 82 % faster with Markov-Cracker than with Password-List. This is based on the fact that cracking on MPPA Developer is only limited through the number of available passwords per time. Markov-Cracker creates with iteration of Markov chains much more passwords per time than Password-List can load from the host storage. The performance difference between the two applications for crypt_SHA512 is vanishingly small. That is because cracking crypt_SHA512 is limited by the CPU performance and the password rate is irrelevant.

Markov-Cracker is the more powerful tool, based on the performance and the flexibility of the Markov-Chains. Password-List has one use case if personal information about the target are available from social engineering etc. which might have been used in its passwords. It is possible to try only this information.

Conclusion

Functions which create password hashes must be more than simple one-way functions with a high collision resistance. They must defy attacks using huge amount of available memory and massive parallelisation. Scrypt, for example, increases the cost of password cracking with high memory usage (cf. Percival 2012). To increase the computation time, the password will be hashed several times. Another way to increase the computation time is to salt every password, it makes rainbow tables senseless. These techniques make an attacker's job to crack the passwords much harder, but unfortunately only if the target does not use a high probability (and therefore weak) password.

Brute force attacks against passwords with characters picked from a large set of characters are too time-intensive. Rainbow tables are much faster than brute force attacks but these kinds of attacks is too slow and too memory-intensive as well. These attacks are outdated for passwords generated by humans. This is shown by current projects like Hashcat[4] or John the Ripper[5] which do not support rainbow tables.

Attacks based on password statistics are much faster because they only use relevant passwords. Attacks based on Markov chains do not need much memory and they are very effective against the most common passwords. Password lists can be used to iterate personal information or the top 1000 of the most frequently used passwords. If the password is random and the characters uniformly distributed, only rainbow tables or brute force attacks are possible. Unfortunately, people still are too careless with their passwords and that is why attacks like those based on Markov chains are more often successful than not.

To practically crack passwords, powerful hardware is necessary. GPUs are state of the art for password cracking, they allow massive parallelisation and speed up the hash calculation. Also hash algorithms with high memory usage like scrypt are no problem, because modern GPUs have much memory with high bandwidth (GDDR5, HBM). If hash functions become more complex, multi-core CPU might come more into consideration, but today multi-core CPUs are of less importance regarding password cracking. The number of cores (256) of the multi-core MPPA-256 is too low and they are also clocked with a low clock rate (400 MHz). In comparison a GPU Radeon HD 6950 (AMD) has 1408 streaming processing units (one core on a GPU) distributed (divided) on 22 processors, and a GPU clock speed of 810 MHz. The difference between the benchmarks of MPPA 256 and a benchmark with oclHashcat shows the performance difference. MD5 is approximately 2600 % faster on a GPU with oclHashcat than on MPPA-256 with Markov-Cracker and SHA512-crypt is about 600 % faster on a GPU.

Endnotes

1. From the Crackers point of view
2. https://gcc.gnu.org/
3. One on the processor behind the interface and the other on the Cluster
4. https://hashcat.net
5. http://www.openwall.com/john/

Bibliography

Alleyne, Richard. 2010. English Language Has Doubled in Size in the Last Century. http://www.telegraph.co.uk/technology/internet/8207621/English-language-has-doubled-in-size-in-the-last-century.html.

Bauer, Friedrich L. 2000. *Entzifferte Geheimnisse: Methoden und Maximen der Kryptologie.* Springer-Verlag, Heidelberg, 3 edition.

Beierlein, Thomas and Olaf Hagenbruch. 2011. *Taschenbuch Mikroprozessortechnik.* 4th ed. München: Carl Hanser Verlag.

Dinkla, Jörn. 2013. *Massiv Parallel. iX Developer*, no. 1.

Hashcat. 2013. Performance. http://hashcat.net/oclhashcat/.

Hellmann, Martin E. 1980. A cryptanalytic time-memory trad-off. IEEE Transactions On Information Theory, 26(4).

Kalray. 2012. MPPA DEVELOPER. http://www.kalrayinc.com/IMG/pdf/FLYER_MPPA_DEVELOPER.pdf.

Kalray. 2013. MPPA MANYCOR. http://www.kalrayinc.com/IMG/pdf/FLYER_MPPA_MANYCORE.pdf.

Menezes, Alfred J. and Paul C. Van Oorschot and Scott A. Vanstone and R. L. Rivest. 1997, *Handbook of Applied Cryptography.* 1st ed. CRC Press.

Miessler, Daniel. 2013. "Rockyou-Withcount.txt. https://github.com/danielmiessler/SecLists/blob/master/Passwords/rockyou-withcount.txt.

NIST. 2015. Secure Hash Standard. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf.

Oechslin, Philippe. 2003. Making a faster cryptanalytic time-memory trad-off. Proceedings of Crypto'03.

Percival, Colin. 2012. Stronger Key Derivation via Sequential Memory-Hard Functions.

Rivest, Ronald L. 1992. The MD5 Message-Digest Algorithm. https://www.ietf.org/rfc/rfc1321.txt.

Thies, Klaus-Dieter. 2008. *Echtzeit-Multitasking.* 4th ed. Aachen: Shaker Verlag GmbH.

Ziegler, Manuel. 2014. *Web Hacking: Sicherheitslücken in Webanwendungen – Lösungswege Für Entwickler.* 1st ed. München: Carl Hanser Verlag.

Kontakt:

Tilo Fischer

Ostbayerische Technische
Hochschule (OTH) Amberg-Weiden
Fakultät Elektrotechnik, Medien und Informatik
Laboratory for Safe and Secure Systems (LaS³)
Kaiser-Wilhelm-Ring 23
92224 Amberg

ti.fischer@oth-aw.de

Prof. Dr. Andreas Aßmuth

Ostbayerische Technische
Hochschule (OTH) Amberg-Weiden
Fakultät Elektrotechnik, Medien und Informatik
Laboratory for Safe and Secure Systems (LaS³)
Kaiser-Wilhelm-Ring 23
92224 Amberg

a.assmuth@oth-aw.de