FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Authentication Methods and Password Cracking
Student:	Jiří Hájek
Supervisor:	Ing. David Knap
Study program:	Informatics
Branch / specialization:	Computer Security and Information technology
Department:	Department of Computer Systems
Validity:	until the end of summer semester 2021/2022

Instructions

1. Explore available methods of user authentication (username and password, certificate, biometry...) and compare their pros and cons from standpoints of security and ease of use.
2. Focus on password authentication as the most common authentication method used nowadays. Discuss its history and how its security developed over time (e.g. deprecating ciphers). Assess future development in password security (growing computational power, quantum computers).
3. Review available methods and tools for password cracking. Discuss their specifics, performance, and parametrization options. Implement at least three different approaches to password cracking, perform experiments and tests, and discuss their results.
4. Based on knowledge gained, create a set of recommendations for penetration testers describing which password-cracking methods work best in various scenarios (depending on sample size, used cipher, etc.).

Electronically approved by prof. Ing. Pavel Tvrdík, CSc. on 18 November 2020 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Authentication Methods and Password Cracking

Jiří Hájek

Department of Computer Systems
Supervisor: Ing. David Knap

May 13, 2021

Acknowledgements

First of all, I would like to thank T-Mobile for offering me this thesis topic. I would also like to thank my supervisor, Ing. David Knap for all the support and continuous feedback he was giving me as I was writing this thesis. Last but not least, I would like to thank Microsoft Azure for quickly providing me with access to their GPU-accelerated virtual machines.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 13, 2021 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Hájek, Jiří. *Authentication Methods and Password Cracking.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstract

In the beginning of this thesis, we compare authentication methods commonly used today and dive into the history, state of the art as well as the future of password security.

Later on, we use the tool Hashcat to experiment with brute-force and dictionary attacks accelerated with Markov models and word mangling rules. We also compare two hardware approaches — regular computer and cloud computing.

Based on our findings, we finally conclude with a set of password-cracking recommendations with focus on hardware, dataset size and used hash function.

Keywords password cracking, comparison of password-cracking methods, recommendations for password cracking, brute-force attack, Markov models, dictionary attack, word mangling rules, cloud computing, Hashcat, authentication methods

Abstrakt

Na začátku této práce porovnáváme dnes běžně používané metody autentizace a také mluvíme o historii, současnosti a budoucnosti zabezpečení hesel.

Později využíváme nástroj Hashcat k experimentům s útoky hrubou silou a slovníkovými útoky, které zrychlujeme s pomocí Markovových modelů a pravidel pro manipulaci se slovy. Porovnáváme také dva hardwarové přístupy — běžný počítač a cloud computing.

Nakonec na základě našich poznatků práci uzavíráme souborem doporučení na prolamování hesel s důrazem na hardware, velikost datové sady a použitou hašovací funkci.

Klíčová slova lámání hesel, porovnání metod lámání hesel, doporučení pro lámání hesel, útok hrubou silou, Markovovy modely, slovníkový útok, pravidla pro manipulaci se slovy, cloud computing, Hashcat, autentizační metody

Contents

List of Figures

List of Tables

Introduction

There are many things in the digital world that we want only specific people to access. This is where authentication comes into play. Nowadays, most of us have something on the internet that is locked behind authentication. However, authentication in the digital world can be relatively easy to bypass and therefore opens up the possibility that someone else might access our personal data. In most cases, all that the bad guy has to do to successfully authenticate as us is put in our password.

One way of getting a particular user's password is cracking the hash of their password. There are many ways to acquire this hash. It could, for instance, leak from some compromised internet service or simply be extracted from the victim's machine.

Although it may sound a little bizarre for some at first, the good guys, sometimes known as white hat hackers, crack passwords too. Some of the motivation behind this could be, for example, catching bad guys or identifying weak passwords during penetration testing or security audits. Imagine a company, where each employee has their own company account with access to some sensitive data. We would probably not want any of that sensitive data to leak outside the company just because of weak passwords.

The goal of this thesis is to create a set of recommendations for penetration testers to help them decide what approach to take when cracking password hashes based on various parameters such as sample size or used hashing algorithm. Leading up to this main goal, there are several other subtasks, which we address in the first four chapters.

In Chapter 1, we talk about the origin of user authentication, explore the most popular authentication methods used today and compare them with focus on ease of use and security.

In Chapter 2, we focus on password authentication and security. We talk about the history, recent developments as well as where the future could be headed.

In Chapter 3, we perform an analysis of methods, tools and hardware used for cracking password hashes.

In Chapter 4, we present an implementation of several approaches to cracking password hashes and discuss the results of various experiments.

Finally, in Chapter 5, we sum up the findings of this thesis to form a set of recommendations to help penetration testers crack passwords more effectively based on the parameters of those passwords and their hashes.

Chapter 1

User authentication

In this chapter, we are going to talk about what user authentication is and why we need it by uncovering a little bit of its history. We will then discuss authentication methods commonly used today, specifically passwords, fingerprint scanning, facial recognition, public-key cryptography and one-time passwords. We will evaluate each of those methods in a separate section to familiarize the reader with the state of said authentication methods. We conclude the chapter with a brief comparison of presented authentication methods from standpoints of security and ease of use.

Let us start with the basics. According to [1], authentication is *"an act, process, or method of showing something (such as an identity, a piece of art, or a financial transaction) to be real, true, or genuine"*. This is precisely what we need in the modern age of computers to, for instance, be able to impose restricted access to resources, be it online storage or a desktop computer. However, authentication is present in the real world as well. When we vote, we have to present an identity document. When we pay with our credit card, we usually have to type our Personal Identification Number (PIN) on a terminal.

Digital authentication is believed to have started at Massachusetts Institute of Technology (MIT) in the early 1960s when Fernando Corbató helped deploy the first known computer password. There was a mainframe that used password authentication to separate user accounts and files to serve as fundamental means of privacy so that people could not just nose around in everybody's files. [2]

1.1 Authentication factors

Let us first talk about authentication factors. We usually identify three main factors of authentication — something we know, something we have and something we are [3].

1.1.1 Something we know

The idea behind this factor is a shared secret, often called a password [3]. The knowledge of this password then separates us from the others who do not know our password. To authenticate, we present the password to an authentication system, which has a way of telling whether we presented the correct password.

1.1.2 Something we have

This means using something in our possession as means of authentication. Many of us use this authentication factor daily with credit or debit cards — whoever has the card can purchase things using our money (though there is usually an additional factor in the form of a PIN).

1.1.3 Something we are

As the name suggests, this factor is based on something that a person is — both behavioral and physiological characteristics are employed, e.g., retinal scan, fingerprint, voice print, keystroke timing or signature [3]. We commonly utilize this factor in today's smartphones that often come equipped with a fingerprint scanner and facial recognition hardware.

1.2 Common authentication methods

In this section, we are going to discuss authentication methods that are commonly used today.

1.2.1 Password authentication

Passwords are everywhere. Chances are, if we create some sort of account, it is going to support password authentication and, in many cases, even require us to use it as the primary authentication factor.

Although passwords are very widely used, they do have problems. Unfortunately, users tend to choose passwords that are easy to remember, which usually means easy to guess [3]. Passwords that are hard to guess also tend to be hard to remember, which sometimes leads people to write them down, making it much easier for a potential attacker to find them [3]. In fact, [4] says 39 % of people write their passwords down. Some people create really strong passwords, but they often reuse one or a couple of them across multiple accounts since they are hard to remember. This is not a good idea, because once some service leaks some user's password, not only is that user's account on that service compromised, but all other accounts of that user (even from other services), which used the same password are likely going to be compromised — especially if they used the same username, which is often just the user's email address.

Another disadvantage of passwords is that in order to use one, we have to reveal it in a way. Even though password forms usually hide the characters we put in them, we still have to type out the password. We usually do that with keyboards, which are prone to shoulder surfing — people can simply figure out our password by seeing what keys we press in what order [3].

Sharing/distributing passwords is very easy — everyone has the ability to use the password, that is, as long as they know what the password is. Easy distribution is definitely useful, however, it can get out of control when the password is redistributed further than was intended and suddenly, people who were not meant to be able to authenticate can authenticate, because they know what the password is.

One aspect of passwords that, depending on where we live, can be a significant advantage is the fact that a password simply cannot be forcibly taken from us. If a strong enough[1] password is in place and we decide not to share it with anybody, nobody will ever be able to authenticate as us[2].

Most of the security issues of passwords that we talked about can be avoided by using a password manager, though [4] also found that only 28 % of people[3] used one. Another survey [5] got somewhat similar results — 22.5 % of respondents[4] were using a password manager. What could be even more surprising is another finding of [5] that 65 % of respondents did not trust password managers and 48 % even went on to say that nothing could motivate them to use a password manager in the future. Out of those 65 %, 34 % of respondents worried that their password manager could be hacked, while 31 % said they did not trust companies behind password managers with their information.

The author believes this lack of trust was mainly caused by the respondents not being familiar with some practices in digital security. It should be noted that when companies store encrypted data that only the user can decrypt, there is nothing for the user to worry about[5]. This is called End-to-End (E2E) encryption — the encryption key never leaves the user's device and the data is only ever encrypted/decrypted on that specific device. For instance, 1Password uses Public Key Infrastructure (PKI) to achieve just that [6]. In [6], they say *"1Password for Teams server never has a copy of the decrypted vault key and so is never in a position to share it. Only someone who has that key can encrypt a copy of it. Thus, an attack on our server could not result in unwanted sharing."*

If the user wants full control over their data, offline password managers are also an option, though they may come at a portability cost. Since the data typically does not leave the device, synchronizing or migrating passwords

[1] as in uncrackable by known attacks

[2] assuming the password cannot be obtained from anywhere but its owner's memory

[3] 1,000 Google users in the United States from ages 18 and up

[4] 1,283 Americans from ages 18 and up

[5] as long as the encryption algorithm is secure and used correctly

to another device might be a tedious process[6]. For comparison, 1Password managed to make sharing and migration of passwords seamless [7] and secure at the same time thanks to PKI.

Although there are many problems with passwords, there are reasons why they are still around. Compared to other authentication methods, they are relatively easy to implement and many solutions already exist, free for developers to use. This implies a low barrier to entry for developers. The barrier to entry is also not very high on the user side — all that is required is that the user remembers a secret phrase.

The convenience of using passwords may vary depending on how serious users are about password security. Those who rely on their memory can easily use their passwords from anywhere (e.g., home, work, school, etc.) as long as they do not forget them, while others, who might be using a password manager, either depend on a piece of software being installed on the device they are trying to authenticate on or are bound to carry around a device on which they can access their passwords.

As to what users think, passwords are just not convenient — 57 % of internet users reported they would prefer using some form of passwordless authentication [8]. This does not come as much of a surprise since about 66 % of users are more concerned with security than convenience when creating passwords [4]. As we have already discussed in this section, due to the nature of passwords, it is hard to have both security and convenience when it comes to password authentication.

1.2.2 Biometric authentication

Biometric authentication seems to be the second most common authentication method. Biometry refers to measuring and analyzing an individual's physical traits, such as fingerprints, iris patterns, or even the way a person walks [9]. Many of the modern smartphones nowadays have the ability to authenticate us with our fingerprint or our face. Therefore, it is not much of a surprise that in 2017, [10] found out 65 % of consumers[7] were already familiar with biometric authentication and 86 % were interested in using it to verify identity or to make payments. Furthermore, [11] revealed that 62 % of companies[8] were already using biometric authentication in the workplace in 2018 and that another 24 % were planning to do so by 2020.

Biometric authentication, when implemented correctly, is very convenient for users. There is no need to remember anything since the authentication is based on something that the user "is", e.g., their fingerprint, face or some other property of their body. One of the advantages biometry has over passwords is

[6] depending on abilities and extensibility of the specific password manager

[7] U.S. adult consumers who use at least one credit card, debit card and/or mobile pay

[8] based on 492 respondents from North America and Europe

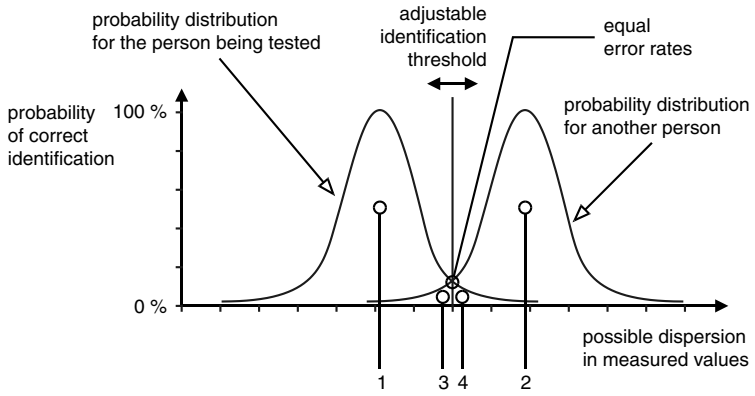

Figure 1.1: Probability distribution and decision regions for a biometric feature measurement: (1) true positive identification (true acceptance); (2) true negative identification (true rejection); (3) false positive identification (false acceptance); (4) false negative identification (false rejection). Adapted from [15].

that shoulder surfing attacks are ineffective — attackers do not gain anything from seeing us, for instance, use a fingerprint scanner or looking into a camera.

Although attackers themselves do not gain anything from seeing us authenticate using biometry, if they somehow manage to extract, let us say, our fingerprint or face scan, we get into a tough situation, because unlike in the case of passwords, we cannot easily, if at all, change something like our fingerprints or our face. This is a real threat, because research has shown that it can be done — both fingerprint readers [12] and face scanners [13, 14] have been fooled in the past. Do not forget that biometric data is something that, again, unlike passwords, can be forcibly taken from us once we are, for example, abducted by law enforcement or some bad actors.

Let us also talk about how biometric data is verified. Since the data measured by the reader/scanner are not going to be exactly the same every time, there is a system in place that decides what, let us say, fingerprint is or is not going to be accepted. The stricter the system is, the more fingerprints are going to be rejected, sometimes even the correct ones (false negatives). On the other hand, the more lenient the system is, the more fingerprints are going to be accepted, sometimes even the incorrect ones (false positives). Since we cannot reduce the amount of both false negatives and false positives, developers often strive to hit the sweet spot, where the system is both reasonably convenient and secure at the same time. See Figure 1.1 for more details. [15]

To conclude biometric authentication, it is, without a doubt, much more convenient than passwords. Although it does not beat correctly used passwords in security, it may be a step in the right direction since the majority of users simply do not use passwords correctly [4].

1.2.3 Public-key cryptography

PKI is something not nearly as many people may know about compared to passwords and biometry. However, it is still something that is relatively widely used for user/client authentication nevertheless. If nothing else, the Secure Shell (SSH) protocol, used by tools such as ssh [16] or git [17], often uses PKI, because it is considerably more secure than passwords [18].

The gist of it is an asymmetric encryption key pair consisting of two keys — a public key and a private key [18]. As the word asymmetric suggests, things encrypted with the public key can only be decrypted with the private key [19] and vice versa — things encrypted with the private key can only be decrypted with the public key [20]. As suggested by the names of the actual keys, the public key is something we share with others, while the private key is something we want to keep private at all costs [18].

Let us now talk about how this is used for authentication. First of all, the client has to give the server its public key. After that, when the client wants to log in, they are presented with a challenge that can only be solved with the corresponding private key. The solution can then be verified with the client's public key on the server because of how asymmetric cryptography works. Sometimes we use a digital certificate containing the subject's public key instead of just the public key [21]. A digital certificate is a file signed by a trusted Certificate Authority (CA) that ties a public key to a particular entity [22]. Therefore, as long as we trust the CA, we trust the certificate's claim that the public key it contains is owned by the entity it contains.

To get started with PKI, the user must first generate a key pair, which, in the author's opinion, could be a relatively big barrier to entry for the average user, who has never heard of this concept. Once the user gets past this initial setup, the authentication flow is effortless and convenient since everything happens automatically.

There is just one caveat — it is recommended to protect (encrypt) the private key with a password to increase security and prevent any breaches if the key happens to get stolen or leaked [18]. This might not even affect the convenience that much, because private keys can often be set up so that the password is only required the first time the key is used in a computer session. On the other hand, security could be at risk, because as we have learned in Section 1.2.1, most users do not practice good password habits. Still, even if the private key is not secured with a password, it will be more secure than just using a password. To exploit an unprotected private key, a potential attacker would first have to get access to the victim's machine, which, unless the victim is a high-profile[9] target, is probably not going to be worth it for the attacker.

Another thing to consider is the key algorithm being used. A widespread algorithm choice is Rivest–Shamir–Adleman (RSA), with the security factor being the key length. Various security agencies recommend keys be at least

[9] meaning they attract a lot of attention

2,048 bits long, with some, such as the National Security Agency (NSA), recommending keys of size 3,072 bits [23].

Though handling and using a key pair may prove to be difficult for the average user, the author could imagine some form of PKI authentication becoming mainstream if awareness of it were to rise, more tools to aid the average user in using it were developed and more services actually started offering PKI authentication.

1.2.4 One-time password

One-Time Password (OTP) is an authentication method that is mainly used as the second factor in Two-Factor Authentication (2FA), which has become more popular in the past couple of years. In fact, [24] found that adoption of 2FA has almost doubled between 2017 and 2019. In this section, we will focus on a specific variant of OTP — Time-Based One-Time Password (TOTP)[10], which is based on HMAC-Based One-Time Password (HOTP)[11]. According to a report from 2019 [24], as much as 19 % of us have already encountered a TOTP[12].

TOTP, as the name suggests, is a password that is generated based on the current time and can only be used once. The actual password, although it could theoretically be up to nine digits long [25], tends to be a six-digit numeric code that can only be used within a relatively short time window [26]. For optimal balance between security and usability, this time window should ideally last 30 seconds, during which the correct code may only be used once [26]. The user therefore has to wait for the next time window if they wish to reauthenticate.

A TOTP generator is uniquely identified by a hash algorithm, secret key, time-step size and length [25, 26]. From the author's experience, services usually indirectly push the user to store TOTP data on their phone using apps such as Google Authenticator[13] or Aegis[14] by encoding the necessary information into a Quick Response (QR) code and not letting them copy the embedded *otpauth://* Uniform Resource Identifier (URI) directly. Users scan the QR code to add the generator to their TOTP app/manager, then they are usually prompted to enter the currently valid code to verify that everything is working as intended and they are all set. Therefore, the barrier to entry of TOTP is relatively low since the average user should be more than capable of downloading an app to their phone and scanning a QR code.

[10] `https://www.rfc-editor.org/rfc/rfc6238.txt`

[11] `https://www.rfc-editor.org/rfc/rfc4226.txt`

[12] The report labels it as "Authenticator App (OTP)", but the author, based on their extensive exposure to many services utilizing OTP for 2FA, believes the vast majority, if not all, of those responses relate to TOTP.

[13] `https://play.google.com/store/apps/details?id=com.google.android.apps.authenticator2`

[14] `https://getaegis.app`

A quick search on Google Play[15] shows that as of April 2021, based on downloads, Google Authenticator is by far the most popular app for storing TOTP data. Out of the top 30 search results for *TOTP*, Google Authenticator had roughly twice as many downloads (50 million) as the remaining 29 apps combined (27 million). It is therefore safe to assume that, at least on Android, most people are storing TOTP data using Google Authenticator.

TOTP is great when used as an additional authentication factor. Using it as the only authentication factor might be viable, though users must be careful, because some of the mainstream mobile apps that store TOTP data, including Google Authenticator, are very open. Google Authenticator, as of April 2021, does not require any authentication to access your data when opened — one would expect to be prompted for the phone's passcode or be forced to authenticate with biometry for increased security.

The next problem with some TOTP managers, including Google Authenticator, is that once we open them (and perhaps authenticate), the authentication codes are usually all revealed, which allows for easy shoulder surfing. Yes, the codes only last for a relatively short amount of time (usually 30 seconds[16]), but shoulder surfing could still, in theory, be effective. Although the codes are only valid for one-time use within their time window, if they are all revealed at once, an attacker simply has to pick any code other than the one we are looking for at that moment.

Even when the user side is secure, implementation of the server side may become tricky when it comes to storing the parameters of TOTP generators. Unlike passwords, these parameters cannot be hashed, because the server has to read them in order to know what authentication codes it should expect from the user. To prevent potentially catastrophic data breaches, the TOTP parameters should be encrypted and the encryption key/keys stored separately in a different place. We could also let each user set their own encryption key (password) for their TOTP parameters, but at that point, it becomes 2FA, which is what TOTP is, and in the author's opinion should be, mainly used for.

A disadvantage of TOTP is that when its data, specifically parameters of the generator that yields the authentication codes, is lost, the access to the accounts, where it was used, is also lost, though the same can be said for passwords (lost password manager data) and PKI (lost private key). As a fail-safe, many services nowadays generate a set of backup codes when the TOTP is first set up. Those codes, each limited to one-time use, can then be used instead of the TOTP to authenticate. While these backup codes might be useful, the user has to store them in a secure manner, which by itself could, for some, potentially be a problem. One could just as easily store every TOTP

[15]The place, where most Android users download mobile apps. Located at `https://play.google.com/store`.

[16]though some services also accept codes from the previous time window for increased convenience, effectively doubling the size of the time window

on multiple devices at once (e.g., laptop + phone) and have that serve as a backup in case access to one of the devices is lost.

As for convenience, it depends on how the user is storing their TOTP data. From the author's experience, an extensible password manager, such as pass[17], paired with a few simple scripts and some keybindings, can bring the time needed to get the authentication code in the clipboard all the way down to about five seconds. On the opposite end of the spectrum, we would probably not be surprised if some users were annoyed by having to reach (or perhaps search) for their phone, unlock it, open an app and then manually type out a code on their computer.

1.3 Comparison

In this section, we will compare authentication methods presented in Section 1.2 from standpoints of security and ease of use. We are going to assume that the server side, which decides whether or not to authenticate users, is implemented correctly and follows best practices for maximum security. Thus, the main focus is going to be on the user side.

Passwords, when used correctly, are secure but inconvenient. Managing a password manager, being dependent on a device to be able to authenticate on the go and sometimes having to type strong passwords out all contribute to lowering the convenience of passwords. It should be noted that most users do not use passwords correctly [4, 5], which might be the reason why companies are starting to shift to biometric authentication [11].

The opposite of passwords in terms of convenience is biometric authentication. When set up correctly, it is both fast and effortless. As long as the biometric data is only stored locally and the actual authentication offloaded to something that cannot leak our biometric data, biometric authentication is, especially for the average user[18], much more secure than passwords. That way, if the server leaked all of our information, our biometric data would not be compromised. For example, we could use our fingerprint to unlock a locally-stored private key, which would then be used for the actual authentication. A big disadvantage of biometric authentication is that biometric data, unlike passwords, can be forcibly taken from us and cannot be changed as easily, if at all.

PKI is commonly used in machine-to-machine authentication since it provides great security and is harder to compromise, because it is stored on the authenticatee's machine and often protected with a password on top. Some services, such as 1Password, use it behind the scenes to improve security [6]. In the case of 1Password, their software manages everything for the user automatically, which also keeps the use of PKI relativelly convenient at the same

[17] `https://www.passwordstore.org`

[18] because the average user does not use passwords correctly

time. PKI is still widespread in some groups of people thanks to applications that depend on it, such as most things that communicate via the SSH protocol.

Overall, PKI is probably the most secure and arguably the least user-friendly/convenient authentication method out of the ones we have talked about. That is, unless it is managed by software that provides integration with third parties, which considerably increases convenience for the average user and lowers the barrier to entry. PKI is cryptographically much stronger than passwords [18] and, unlike with some implementations of biometric authentication, the server does not have to store any sensitive data — all it needs is the public key.

TOTP is great as a second authentication factor to be used alongside something else. It could theoretically be used as the only authentication factor, but that would not be very secure in most situations, especially for the average user. The convenience level highly depends on the user's setup, ranging from very convenient to very inconvenient. Authenticating on the go is the same story as with passwords, specifically password managers. We depend on a device to generate the current authentication code for us. The problem with TOTP is that the parameters of the TOTP generator must be known to the server so that it can verify the codes. If TOTP was used as the only authentication factor, this could lead to a catastrophy if there happened to be a data breach.

Chapter 2

Password authentication

In this chapter, we are going to start out by talking about the history of passwords and how their security developed over time. We will conclude the history of password security with state of the art key derivation functions (scrypt, Argon2) and then move on to discuss where the future of password security might be headed.

2.1 History

As we have briefly mentioned in Chapter 1, passwords date back to the 1960s and mark what is believed to be the beginning of authentication. It all started when Fernando Corbató helped deploy the first known computer password to prevent users from accessing each other's files [2] on a shared system called the Compatible Time-Sharing System (CTSS) [27]. This was probably also the first system to experience a data breach [27].

2.1.1 Early data breaches

Allan Scherr, at the time a PhD researcher at MIT, was using the CTSS to run simulations, which required more computing time than the 4 hours he was allocated per semester. At first, thanks to his access privileges, he was able to add an instruction to one of his programs that could reset his usage time. Once that stopped working in the spring of 1962, Scherr found a way to print out all of the passwords on the system, which allowed him to continue using more computing time than he was allocated. He only confessed nearly 25 years later. [28]

Another data breach that occured on the CTSS at MIT also happened in the early 1960s. Two system administrators were editing the password file and the daily message printed on everyone's terminal on login at the same time. Due to a software bug, the temporary files created by the editor were

interchaged, resulting in the password file being the daily message printed on login. [29]

Let us fast-forward to the late 1960s / early 1970s when the UNIX system was first implemented [30]. UNIX, just like the CTSS, was first implemented with a file that contained all users' passwords in plaintext, which had to be heavily protected. This was problematic, because system administrators had unlimited access to the passwords and were able to uncontrollably distribute them. On top of that, anyone who had physical access to backups of the system could also read the password file.

2.1.2 Password hashing

Since storing passwords in plaintext has proven to be dangerous, people have come up with password hashing. In cryptography, hash functions are one-way[19] functions, which transform input data of arbitrary size to a result of fixed size, which is called a hash [31]. Different inputs are expected to produce different outputs so it should be extremely unlikely to find a collision [31]. The first implementation of password hashing on UNIX was a symmetric encryption used in a non-tradional way [29].

2.1.3 Initial encryption

The first attempt at getting rid of plaintext passwords was using a program that simulated the M-209 cipher machine used by the U.S. Army during World War II. With a given key, inverting cipher text back into plaintext was rather trivial. On the other hand, trying to find the key for a given cipher text and plaintext was much more difficult. For that reason, the password was used as a key to encrypt a fixed plaintext and the resulting ciphertext was then used as the password's hash. [29]

This seemed to be a good approach, but it turned out the encryption was too fast — it took about 1.25 ms to encrypt a single password on hardware from that time. Experiments conducted in [29] tried to determine the typical user's password habits and found that 86 % of collected passwords could be cracked within a relatively short amount of time. A third of those 86 % were found in just 5 minutes by searching various dictionaries, while the rest could easily be cracked with a brute-force attack. [29]

2.1.4 DES

To avoid such attacks, UNIX has made several security improvements. First of all, UNIX adopted the Data Encryption Standard (DES) encryption algorithm that was announced in 1975 by the National Bureau of Standards

[19] i.e., unfeasible to invert

(NBS) [32] or as we know it today, National Institute of Standards and Technology (NIST). [29]

DES was extremely slow when implemented in software and very fast when running on a comercially available chip designed specifically to perform DES operations. To prevent the use of such chips for password cracking, the UNIX implementation of DES modified one of the internal tables used by the algorithm so that it would depend on a random 12-bit number (salt[20]). This would force attackers to build their own DES chips, which they obviously could do, but it would come at a very high cost. [29]

2.1.5 Key stretching

Key stretching is a technique used to enhance the security of a cryptographic key by transforming the initial key into an enhanced key [33]. In the case of passwords, this is sometimes achieved with hash iterations — taking a hash of a hash of a hash and so on. For example, the DES-based *crypt* algorithm described in [29] was doing 25 iterations after the initial encryption. The resulting 64 bits were then repacked to become a string of 11 printable characters — the resulting password hash.

2.1.6 Salted passwords

Together with DES, UNIX also introduced password salting [29]. Password salting is a key stretching technique that adds (usually prepends or appends) random data (salt) to the given password before it is encrypted or hashed. This salt is then stored in plaintext together with the encrypted/hashed password and applied during authentication to verify passwords.

Here, the salt was not prepended or appended to the input, but rather used as a parameter for the DES encryption instead. When a password was first entered, UNIX read the real-time clock to obtain a 12-bit random number, which was then used as the password's salt. While this modification did not make finding any individual password more difficult, it multiplied the amount of work needed to test a given string against a large collection of password hashes by up to 4,096 (2^{12}). [29]

2.1.7 Password policy

Since people behind UNIX knew that short and simple passwords were much easier to crack, they modified the password entry program to urge users to use more "obscure" passwords. It suggested alphabetic (either all uppercase or all lowercase) passwords be at least six characters long. If a larger character set was used, the recommended minimum length dropped to five characters. Although this seemed to be a step in the right direction, [29] warned that

[20]The concept of password salting is explained in Section 2.1.6.

users were not required to cooperate and even if they did, they could still use passwords that were easy to guess, such as names of their relatives or words from the dictionary. Keep in mind the maximum password length was limited to eight characters (the length of a DES key). [29]

2.1.8 Preventing user enumeration

When logging in on UNIX, both the username and password must be valid. When DES was first implemented for authentication, the encryption was done only if the username was valid. This resulted in login responses having a noticeable delay when the username was valid and no delay at all when the username was invalid. This allowed attackers to easily find out whether a particular username was valid. This flow was later modified to do the encryption even when the username was invalid to prevent such attacks. [29]

2.1.9 Growth of computing power

Let us fast forward again, this time to 1989. Ten years later after [29] wrote about adaption of DES in UNIX, it was still being used for password encryption (UNIX's *crypt* function). Because of many advances in both hardware and software, DES encryption has become very fast. In fact, [34] found out that the crypts/second/dollar[21] ratio has increased by five orders of magnitude compared to ten years ago. DES encryption was now even faster than its predecessor — the M-209 cipher on 1979 hardware. The software optimization alone led to a 102.9 times speedup[22]. If this trend were to continue, no password would soon be safe since they were limited to eight characters. [34]

2.1.10 Passphrases

The authors of [34] suggested extending the present algorithm to allow passphrases[23] that would go beyond just eight characters. Since English text has a lower bound of 1–2 bits of entropy per character, an English phrase of 5–10 words would have sufficient entropy as a passphrase. To make this usable in the UNIX's *crypt* algorithm, a hash function would be needed to fold the input into a 56-bit DES key, where each character would affect the resulting key. [34]

2.1.11 Password file shadowing

Storing all passwords in a publicly available */etc/passwd* file was a problem. With no special privileges required, anyone could extract the password hashes

[21] i.e., hashes computed by the *crypt* function per second per U.S. dollar, adjusted for inflation

[22] when compared to 4.2 Berkeley UNIX on a Sun 3/50 computer

[23] passwords consisting of multiple words, forming a phrase

from the system. To prevent the average user from accessing this file, authors of [34] suggested a technique called password file shadowing, which essentially means storing passwords in a separate file that would only be accessible by the system's administrators. This technique is still being used today on GNU/Linux and many other UNIX-based systems, often in the form of an */etc/shadow* file. [34]

2.1.12 MD5

Message Digest 5 (MD5) is a hash function first designed in 1992 by Ronald Rivest at MIT [35]. Two years later in 1994, FreeBSD introduced an MD5-based *crypt* library as an alternative to the DES-based *crypt* [36]. It enabled the use of long passwords/passphrases, used 1,000 iterations of MD5 and supported up to 48 bits long salts [36]. Most GNU/Linux systems adopted this implementation by the late 1990s [36].

Around the same time, in the mid to late 1990s, web applications became popular, which made PHP gain popularity due to its ease of use [37]. Because of cross-compatibility issues as well as U.S. export restrictions, developers started using MD5 hashes for storing passwords [37]. This was a problem, because the implementation lacked salting and did not perform any additional iterations [36].

2.1.13 SHA

Secure Hash Algorithm (SHA) is a family of cryptographic hash functions first published in 1993 by NIST. This initial version is now named SHA-0 and was followed by SHA-1 (1995), SHA-2 (2001) and SHA-3 (2015). [35]

SHA has become widely adopted over the years not just for passwords [35, 38]. SHA-1 is not that common anymore due to the increasing number of collisions [35]. Although git[24] still uses SHA-1 by default to this day[25] to identify files and revisions, there have been efforts to move away from SHA-1 in favor of a more secure algorithm [39].

On the other hand, SHA-2 hash functions, the successors of SHA-1, have remained in use up to this day. Bitcoin plus some other cryptocurrencies use SHA-256 for verifying transactions as well as calculating proof-of-stake [35] and many modern GNU/Linux systems use SHA-512 for authentication by default [37]. Other applications of SHA-2 include Transport Layer Security (TLS), Pretty Good Privacy (PGP), SSH and DomainKeys Identified Mail (DKIM) [35].

The newest SHA family, SHA-3, was not meant to replace SHA-2 [35]. It was because of successful collisions and attacks on MD5 and SHA-1 that NIST anticipated the need for a new, dissimilar hash function, which resulted

[24] `https://git-scm.com`
[25] Tested with git version 2.31.1.

in SHA-3 [35]. SHA-2 is still widely used by developers and considered cryptographically strong enough for modern applications [31]. Although there is not much incentive to switch to SHA-3, that does not mean SHA-3 is not used at all. To name a few, Ethereum[26], Monero[27] and Tor[28] all make use of SHA-3 [38].

2.1.14 PBKDF2

Back in the early 2000s, a simple Key Derivation Function (KDF) called Password-Based Key Derivation Function 2 (PBKDF2) was developed [36]. KDF is a function that aims to securely derive a key from a password. Compared to a plain hash, KDFs often come with built-in key stretching. Therefore, it should not surprise us that one of their use cases is password hashing. [31]

PBKDF2 is resistant to dictionary attacks as well as rainbow table attacks and even comes with built-in salting and an iterations count parameter. The algorithm is based on iteratively deriving Hash-based Message Authentication Code (HMAC)[29] many times with a configurable hash function. [31]

Although PBKDF2 is still approved by NIST [40], nowadays it is considered old-fashioned given that we have better options, such as bcrypt, scrypt and Argon2. The main weakness of PBKDF2 is that it is not resistant to Graphics Processing Unit (GPU) nor Application-Specific Integrated Circuit (ASIC) attacks, because it uses a relatively small amount of memory and can be efficiently implemented on previously mentioned hardware. [31]

2.1.15 bcrypt

Now let us go back even further to 1999 when [41] presented bcrypt as a future-adaptable KDF. One of the selling points of bcrypt, apart from built-in salting [31], was that it could keep up with growing computing power thanks to its variable cost [41]. This was a big advantage compared to the fixed-cost MD5-based *crypt* [41]. Unlike PBKDF2, bcrypt is both GPU-resistant and ASIC-resistant [31].

Authors of [41] claimed that bcrypt would likely remain secure 20 years into the future unless there was a major breakthrough in complexity theory — and they were right, sort of. For instance, OpenBSD and some GNU/Linux systems still use bcrypt by default for hashing passwords [37]. When hardware becomes too fast, the cost of bcrypt can simply be increased to compensate for the speedup [41]. The cost information is encoded into the hash so that

[26] `https://ethereum.org`
[27] `https://www.getmonero.org`
[28] `https://www.torproject.org`
[29] `https://www.rfc-editor.org/rfc/rfc2104.txt`

hashes with different costs remain compatible [31]. With that being said, while bcrypt is still good enough, there are better alternatives out there [31].

2.1.16 scrypt

Ten years later in 2009, a KDF called scrypt was published. Scrypt is memory-intensive and designed to prevent Field-Programmable Gate Array (FPGA), ASIC and GPU attacks with hardware highly efficient for password cracking. Like bcrypt, it also requires a salt to be used, though it is considered more secure — primarily thanks to its superior resitance to GPUs, ASICs and FPGAs. On the other hand, unlike bcrypt's single cost parameter, scrypt offers three parameters to adjust the cost — N (iterations count), r (block size) and p (parallelism factor[30]). All three of those parameters affect the usage of both the Central Processing Unit (CPU) and memory. [31]

As an example of a service that uses scrypt, let us take a look at Firebase Authentication [42]. By examining the source code[31], we can see that Firebase sets the scrypt parameters to $N = 2^{mem\_cost}$, $r = rounds$ and $p = 1$. When we create a fresh Firebase project[32] and check the Firebase Authentication settings, we see that the default values are $mem\_cost = 14$ and $rounds = 8$, which would translate to $N = 2^{14}$ and $r = 8$.

2.1.17 Argon2

Argon2 is the most recent KDF / hash function we are going to talk about. It is the winner of the Password Hashing Competition (PHC) that ran between 2013 and 2015. This was an open competition, where a group of cryptographers aimed to select one or more password hashing schemes for recommendation as an industry standard. [43]

Argon2 is resistant to hardware acceleration and is believed to be even more secure than scrypt when configured correctly. There are three main variants of Argon2:

- Argon2d — provides strong GPU resitance, but is vulnerable to potential side-channel attacks exploitable in very special situations,
- Argon2i — provides less GPU resistance, but is not vulnerable to side-channel attacks,
- Argon2id — recommended, combines Argon2d and Argon2i.

Argon2 allows for variable cost via a couple of parameters that affect the CPU and memory usage — t (number of iterations), m (amount of memory to use

[30] i.e., how many threads to run in parallel
[31] available at `https://github.com/firebase/scrypt`
[32] at `https://firebase.google.com`

in kB) and p (parallelism factor). Just like the previously mentioned KDFs, it also requires a salt to be used. [31]

Argon2 is one of the best schemes available in the industry when it comes to password hashing. It is highly secure and generally recommended over PBKDF2, bcrypt and scrypt. [31]

2.2 Future development

Thanks to the way modern KDFs such as scrypt or Argon2 are designed, advances in software and hardware can both be defeated by increasing the function's cost. We may proactively come up with better KDFs in the future, but unless there is a major breakthrough in computing, current KDFs will definitely stay with us for a while. However, there is at least one potential breakthrough in computing that we know is coming — quantum computing.

2.2.1 Quantum computing

The direct threat to post-quantum password hashing is Grover's algorithm, which, on a quantum computer, can search an unordered list for an item in $O(\sqrt{N})$, where N is the number of elements in the list [44]. As described in [45], the search requires a function $f : \{0, \ldots, N-1\} \to \{0, 1\}$, which recognizes the solution:

$$f(i) = \begin{cases} 1 & \text{if } i \text{ is the searched element} \\ 0 & \text{otherwise.} \end{cases}$$

Such function could easily be implemented with any hash function or KDF whose hash we are trying to reverse. This would mean a quadratic speedup for brute-force attacks, leading to a complexity that is orders of magnitude lower, though the attack would still run in exponential time [44]. For instance, [46] shows that brute-forcing a PBKDF2-SHA256 hash of a 10-character password would still take more than 10^7 years. They say this is because of the need for Quantum Error Correction (QEC) for deploying Grover's algorithm and that the time frame could be reduced through reduction of QEC overheads in the future.

The second potential threat to post-quantum passwords is Shor's algorithm, which brings an exponential speedup to the Discrete Logarithm Problem (DLP), the Elliptic-Curve Discrete Logarithm Problem (ECDLP) and factoring problems — algorithms widely used in cryptographic applications, such as RSA and Diffie-Hellman Key Exchange (DHKE) [44]. Since most secure protocols rely on DHKE [44], this could be a problem. Once those protocols are broken, we will no longer have a way to transfer passwords securely over the internet from the client to the server. If someone captured our traffic

while we were authenticating to a server, they would eventually be able to decrypt the data we sent, including our password.

On the other hand, [44] also says we will probably not see quantum computers capable of breaking RSA in this century and the authors of [46] say it is highly unexpected that a quantum computer capable of breaking RSA-2048 would be built within the next decade. In 2017, NIST proactively started an open competition in order to develop and standardize public-key cryptography schemes that would not be broken by quantum computers [44].

Although capable quantum computers are still relatively far, the development, standardization and deployment of post-quantum cryptography is critical to prevent a potential security and privacy disaster in the future [46].

Chapter 3

Password cracking: analysis

The goal of this chapter is to familiarize the reader with the options we have when it comes to cracking password hashes. First, we will describe methods commonly used for cracking password hashes. Then, we will go through available tools for cracking password hashes and talk about how we can use them as well as how well they perform. After that, we will discuss a couple of hardware options and finally conclude the analysis by laying out which of the methods, tools and hardware we are going to use in the implementation.

3.1 Methods

In this section, we will introduce a few popular password-cracking methods/attacks.

3.1.1 Brute-force attack

The first and probably the most straightforward method is a plain brute-force attack. This simply means testing every single possibility until we eventually arrive at the correct one.

The main drawback of this approach is that it could take a lot of time. For example, there are $62^8 \approx 2.18 \cdot 10^{14}$ possible values for a password consisting of eight alphanumeric characters[33]. For some weak hashing schemes, such as a plain MD5 hash, we could go through all possibilities in under an hour with a single GPU [47]. On the other hand, when a stronger scheme, such as bcrypt with cost even as low as 5, is used, the very same GPU would take 49.4 years to do the same thing [47].

The amount of possibilities can be significantly reduced if we know what character sets are being used in the given password, e.g., going from eight alphanumeric characters to eight lowercase characters makes the key space

[33] assuming we limit ourselves to the English alphabet

1000 times smaller and the cracking process 1000 times faster — years become hours, days become minutes.

The advantage of this approach is that by going through all possible values of a given password, we are, obviously, guaranteed to eventually crack it.

3.1.2 Dictionary attack

This method is a bit more sophisticated than plain brute force. Instead of going through every single possibility, we go through words in a dictionary. This results in a lot less password candidates that are more likely to be the password compared to random strings of characters. Since we are not testing all possibilities, we are not guaranteed to be successful. The sucess of the attack depends entirely on the dictionary.

Commonly used dictionaries consist of passwords obtained from previous data breaches. An example of this would be the RockYou dictionary from 2009 when a company called RockYou was hacked and exposed over 32 million accounts with passwords in plaintext [48].

Dictionary attacks are usually used in conjunction with word mangling — creating mutations of words from the dictionary based on a set of rules. For instance, we could have two rules — one that does nothing and another one that appends *123* to our password candidates. If our dictionary only contained the word *horse*, then after applying these rules, it would contain *horse* and *horse123*.

Word mangling is effective against passwords created by the users themselves[34], because people tend to follow certain patterns when creating passwords, e.g., capitalizing the first character, beginning with a hashtag, ending with an exclamation mark, period or a couple of numbers and many more. If the RockYou dictionary is any indication of users' habits when it comes to creating passwords, [49] found out that 75 % of passwords from RockYou were either all lowercase characters, all numbers or lowercase characters followed by numbers, though keep in mind that the only requirement of RockYou's password policy was that passwords be at least five characters long [48].

Another thing to consider with dictionaries is optimizing them with additional information about the target users. Tailoring our dictionary based on users' preferred language, country they live in, gender or age has shown to yield slightly better results than using a single dictionary for everything [50].

3.1.3 Markov models

In this section, we are going to describe what a Markov model is and how we can use it in password cracking.

[34] i.e., not randomly generated by a computer

A Markov model defines a probability distribution over sequences of symbols, i.e., it allows sampling character sequences that have certain properties. [51]

In a *zero-order* Markov model, each character is generated independently of the previously generated characters according to the underlying probability distribution. In a *first-order* Markov model, each diagram (ordered pair) of characters is assigned a probability and each character is generated by looking at the previous character. In other words, for the zero-order model,

$$P(\alpha) = \prod_{x \in \alpha} \nu(x)$$

and for the first-order model,

$$P(x_1 x_2 \dots x_n) = \nu(x_1) \prod_{i=1}^{n-1} \nu(x_{i+1}|x_i),$$

where P is the Markovian probability distribution of character sequences, x_i are individual characters and the ν function is the frequency of individual letters and diagrams in a given language. As demonstrated in [52], the ν function could also be approximated by performing frequency analysis on a large enough dictionary. [51]

Alphabetical passwords generated by humans, even when they are not dictionary words, are unlikely to be uniformly distributed in the space of alphabet sequences. In fact, an English-speaking user, when asked to pick a sequence of characters at random, will likely generate a sequence, in which each character is roughly equidistributed with the frequency of its occurrence in English text. [51]

Using Markovian filtering, we can turn an existing dictionary into a Markovian dictionary. To create such dictionary, we discretize the probabilities into two levels by applying a threshold θ. We define the zero-order dictionary as

$$\mathcal{D}_{\nu,\theta} = \{\alpha : \prod_{x \in \alpha} \nu(x) \geq \theta\}$$

and the first-order dictionary as

$$\mathcal{D}_{\nu,\theta} = \{x_1 x_2 \dots x_n : \nu(x_1) \prod_{i=1}^{n-1} \nu(x_{i+1}|x_i) \geq \theta\}.$$

Markovian dictionaries drastically reduce the size of the plausible password space by eliminating the vast majority of character sequences from consideration. [51]

Although the zero-order model produces words that do not look very natural to humans, it can already drastically reduce the size of the plausible password space by eliminating the vast majority of character sequences from consideration. [51]

The authors of [51] showed a zero-order model with a very promising keyspace compression. For a keyspace of all eight-character sequences and a θ such that the size of the resulting Markovian dictionary was $\frac{1}{7}$ of the keyspace, any sequence generated according to their model had a 90 % probability of being included in the dictionary. In other words, roughly 14 % of the password space contained 90 % of all *plausible* passwords. They also showed a first-order model, which had even better results. [51]

3.1.4 Probabilistic context-free grammars

In this section, we are going to describe what a PCFG is a how we can use it in password cracking.

Let us start with the definition of a Context-Free Grammar (CFG). A CFG is defined as $G = (V, \Sigma, S, P)$, where V is a finite set of *variables*/*non-terminals*, Σ is a finite set of *terminals*, S is the *start variable* and P is a finite set of *production rules* of the form $\alpha \to \beta$, where α is a single variable and β is a string consisting of variables or terminals. [53]

PCFG is a CFG that has probabilities associated with each production rule such that for a specific α, probabilities of all associated productions add up to one. [53]

In these grammars, in addition to the start symbol, we only use variables of the form L_n, D_n and S_n, where $n \in \mathbb{N}^+$. We call these variables *alpha variables*, *digit variables* and *special variables* respectively. Rewriting alpha variables is done using a dictionary similar to one that might be used in a traditional dictionary attack described in Section 3.1.2. [53]

A string derived from the start symbol is called a *sentential form*, which may contain both variables and terminals. The probability of this form is equal to the product of the probabilities of the productions used in its derivation. [53]

In the pre-processing phase, we derive a PCFG from our training set of passwords. See Table 3.1 for an example of such PCFG. Notice that the PCFG does not have any production rules that rewrite alpha variables. The idea is that "maximally" derived sentential forms, which only consist of terminal digits, special characters and alpha variables, define mangling rules that can be directly used in a distributed password cracking trial. For example, we could compute these mangling rules and then pass them in order of decreasing probability to a distributed system to fill in words from a dictionary. [53]

3.2 Tools

In this section, we will talk about two popular choices among the tools for password cracking — Hashcat[35] and John the Ripper (JtR)[36]. Both of them

[35] `https://github.com/hashcat/hashcat`

[36] `https://github.com/openwall/john`

α	β	Probability
S	$D_1L_3S_2D_1$	0.75
S	$L_3D_1S_1$	0.25
D_1	4	0.60
D_1	5	0.20
D_1	6	0.20
S_1	!	0.65
S_1	%	0.30
S_1	#	0.05
S_2	$$	0.70
S_2	**	0.30

Table 3.1: An example of PCFG production rules. Adapted from [53].

are free and open source. We will first discuss each of those tools separately and then compare their performance.

3.2.1 Hashcat

Hashcat claims to be the world's fastest and most advanced password recovery tool [54]. It utilizes Open Computing Language (OpenCL) and Compute Unified Device Architecture (CUDA) for GPU acceleration and additionally supports CPUs and FPGAs/DSPs/Co-Processors as well [54].

Many hash functions are included out of the box — for a complete list, refer to [54]. To crack passwords, we can choose from five attack modes — *straight*, *combination*, *brute-force*, *hybrid wordlist + mask* and *hybrid mask + wordlist* [54]. We will now talk about how each one of the attack modes works.

Straight mode is a simple dictionary attack as described in Section 3.1.2. Word mangling is supported in the form of rules [55].

Combination mode takes two dictionaries, let us call them D_1 and D_2, as an input and generates candidate passwords by appending words from D_2 to D_1. We can also specify mangling rules for each of the dictionaries by specifying the `-j` / `--rule-left` and `-k` / `--rule-right` parameters. For example, combining `yellow` from D_1 and `car` from D_2 with `-j '$-'` and `-k '$!'` would result in `yellow-car!`. [56]

Brute-force mode uses a brute-force attack as described in Section 3.1.1, which uses pre-trained per-position[37] Markov models under the hood to test

[37] i.e., it also considers positions of characters instead of just what preceeds them

the most probable candidates first, though it still searches the whole keyspace. We can turn off the character position context with the `--markov-classic` parameter or disable Markov models altogether with the `--markov-disable` parameter to approach the keyspace in a lexicographical order. To reduce the keyspace with Markov models, we can set a threshold with the `-t` / `--markov-threshold` parameter — the lower the threshold, the smaller the keyspace.

Brute-force mode also supports password masking to further reduce the searched keyspace [57]. For example, a mask `pass?l?l?l?l19?d?d` would result in the keyspace `passaaaa1900`–`passzzzz1999`. If we do not specify a mask, Hashcat will use a set of pre-defined masks and run in an incremental mode — it starts with passwords of length one and gradually increments the length as it exhausts each keyspace. We can limit the range of tested lengths by explicitly turning on the mask increment mode with `-i` / `--increment` and then setting the `--increment-min` / `--increment-max` parameters.

The idea behind the remaining *hybrid wordlist + mask* and *hybrid mask + wordlist* modes is appending (*hybrid wordlist + mask*) / prepending (*hybrid mask + wordlist*) a mask to words from a dictionary. Both of these modes can be emulated by using the straight attack mode with a set of rules corresponding to the used mask. As an example, let us say we defined a mask `?d?d?d?d` and our dictionary contained the words `password` and `hello`. In the case of *hybrid wordlist + mask*, the resulting keyspace would then be the union of `password0000`–`password9999` and `hello0000`–`hello9999`. Similarly, in the case of *hybrid mask + wordlist*, the resulting keyspace would be the union of `0000password`–`9999password` and `0000hello`–`9999hello`. [58]

3.2.2 John the Ripper

JtR is another password recovery tool. It can utilize both the CPU as well as the GPU for password cracking and supports GPU acceleration with OpenCL for some hash functions — 88 out of 495 hash modes were implemented with OpenCL as of version `1.9.0-jumbo-1`.

JtR comes with seven attack modes out of the box — *wordlist*, *single crack*, *PRINCE*, *incremental*, *mask*, *Markov*, *subsets* and *external* [59, 60].

JtR's *wordlist* mode is a simple dictionary attack that we described in Section 3.1.2. Just like in the case of Hashcat, word mangling is supported in the form of rules. [59]

Single crack mode is designed for files with General Electric Comprehensive Operating Supervisor (GECOS) fields, such as username, full name, home directory name or phone number. It uses this metadata as candidate passwords with a large set of mangling rules applied. Optionally, we can also feed this mode a dictionary with the `--single-wordlist` parameter. This will result in combinations of words from the dictionary and data from the GECOS fields. [59]

PRINCE mode is an extension of the *wordlist* mode, where we also combine words to longer and longer candidates in a clever order. Thus, a fairly small dictionary can result in a vast amount of candidates. [60]

Incremental mode searches the entire keyspace as it is esentially a brute-force attack. This mode uses trigraph frequencies to test the most probable candidates first and crack more passwords early on in the session. To use it, we either have to select one of the pre-defined incremental mode configurations or create our own. One of the pre-defined options is the `ASCII` configuration, which works with all 95 printable ASCII characters and creates candidate passwords 0–13 characters long by default [61]. Minimum and maximum length can be set with the `--min-length` and `--max-length` parameters respectively. [59]

Mask mode can be either used on its own to produce words based on the provided mask or in conjunction with other cracking modes [62]. In the latter case, the mask would be used as a form of word mangling on words generated by the other mode [62], e.g., if *incremental* mode generated the word `abc` and we used a mask `?w?l?l?l`, the candidate `abc` would be replaced with an entire keyspace of candidates `abc000`–`abc999`.

Markov mode is similar to Hashcat's brute-force mode. It uses Markov models to generate password candidates [59] and reduce the keyspace. To use it, we pass the `--markov=LEVEL` parameter, where `LEVEL` can be either a range (`MIN-MAX`) or a value of the maximum strength of passwords that are going to be tested [63]. As the maximum `LEVEL` rises, the number of passwords that are going to be tested increases exponentially [63]. If we pass the `--markov` parameter without a level, a default value specified in JtR's configuration will be used instead [63].

Subsets mode is a brute-force variant that tries to generate candidate passwords in order of complexity without resorting to advanced techniques like Markov models [64]. It aims to exploit low-entropy passwords and starts out with very long password candidates consisting of few unique characters before generating shorter ones with more unique characters [59]. This mode uses a charset of all printable ASCII characters by default, though we could also pass a custom charset with the `--subsets[=CHARSET]` parameter if we wanted to [64].

Finally, there is the *external* mode. JtR allows us to define custom modes and then use them for password cracking via the `--external` parameter. All that is required is to define the mode with a couple of functions in a subset of the C language, which JtR then compiles at startup when the mode is requested. [59]

3.2.3 Performance comparison

Let us now talk about performance. Both Hashcat and JtR come with built-in benchmarking mode, which allows us to measure the cracking speed for each

	HC-CPU	HC-GPU	JtR-CPU	JtR-GPU
MD5 (MH/s)	764	10,404	51	4,850
SHA-1 (MH/s)	348	3,501	22	2,425
sha512crypt (H/s)	3,161	48,020	4,846	38,149
bcrypt (H/s)	188	388	288	205
scrypt (H/s)	82	10	190	N/A

Table 3.2: Benchmark results of Hashcat (CPU), Hashcat (GPU), JtR (CPU) and JtR (GPU)

of their supported hash functions. We will test two weak and three strong hash functions, namely MD5, SHA-1, sha512crypt (5000 iterations), bcrypt ($cost = 10$) and scrypt ($N = 2^{15}, r = 8, p = 1$). We base those parameters on suggestions from Go's implementations of these KDFs [65, 66].

MD5, SHA-1 and sha512crypt are going to be benchmarked with the tools' built-in benchmark modes — Hashcat's `--benchmark` and JtR's `--test`. For our JtR runs, we are always going to set the `--mask` flag as it yields significant performance improvements.

With bcrypt and scrypt, things get a bit more complicated. Both tools are testing bcrypt with $cost = 5$, which is far from our target $cost = 10$, and when it comes to scrypt, the tools differ in the parameters they test it with. On top of that, these two GPU-resistant KDFs may be faster on a CPU than a GPU. For that reason, we will run our tests on both the GPU and the CPU. Since the benchmarks do not fit our needs, we will tests these KDFs with a brute-force attack with the mask `?l?l?l?l?l?l?l?l` on a single hash.

For GPU acceleration, Hashcat will be tested with CUDA, because based on the author's prior interactions, it seems to perform better in all of our cases. JtR will only be tested with OpenCL as it does not support CUDA. The JtR scrypt test will only be performed on the CPU, because JtR does not come with a GPU implementation. The tests will be run on the author's laptop, which we later describe in Section 3.4 and will be conducted with Hashcat version `6.1.1` and JtR version `1.9.0-jumbo-1`.

From the benchmark results in Table 3.2, we can see that Hashcat dominates JtR in the weak hashes. JtR starts to catch up in CPU tests with stronger hashes, but since Hashcat allows us to use both the CPU and the GPU at the same time[38], JtR still comes short in all scenarios but scrypt, where its CPU speed is two times higher than Hashcat's CPU+GPU speeds combined.

[38] at the cost of a minor GPU performance penalty

3.3 Hardware

In this section, we are going to talk about three common hardware choices for running password-cracking tools. The resulting performance may vary based on the hash function we are trying to reverse. In some cases, there could be a big performance boost when a GPU, ASIC or FPGA is used and the hash function is not resistant to acceleration provided by such hardware.

3.3.1 Regular PC

A regular Personal Computer (PC) means a consumer-grade laptop / desktop computer. Some of those PCs people have at home, especially gaming PCs, which tend to have high-end CPUs and GPUs, could perform very well in password cracking, though they are still going to be slow when compared to what cloud computing has to offer.

3.3.2 Cloud computing

Cloud computing is an on-demand access, via the internet, to computing resources hosted at a remote data center managed by a cloud service provider, who usually bills users of such resources according to usage [67]. This way, we can rent high-performance computer resources for relatively cheap compared to buying the hardware.

The bills can still get quite expensive relatively fast though. For example, on Amazon Web Services (AWS), the top-of-the-line Elastic Compute Cloud (EC2) p4d.24xlarge instance with 96 virtual CPU cores, 1152 GB of memory and eight NVIDIA A100 GPUs costs for $32.7726 per hour[39] (billed per second) [68, 69], which sums up to about $24,000 per month.

Some of the other major providers of cloud computing resources are Microsoft Azure and Google Cloud, both of which, among other things, rent out virtual machines with powerful GPUs [70, 71].

3.3.3 Dedicated hardware

This category includes ASICs, FPGAs and other hardware crafted specifically for the purpose of computing hash functions quickly. In the recent years, there has been emerging research of hybrid CPU-FPGA accelerators [72, 73, 74]. This approach shows a promising application prospect thanks to its flexibility and energy efficiency compared to ASICs and FPGAs, which suffer from high design cost and poor flexibility [74].

[39] based on on-demand pricing in the US East (N. Virginia) region

3.4 Conclusion

Let us first talk about tools. Based on our findings, Hashcat seems to outperform JtR in all scenarios except the ones with highly GPU-resistant KDFs such as scrypt. For that reason and the sake of consistency, we will be using Hashcat for our experiemnts in Chapter 4. It should be noted that JtR is a great tool for cracking highly GPU-resistant KDFs, especially if the system running it does not have powerful GPUs. In any case, the reader should always check which tool yields better performance for their specific use case.

Our attack methods will depend on the specific capabilities of the tool we chose — Hashcat. We are going to focus on smart brute-force attacks with Markov models and dictionary attacks with word mangling.

Finally, as for hardware, we are going to focus on regular PC setups and cloud computing. Regular PCs are going to be represented by the author's Lenovo ThinkPad X1 Extreme (Gen 2) laptop[40]. This laptop is running a GNU/Linux system and is equipped with an Intel Core i7-9750H CPU (6 cores / 12 threads), 32 GB of memory and an NVIDIA GeForce GTX 1650 Max-Q GPU.

For cloud computing, we are going to use an NC24s v3 Virtual Machine (VM) from Microsoft Azure. This VM costs $12.24 per hour[41] (billed per second) and comes with 24 virtual CPU cores, 448 GB of memory and four NVIDIA Tesla V100 GPUs [75]. In case the reader is considering AWS for their cloud computing resources, this VM should perform about the same as an EC2 p3.8xlarge instance, which comes with the exact same GPU setup [68].

[40] `https://www.lenovo.com/us/en/laptops/thinkpad/thinkpad-x1/X1-Extreme-Gen-2/p/22TP2TXX1E2`

[41] based on pay as you go pricing in the East US region

Chapter 4

Password cracking: implementation

In this chapter, we are going to implement and evaluate a couple of approaches to password cracking. We will first experiment with Markov models used in Hashcat's brute-force mode and run a variety of dictionary attacks. Those experiments will be run on the author's laptop, which we have described in Section 3.4. After that, we will repeat some of the experiments in a cloud computing environment to see how much performance there is to gain.

Throughout the chapter, we are mostly going to work with a subset of passwords from the 2012 LinkedIn data breach [76]. Although the passwords were originally leaked in the form of SHA-1 hashes, we have plaintexts available for the entire subset of 2,285,273 passwords. In each experiment, we are going to hash these passwords with the given hash function and therefore always end up with a set of the same size.

The hash functions / KDFs we will be testing are unsalted SHA-1, salted SHA-256, bcrypt and scrypt. To save space in tables, we will abbreviate *passwords cracked* to *PC* and *passwords cracked per second* to *PC/s*.

4.1 Methods

In this section, we are going to talk about the specifics of the password-cracking methods, which we will later use in our expriments.

4.1.1 Brute-force attack

For the brute-force attack, we will first run the default brute-force mode and then tweak the `-t` / `--markov-threshold` parameter to reduce the keyspace and potentially crack more longer passwords.

By observing Hashcat's source code[42], we can see that the default value of `--markov-threshold`, which results in no keyspace reduction, is 256. We can also see that Hashcat sets the threshold to 256 when `--markov-threshold` is 0. For those reasons, we will only test threshold values between 1 and 256.

Our brute-force attacks will be limited in runtime. Note that we will not be using Hashcat's built-in `--runtime` parameter, because it resets the limit each time Hashcat transitions to a new keyspace. Instead, we will use the *timeout* command from GNU core utilities[43] to limit the runtime. This means that any initial setup Hashcat does before the cracking process will also count towards the runtime limit.

4.1.2 Dictionary attack

Even though the plaintexts are available to us, we will not base our dictionaries on passwords from the 2012 LinkedIn data breach. For our dictionaries, we will be using the RockYou dictionary[44], the free Openwall dictionary[45] and a combination of the two.

We will also experiment with word mangling. More specifically, we are going to use the following rule sets:

- Best64 (77 rules)[46],
- T0XICv1 (11931 rules)[47],
- Generated2 (65117 rules)[48],
- all of the above combined.

4.2 Regular PC

The experiments in this section will be run on the author's laptop. For the hardware specifications, refer to Section 3.4. We are going to use both the CPU and the GPU, because, as we can see in Table 4.1, combining them gives us better performance than using the GPU by itself. SHA-1 and SHA-256 were tested with Hashcat's built-in `--benchmark`, while the speeds of bcrypt and scrypt were measured with a brute-force attack on a single hash with the mask `?l?l?l?l?l?l?l?l`.

[42] available at `https://github.com/hashcat/hashcat`

[43] `https://www.gnu.org/software/coreutils/coreutils.html`

[44] `https://gitlab.com/kalilinux/packages/wordlists/-/blob/kali/master/rockyou.txt.gz`

[45] `https://download.openwall.net/pub/wordlists/all.gz`

[46] `https://github.com/hashcat/hashcat/blob/master/rules/best64.rule`

[47] `https://github.com/hashcat/hashcat/blob/master/rules/T0XlCv1.rule`

[48] `https://github.com/hashcat/hashcat/blob/master/rules/generated2.rule`

Hash function	CPU+GPU speed	GPU speed
SHA-1	3,725 MH/s	3,499 MH/s
SHA-256	1,423 MH/s	1,324 MH/s
bcrypt ($cost = 10$)	568 H/s	381 H/s
scrypt ($N = 2^{14}, r = 8, p = 1$)	265 H/s	37 H/s

Table 4.1: Comparison of Hashcat's CPU+GPU and GPU-only hash rates (regular PC)

4.2.1 Raw unsalted SHA-1

We will begin with raw unsalted SHA-1 hashes — the same hashes LinkedIn was using back in 2012 when some of their passwords leaked. This will be a good demonstration of what we can expect in a scenario, where we are cracking many weak unsalted hashes.

Since this is a weak hash that can be computed very quickly at 3.7 GH/s on our machine[49], we are going to run our attacks on all 2.2 million hashes.

Let us look at the results of our brute-force attacks — Figure 4.1 and Figure 4.2. Per-position Markov models consistently performed better than classic Markov models, beating them by 2–50 % in all scenarios. The difference is most prominent in thresholds 10 and 25 (33–50 %). Other thresholds stay within 2–9 % of increase in performance, except threshold 100, which performed 5–20 % better with per-position Markov models.

When we focus on the 30minute runtime, the most promising Markov thresholds seem to be 50 and 75. For that reason, we chose these two thresholds to show how the fraction of cracked passwords changed with longer runtimes — one hour and two hours. We only used per-position Markov models in these tests, because they have so far outperformed classic Markov models in every scenario. The results are in Figure 4.3 and Table 4.2.

We can see that Markov threshold 50 yields better results in shorter runtimes, though the difference becomes negligible once we get to one or two hours. Threshold 75 even beats threshold 50 by 0.02 % in the case of the two-hour runtime.

An interesting finding is that for the five-minute runtime, Markov threshold 50 performed worse than threshold 25. This may be because there was not enough time to iterate to larger keyspaces (longer masks), which may contain more passwords.

On the other hand, per-position Markov models with threshold 25 and their runtime limited to 5 minutes had the highest PC/s value at 432 passwords cracked per second. If we ignore Markov threshold 10, which performed very

[49] based on Hashcat's built-in benchmark

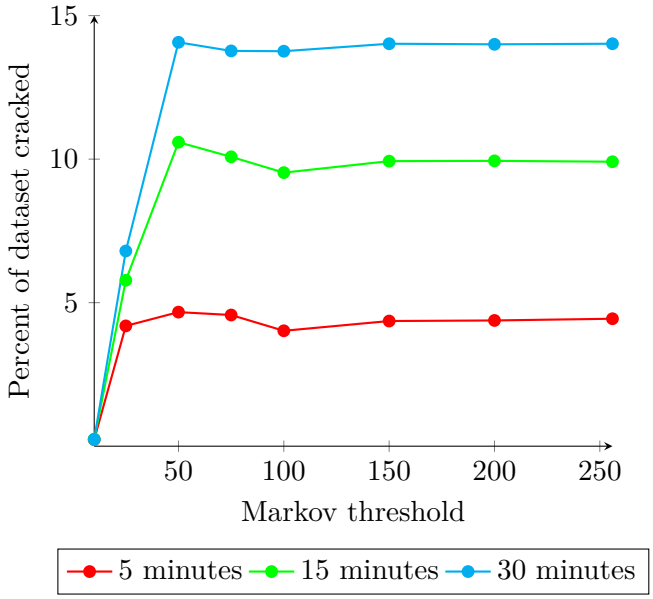

Figure 4.1: Results of brute-force attacks on raw unsalted SHA-1 with *classic Markov models* grouped by *runtime limit* (regular PC)

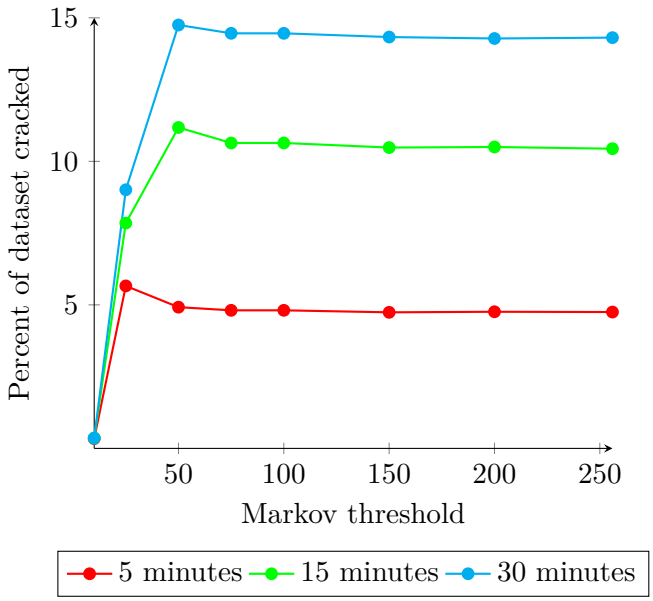

Figure 4.2: Results of brute-force attacks on raw unsalted SHA-1 with *per-position Markov models* grouped by *runtime limit* (regular PC)

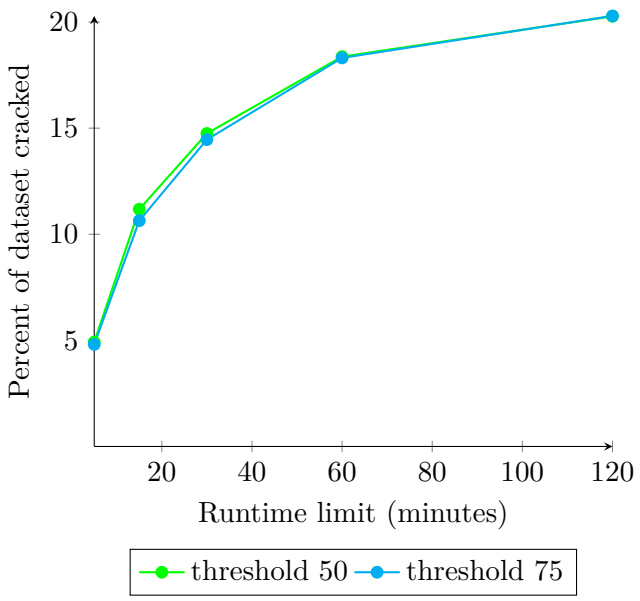

Figure 4.3: Results of brute-force attacks on raw unsalted SHA-1 with *per-position Markov models* grouped by *Markov threshold* (regular PC)

Threshold	**Runtime**	**PC**	**PC/s**
50	00h 05m 00s	112,442 (04.92 %)	375
75	00h 05m 00s	109,966 (04.81 %)	367
50	00h 15m 00s	255,519 (11.18 %)	284
75	00h 15m 00s	243,056 (10.64 %)	270
50	00h 30m 00s	336,968 (14.75 %)	187
75	00h 30m 00s	330,553 (14.46 %)	184
50	01h 00m 00s	419,833 (18.37 %)	117
75	01h 00m 00s	418,525 (18.31 %)	116
50	02h 00m 00s	463,277 (20.27 %)	64
75	02h 00m 00s	463,685 (20.29 %)	64

Table 4.2: Results of brute-force attacks on raw unsalted SHA-1 with *per-position Markov models* (regular PC)

Dictionary	Rules	Runtime	PC	PC/s
RockYou	-	00m 09s	93 (00.00 %)	10
RockYou	Best64	00m 11s	20,091 (00.88 %)	1,826
RockYou	T0XICv1	02m 40s	176,500 (07.72 %)	1,103
RockYou	Generated2	17m 35s	305,875 (13.38 %)	290
RockYou	combined	20m 56s	363,989 (15.93 %)	290
Openwall	-	00m 09s	320 (00.01 %)	36
Openwall	Best64	00m 09s	4,694 (00.21 %)	522
Openwall	T0XICv1	00m 57s	62,399 (02.73 %)	1,095
Openwall	Generated2	06m 36s	86,421 (03.78 %)	218
Openwall	combined	07m 22s	120,369 (05.27 %)	272
combined	-	00m 10s	404 (00.02 %)	40
combined	Best64	00m 11s	22,176 (00.97 %)	2,016
combined	T0XICv1	03m 31s	193,495 (08.47 %)	917
combined	Generated2	24m 01s	328,454 (14.37 %)	228
combined	combined	27m 18s	391,206 (17.12 %)	239

Table 4.3: Results of dictionary attacks on raw unsalted SHA-1 (regular PC)

poorly (5–26 PC/s), other scenarios with per-position Markov models stayed within 361–432 (five-minute runtime), 199–284 (15minute runtime) or 114–187 (30minute runtime) passwords cracked per second. The one-hour runtime did not see any decrease of PC/s from the 30minute runtime, while the two-hour runtime roughly cut the value in half to 64 PC/s.

Moving on to the results of our dictionary attacks in Table 4.3, we can see that the combined dictionary with Best64 mangling rules performed the best when it comes to passwords cracked per second. In just 11 seconds, it cracked 22,176 passwords (2,016 passwords per second), which may be useful in scenarios, where the hash function is not nearly as fast to compute, but by no means is it a lot — 22,176 passwords is only 0.97 % of the dataset.

Next, we are going to step down to scenarios with around 1,000 passwords per second, which have cracked a larger amount of the dataset. The combined dictionary with T0XICv1 mangling rules cracked 193,495 passwords in 3 minutes and 31 seconds (917 passwords per second). This is a much more satisfactory result as it cracked 8.47 % of the dataset.

To move on to better[50] results from there, we have to step down all the way to the 200–300 passwords per second range. Here, the best result is arguably going to be the combined dictionary with all of the mangling rules combined, which cracked 391,206 passwords in 27 minutes and 18 seconds (239 passwords per second). This is the result with the most cracked passwords — 17.12 % of the dataset.

What may be a bit surprising is that dictionaries with no mangling rules, especially the RockYou dictionary, which consists of passwords that people have used in the past, have only cracked a negligible amount of passwords — 93 (RockYou), 320 (Openwall) and 404 (combined). This suggests that password collisions are rare, though people still tend to compose their passwords of the same words in various predictable patterns, as we have seen from our results with mangling rules.

Let us also explore combinations of brute-force and dictionary attacks. If our runtime was limited to around 30 minutes and we decided to only use one approach, our best bets for this dataset would be per-position Markov models with Markov threshold 50 (cracked 14.75 % of the dataset in 30 minutes) and the combined dictionary with combined mangling rules (cracked 17.12 % of the dataset in 27 minutes). However, when we combine brute-force[51] and dictionary attacks to get a combined runtime of around 30 minutes, we always get better results than just sticking to one approach. Five minutes of brute force (4.92 %) and the combined dictionary with combined mangling rules (17.12 %) would crack 19.81 % of the dataset in 32 minutes, 15 minutes of brute force (11.18 %) and the RockYou dictionary with Generated2 mangling rules (13.38 %) would crack 21.12 % of the dataset in 33 minutes and finally, 30 minutes of brute force (14.75 %) and the combined dictionary and T0XICv1 mangling rules (8.47 %) would crack 20.90 % of the dataset in 34 minutes. Therefore, it is seemingly always worth it to divide our efforts into a brute-force and a dictionary attack instead of only using one of the methods.

4.2.2 Raw salted SHA-256

We will follow unsalted SHA-1 with salted SHA-256 hashes. This is another weak hash, which we can compute at the speed of 1.4 GH/s on our machine[52], but the fact that it is salted will make the cracking process much more difficult for a large amount of passwords. The hashes are salted with random 16byte salts encoded as base16 strings.

We are first going to run our attacks on the entire dataset. We would expect to crack little to no passwords with this approach, because all of the salts are unique and thus the amount of hashes we would have to compute to get

[50] as in more passwords cracked

[51] We only considered per-position Markov models with Markov threshold 50, because those performed the best.

[52] based on Hashcat's built-in benchmark

through the whole dataset would be about 2.2 million (the size of our dataset) times more than in the case of an unsalted SHA-1. For that reason, we will only try attacks that previously gave us the best PC/s values in Section 4.2.1 and see how far we get in 30 minutes. Specifically, we will run a brute-force attack using per-position Markov models with Markov threshold 25 and a dictionary attack with the combined dictionary and the Best64 mangling rules. For the brute-force variant, we are going to limit ourselves to six-character passwords (mask `?1?2?2?2?2?2` with `-1 ?l?d?u` and `-2 ?l?d`), which is the minimum length of passwords in our dataset. To make dictionary attacks on salted hashes more feasible, we would need a much smaller and much more personalized/targetted dictionary.

We will then simulate a scenario that is more likely to occur in the real world, where an attacker might divide the dataset into small chunks and spend a fixed amount of time trying to crack each chunk. We will shuffle our dataset, take the first 1,500 hashes and spend one hour trying to crack them. We will use a combination of a brute-force and a dictionary attack, because, as we have discovered in Section 4.2.1, it gives better results than scenarios, where only one of the approaches is used. We will use the same parameters with high PC/s values, which we have described in the previous paragraph, because 1,500 unique salts will still cause a significant slowdown compared to unsalted hashes. Since the dictionary attack would take almost an hour by itself, we are going to limit both attacks to 30 minutes. If an attacker adopted this specific method for this specific dataset, it would take them about two months to get through the entire dataset[53].

Our attacks on the entire dataset at once did not manage to crack any passwords within 30 minutes[54]. This was expected as 2.2 million unique salts result in 2.2 million times more hashes to be computed, i.e., a 2.2 million times slower cracking process. Therefore, cracking this salted dataset with a theoretical speed of 1.4 GH/s would be equivalent to cracking unsalted hashes at 613 H/s.

The attacks we ran on the chunk of 1,500 password hashes managed to crack 1 (brute-force attack) and 11 (dictionary attack) passwords. Since we went about halfway through a dictionary, which should crack 0.97 % of chunks from this particular dataset on average[55], this is about as much as we would expect, but by no means a lot. We could use a larger rule set and let the attacks run for, let us say, days, but that would make the attack, as in going through the entire dataset of millions of passwords, unfeasible.

Although SHA-256 is still a weak hash, running an attack on the entire dataset of 2.2 million salted hashes is unrealistic. We could crack 0.97 % of the dataset with the combined dictionary and Best64 mangling rules in about

[53] regardless of their hardware, because the time spent on each chunk is fixed

[54] meaning each of the attacks ran for 30 minutes

[55] based on our previous results

Cost	CPU speed	GPU speed	Combined speed
4	11,936 H/s	21,474 H/s	33,410 H/s
5	5,966 H/s	11,247 H/s	17,213 H/s
6	2,961 H/s	5,905 H/s	8,865 H/s
7	1,475 H/s	2,991 H/s	4,467 H/s
8	764 H/s	1,515 H/s	2,278 H/s
9	384 H/s	750 H/s	1,135 H/s
10	191 H/s	378 H/s	569 H/s
11	92 H/s	190 H/s	282 H/s
12	47 H/s	96 H/s	143 H/s

Table 4.4: Speed of brute-force attacks on a single *bcrypt* hash based on cost (regular PC)

51 days, but stepping up to larger rule sets would increase the length of the attack to many years. However, if our dataset was much smaller, e.g., if the chunk of 1,500 password hashes was the entire dataset, we could crack 17.12 % of it with the combined dictionary and combined mangling rules in about a month. If it was even smaller, let us say, 50 password hashes, we could do the same in just one day.

4.2.3 bcrypt

With bcrypt, we are slowly approaching the state of the art territory, where computation gets much slower. Since bcrypt is also salted, we can apply methods from Section 4.2.2 to make the attack more feasible (smaller dictionary or cracking smaller chunks for a fixed time), though the hash rate is going to be much more limiting in this case.

Because bcrypt is very slow to compute[56] and salted, we will focus on cracking a single password and compare the speed / hash rate across various values of the cost parameter. We will test this by running a brute-force attack with the mask `?l?l?l?l?l?l?l?l`. Since the cost parameter of bcrypt translates to 2^{cost} iterations internally [41], we would expect the computation to get two times slower/faster for each increment/decrement of the cost parameter.

As running an attack on bcrypt with high cost is very computationally expensive on a regular PC[57], this is where cloud computing will probably start to shine.

[56] when used with recommended parameters

[57] i.e., the author's laptop

N	r	p	CPU speed	GPU speed	Combined speed
2^{10}	8	1	4,322 H/s	3,886 H/s	8,208 H/s
2^{11}	8	1	2,193 H/s	1,394 H/s	3,587 H/s
2^{12}	8	1	1,069 H/s	442 H/s	1,511 H/s
2^{13}	8	1	501 H/s	131 H/s	633 H/s
2^{14}	8	1	219 H/s	37 H/s	256 H/s
2^{15}	8	1	81 H/s	10 H/s	91 H/s

Table 4.5: Speed of brute-force attacks on a single *scrypt* hash based on cost (regular PC)

The results in Table 4.4 follow the trend we hinted at in the previous paragraph — each increment of the cost parameter more or less halves the speed. Also notice that our CPU contributes to about a third of the total speed. This is because bcrypt is somewhat GPU-resistant.

In Section 3.2.3, we mentioned that some implementations like Go's *crypto* package use bcrypt with $cost = 10$ by default. Cracking bcrypt with such high cost would be very ineffective on our machine. Perhaps if we were trying to crack just one password, the attack could be feasible, but at these kinds of speeds, we would need over nine hours to go through our combined dictionary before applying any mangling rules and about a month if we applied the Best64 rule set. However, if a lower cost was used instead, the whole process would be much quicker. For example, $cost = 8$ would mean that instead of a month, it would only take us a week to go through our combined dictionary with Base64 mangling rules. If the cost was even lower, we could start looking into cracking multiple passwords. Still, cracking a whole dataset on the machine we are working with would be unrealistic.

4.2.4 scrypt

Scrypt is, by some, considered state of the art when it comes to password hashing. It is salted, much more GPU-resistant than bcrypt and also takes longer to compute than bcrypt. Methods from Section 4.2.2 still apply to make the attack more feasible, though we are going to be limited by the speed even more than in the case of bcrypt.

Just like with bcrypt, we are going to stick to comparison of cracking speeds across various cost parameters and then discuss, whether there is potential for a feasible attack. We will perform our experiments using the same method, more specifically, we will run a brute-force attack with the mask `?l?l?l?l?l?l?l?l` for various values of N with fixed $r = 8, p = 1$.

From the results in Table 4.5, we can see that the CPU does most of the work here, which confirms that scrypt is much more GPU-resistant than bcrypt. We only went up to $N = 2^{15}$, because Hashcat failed to launch attacks for $N = 2^{16}$ or higher. Doubling the value of N halves the performance on the CPU side, as we would expect, and also cuts the hash rate of the GPU by roughly 3–4 times. For $N = 2^{15}$, the GPU is barely doing any work.

Once again, we are going to reference Section 3.2.3, where we have talked about Go's implementation of scrypt, which recommends parameters $N = 2^{15}, r = 8, p = 1$ for interactive logins. Cracking scrypt with these parameters would be about six times slower than cracking bcrypt with its recommended parameters[58] and nearly impossible on our machine unless the targetted password was very weak. For a single hash, it would take us over 70 hours to go through our combined dictionary without applying any mangling rules. If we applied the Best64 rule set on top of that, we would be looking at about six to seven months of non-stop cracking.

Based on our results, we can confidently say that cloud computing is going to be a must for cracking scrypt hashes. Cracking scrypt on a regular machine is not feasible unless we can drastically limit the searched keyspace to the point, where the attack makes sense, which is unlikely.

4.3 Cloud computing

For our cloud computing experiments, we are going to use an NC24s v3 VM on Microsoft Azure. For the hardware specifications and pricing, refer to Section 3.4. Just like in Section 4.2, we will first run a few a Hashcat benchmarks to check whether it is better to utilize the CPU for password cracking or just use the GPUs by themselves. We will also use the same methods as before — SHA-1 and SHA-256 will be tested with Hashcat's built-in `--benchmark`, while the speeds of bcrypt and scrypt will be measured with a brute-force attack on a single hash with the mask `?l?l?l?l?l?l?l?l`.

Although some of the speeds we compared in our results (Table 4.6) were very close and their values could be equal within the margin of error, we are still only going to utilize the GPUs for all hash functions except scrypt, where the CPU+GPU combination performs much better.

4.3.1 Raw unsalted SHA-1

Compared to our regular PC, the SHA-1 hash rate of the VM was about 17 times higher at 64.2 GH/s. We are first going to run a brute-force attack with per-position Markov models and Markov threshold 50 and compare the results to the regular PC.

[58] i.e., $cost = 10$

Hash function	CPU+GPU speed	GPU speed
SHA-1	63,102 MH/s	64,198 MH/s
SHA-256	24,017 MH/s	24,657 MH/s
bcrypt ($cost = 10$)	8,717 H/s	9,309 H/s
scrypt (N=$2^{14}, r = 8, p = 1$)	981 H/s	573 H/s

Table 4.6: Comparison of Hashcat's CPU+GPU and GPU-only hash rates (cloud computing)

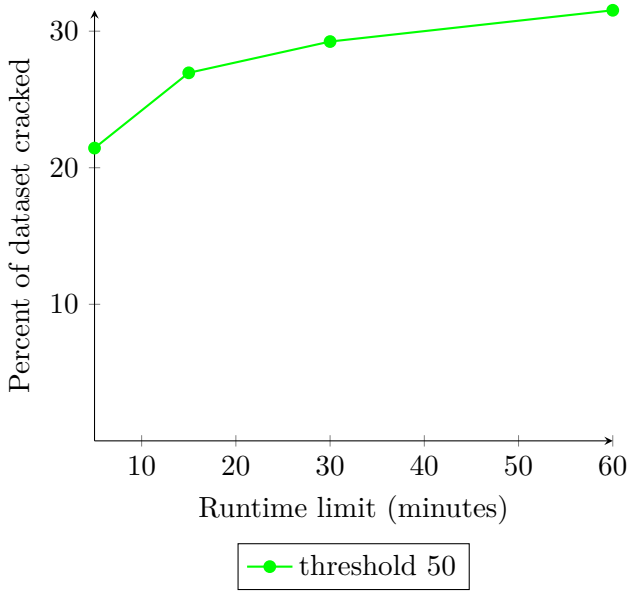

Figure 4.4: Results of brute-force attacks on raw unsalted SHA-1 with *per-position Markov models* (cloud computing)

The results in Figure 4.4 show that in just five minutes (21.44 % of dataset cracked), we have already beaten what the regular PC achieved in two hours (20.27 % of dataset cracked). Table 4.7 shows how drastically the PC/s value decreases with longer runtimes — from 1,633 (five-minute runtime) all the way to 200 (one-hour runtime). This may be because the larger keyspaces we were going through with longer runtimes perhaps did not contain as many passwords.

As for the dictionary attacks, because of the 17 times higher hash rate, going through the combined dictionary with combined mangling rules would likely take us 1–2 minutes. Since the dictionary attack is so fast, it would be a good idea to run it first and only then complement it with a brute-force attack if the initial results were not good enough.

Threshold	Runtime	PC	PC/s
50	00h 05m 00s	489,961 (21.44 %)	1,633
50	00h 15m 00s	615,834 (26.95 %)	684
50	00h 30m 00s	668,134 (29.24 %)	371
50	01h 00m 00s	720,448 (31.53 %)	200

Table 4.7: Results of brute-force attacks on raw unsalted SHA-1 with *per-position Markov models* (cloud computing)

4.3.2 Raw salted SHA-256

Compared to the regular PC, the VM's SHA-256 hash rate was also about 17 times higher at 24.7 GH/s. With this speed, some attacks on the entire dataset become feasible. For instance, we could go through the combined dictionary with Best64 mangling rules in 60–70 hours, though stepping up to the T0XICv1 mangling rules would increase this time frame to anywhere from a year to one year and three months.

We tested the same approach we introduced in Section 4.2.2, which consisted of dividing the dataset into chunks of 1,500 hashes and cracking each one of them for about 30 minutes. We combined a 30minute brute-force attack, which used per-position Markov models and Markov threshold 25, with a dictionary attack, which used the combined dictionary with Best64 mangling rules and only ran for 2 minutes and 30 seconds. Using the same dataset of 1,500 hashes from Section 4.2.2, we managed to crack 15 (brute-force attack) and 19 (dictionary attack) passwords for a total of 34 unique passwords (2.27 % of the chunk). We can see how the brute-force attack progressed in Figure 4.5. If this is any indication of how these methods would perform on other chunks, this approach could crack 2.27 % of the entire dataset in roughly a month.

Let us talk about scenarios with smaller datasets like the one we mentioned towards the end of Section 4.2.2, where our chunk of 1,500 hashes was the entire dataset. In that case, we could use the combined dictionary with T0XICv1 mangling rules for a dictionary attack that would only take 5–6 hours. We could even step up to the combined rule set for a dictionary attack that would take about 35 hours.

If we were only cracking a single hash, the salt would not cause any slowdown and we could, for example, go through the combined dictionary with combined mangling rules within minutes. Similarly with a brute-force attack, checking all guesses up to eight characters[59] with Markov threshold 50 would only take a few minutes as well.

[59] with Hashcat's default mask

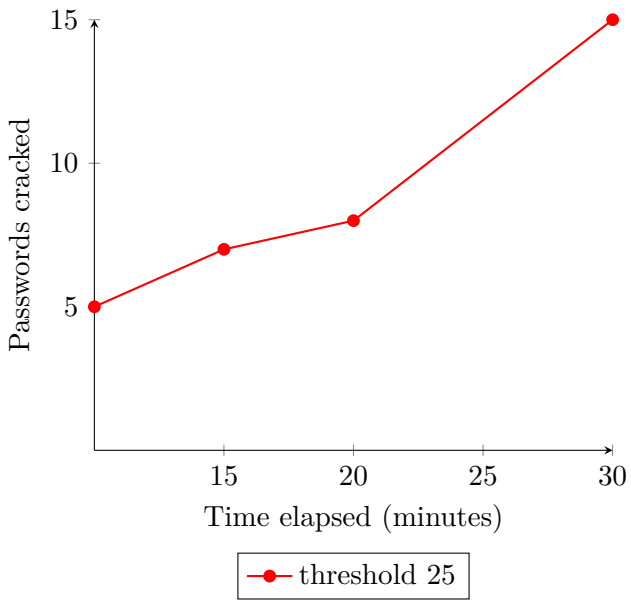

Figure 4.5: Progression of a brute-force attack on a chunk of 1,500 raw salted SHA-256 hashes with *per-position Markov models* (cloud computing)

4.3.3 bcrypt

We will begin by measuring the hash rate of bcrypt across various values of the cost parameter. The results are in Table 4.8.

We see that the VM can compute bcrypt 16–17 times faster than the regular PC. We will only focus on bcrypt with $cost = 10$ as that is what we consider the recommended cost value. That being said, this speedup is not enough to make attacks on multiple hashes or entire datasets feasible with such high cost. For that reason, our discussion will revolve around cracking a single bcrypt hash.

The speedup of our VM allows for a couple of feasible attacks — brute-forcing passwords six or seven characters long with Markov threshold 25 (7 and 170 hours respectively) and dictionary attack with the combined dictionary and Best64 mangling rules (45 hours). If we stepped up our brute-force to eight characters or our dictionary attack to the T0XICv1 rule set, the attacks would take much longer — six months for the brute-force and a year for the dictionary attack.

Since cracking passwords longer than seven characters is likely going to be unfeasible, we will have to rely on dictionary attacks. For dictionary attacks to be efficient, we would need a much more targetted dictionary that would be much smaller than our combined dictionary with 19 million words. Reducing the dictionary size to 50,000 words would make the dictionary attack with T0XICv1 rule set take just one day instead of a year.

Cost	CPU speed	GPU speed	Combined speed
4	-	561,200 H/s	561,200 H/s
5	-	288,400 H/s	288,400 H/s
6	-	146,700 H/s	146,700 H/s
7	-	73,653 H/s	73,653 H/s
8	-	37,125 H/s	37,125 H/s
9	-	18,504 H/s	18,504 H/s
10	-	9,180 H/s	9,180 H/s
11	-	4,655 H/s	4,655 H/s
12	-	2,297 H/s	2,297 H/s

Table 4.8: Speed of brute-force attacks on a single *bcrypt* hash based on cost (cloud computing)

N	r	p	CPU speed	GPU speed	Combined speed
2^{10}	8	1	8,024 H/s	104,081 H/s	112,105 H/s
2^{11}	8	1	4,338 H/s	25,260 H/s	29,598 H/s
2^{12}	8	1	2,121 H/s	7,573 H/s	9,694 H/s
2^{13}	8	1	967 H/s	2,138 H/s	3,105 H/s
2^{14}	8	1	409 H/s	572 H/s	981 H/s
2^{15}	8	1	190 H/s	148 H/s	338 H/s

Table 4.9: Speed of brute-force attacks on a single *scrypt* hash based on cost (cloud computing)

However, if bcrypt was used incorrectly with a lower cost, the proposed attacks would be much faster. As we have mentioned before, each decrement of the cost parameter roughly doubles the speed of our attacks.

4.3.4 scrypt

First, we will measure the hash rate across various cost parameters the same way we did in Section 4.2.4. The results are in Table 4.9

We see that the VM is about 4–14 times faster than the regular PC depending on the value of N. The performance gap shrinks with increasing N. We are going to go down a path similar to bcrypt — we will only focus on cracking a single hash of scrypt with the recommended parameters

($N = 2^{15}, r = 8, p = 1$). Just like in the case of bcrypt on cloud computing, this speedup is not enough to crack multiple hashes or entire datasets efficiently. We see a similar trend that the CPU's hash rate halves as we double the N value, while the GPUs take a bigger performance penalty. When we get to $N = 15$, the CPU computes more hashes per second than the GPUs.

There could potentially be a few feasible attacks. One of them is a brute-force attack on six-character passwords with Markov threshold 25, which would take eight to nine days. We could also run a dictionary attack with the combined dictionary by itself (15 hours) or Best64 mangling rules on top of it (50 days). Stepping up to seven-character passwords (seven months) or the T0XICv1 rule set (21 years) would make the attacks too long.

Scrypt is in a similar situation as bcrypt, except that it is 27 times slower[60]. Feasible attacks could be achieved with much smaller and much more targetted dictionaries — a dictionary of 10,000 words paired with the T0XICv1 rule set would take about 4 days to get through.

Just like bcrypt, if scrypt is used with weak cost parameters, our proposed attacks would be much faster, though attacking a dataset of many passwords with unique salts would still be unlikely to be met with success unless the passwords were very weak.

Since scrypt with high cost is faster on a CPU than a GPU, we also ran a JtR benchmark with the built-in `--test` parameter on the VM and found that for $N = 2^{14}, r = 8, p = 1$, it could do 843 H/s with just the CPU, which is only 14 % short of Hashcat's CPU+GPU performance (981 H/s).

Assuming the CPU hash rate drops by half each time N is doubled, JtR would beat Hashcat for $N = 2^{15}$, where the GPUs receive an even bigger performance penalty. Therefore, using JtR paired with a VM that is more focused on CPU performance rather than GPU performance is something the reader should strongly consider before taking on scrypt hashes with high cost parameters.

[60] when both are used with recommended parameters

Chapter 5

Password cracking: recommendations

In this chapter, we are going to talk about what approaches to cracking password hashes the reader should take based on the results of our experiments from Chapter 4. We will focus on three variables — hardware, dataset size and used hash function.

5.1 Hardware

Although we saw up to 17 times higher hash rates with cloud computing, it was not enough to turn many years of cracking into feasible attacks. Nevertheless, it did a great job at accelerating attacks that were already at least somewhat feasible, e.g., turning 17 months into one month or 17 days into one day.

The attacks could be accelerated even further with Amazon's top of the line EC2 p4d.24xlarge instance, which manages hash rates three to four times higher[61] than the NC24s v3 VM from Microsoft Azure that we used [47]. Unfortunately, AWS did not approve the author's request to access one.

As we have discovered near the end of Chapter 4, using JtR together with more CPU-focused resources is likely going to be more efficient when it comes to highly GPU-resistant KDFs like scrypt with strong parameters.

Keep in mind renting cloud computing resources can get quite expensive quite fast. The VM we used goes for $12.24/hour ($8,935.20/month) with pay as you go pricing, though there are discounts for long-term commitments, specifically a 36 % discount for a one-year commitment, i.e., $7.7970/hour or $5,692.7516/month, and a 68 % discount for a three-year commitment, i.e., $3.9169/hour or $2,859.2786/month[62] [75]. For comparison, Amazon's EC2

[61] on SHA-1, SHA-256 and bcrypt (the source did not benchmark scrypt)
[62] based on pricing in the East US region.

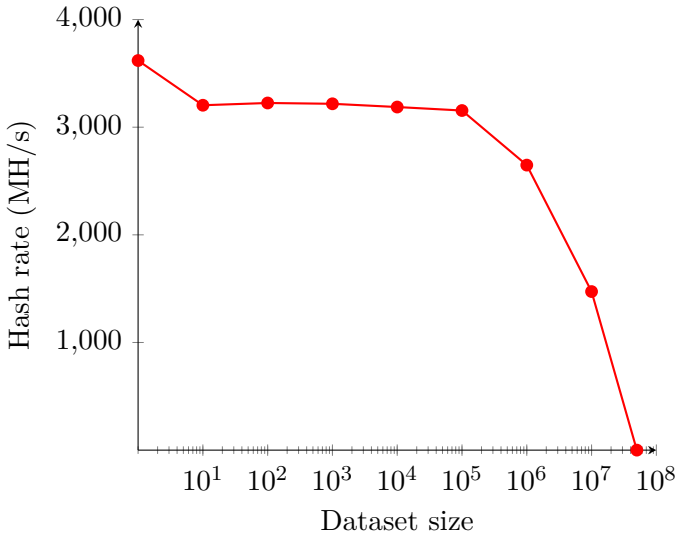

Figure 5.1: Effect of dataset size on the hash rate of SHA-1

p4d.24xlarge instance starts at $32.7726/hour[63] and also offers one-year as well as three-year plans with discounts [69, 77].

5.2 Dataset size

Our intuition may tell us that the size of the dataset only matters when the hash is salted. That is not true — there are massive performance penalties when cracking very large datasets.

We ran a couple of tests to see how much the dataset size affects the hash rate. The tests were run on the author's laptop, which we have described in Section 3.4. The results are in Figure 5.1.

When only cracking a single hash, Hashcats activates its Single-Hash optimizer, which results in roughly 13 % more performance compared to cracking ten hashes. After that, there is not much difference in speed between ten and 100,000 hashes. Going up to 1,000,000 hashes drops the speed by 16 %. Stepping even further causes an overflow of Hashcat's bitmap table, which results in a 45% slowdown. Pushing things even further to 50,000,000 hashes (2 GB of data), which is about as much as Hashcat was able to load, drives the performance to the ground, resulting in a 34,500 times lower hash rate.

When dealing with tens of millions of passwords, consider dividing the dataset into smaller chunks to avoid the performance penalty.

[63] based on on-demand pricing in the US East (N. Virginia) region

5.3 Hash function

In this section, we are going to provide the reader with recommendations for password cracking based on used hash function.

5.3.1 Unsalted weak hash

An unsalted weak hash, for instance MD5 or SHA-1, is a hash that can be computed very quickly — think hash rates of megahashes or gigahashes per second. Thanks to such high hash rates, we can get away with brute-force attacks as well as dictionary attacks with large dictionaries and large rule sets. The combined dictionary with combined mangling rules we used in Chapter 4 had $1.46 \cdot 10^{12}$ entries after applying the mangling rules and only took 27 minutes to get through when attacking unsalted SHA-1 on the regular PC.

For best results, divide your efforts in a brute-force attack and a dictionary attack whenever possible — run a relatively quick[64] dictionary attack first and then use the remaining time for a brute-force attack. As for what Markov threshold should be used for the brute-force attack, we found that threshold 50 seems to yield the best results. When limited by time and/or slow hardware, consider using threshold 25 for quick results. In the opposite scenario, consider threshold 75 to slightly increase the keyspace and potentially crack more password that would have otherwise been missed due to keyspace reduction.

Because the hash is unsalted, the size of the dataset will likely not affect the hash rate by much. Consequences of cracking very large datasets at once were discussed in Section 5.2.

5.3.2 Salted weak hash

A salted weak hash is the same kind of hash we described in Section 5.3.1, except that it is salted. Salting results in having to compute *amount of unique salts* hashes more than in the case of unsalted hashes — if we had a dataset of 10,000 hashes and each one had a unique salt, the cracking process would take 10,000 times longer than if the same hash was used without a salt.

When cracking large datasets, consider dividing the datasets into smaller chunks and spending a fixed amount of time on each one. As we have already suggested in Section 5.3.1, combine brute-force and dictionary attacks for best results whenever possible. Lowering the Markov threshold from 50 to 25 may help when cracking multiple hashes with unique salts. If the dataset is large and running a brute-force attack becomes unfeasible, stick to a dictionary attack and consider using a smaller dictionary and/or a smaller word-mangling rule set. Cloud computing will likely be required for feasible attacks on larger datasets.

[64] as in quick compared to the brute-force attack

5.3.3 Salted strong hash

Salted strong hashes are going to be similar to salted weak hashes with the difference that they are slow to compute even before considering the salt. An example of such hash would be bcrypt.

Since the low hash rate is very limiting, dictionary attacks are going to be the way to go here, as they achieve considerably higher PC/s (passwords cracked per second) values than brute-force attacks and generally search a much smaller keyspace.

Unless a very small dictionary with a relatively small set of mangling rules is used, cloud computing is going to be a must. Cracking more than one hash will be very hard, because the already low hash rate will be divided by the amount of unique salts in the dataset. Cracking larger datasets will likely be impossible with the hardware available today unless the targetted passwords are very weak or a very small and effective dictionary is used.

In the case of hash functions with variable cost like bcrypt, weak cost parameters could significantly speed up the cracking process. For example, since each decrement of bcrypt's cost parameter roughly doubles the hash rate, the cracking speed is going to be 64 times faster with $cost = 4$ than with $cost = 10$ (the recommended value).

5.3.4 Salted strong GPU-resistant hash

The one salted and highly GPU-resistant hash we tested is scrypt. The main difference between scrypt and bcrypt is that scrypt is much more GPU-resistant, resulting in larger performance penalties for the GPU with increasing cost parameters.

When scrypt is used with the recommended parameters ($N = 2^{15}, r = 8, p = 1$), the hash rate drops to just hundreds of hashes per second with cloud computing. The regular PC did not even reach 100 H/s on scrypt with the recommended parameters. Because of this, the only feasible attacks are going to be dictionary attacks with small dictionaries and small rule sets. Even with cloud computing, cracking a single scrypt hash is going to be hard. Attacking multiple hashes or larger datasets seems very unrealistic.

However, we could potentially speed up cracking of scrypt with strong cost parameters by using cloud computing resources that are more focused on CPU performance rather than VMs with multiple GPUs, which we have used for our cloud computing experiments in Chapter 4. Such resources combined with JtR, which achieved about 2.3 times higher hash rate on the CPU than Hashcat[65] would likely result in some speedup, though it would most definitely not be enough to invalidate what we said in the previous paragraphs of this section. Keep in mind that lowering N significantly boosts the hash rate of GPUs, thus making JtR much less competitive.

[65] in an scrypt test with $N = 2^{15}, r = 8, p = 1$ on the regular PC

Just like in the case of bcrypt, weak cost parameters can significantly speed up the cracking process — scrypt with $N = 2^{10}, r = 8, p = 1$ was over 300 times faster than scrypt with $N = 2^{15}, r = 8, p = 1$ on the VM we used in Chapter 4. Dividing N by 2 resulted in roughly four times higher hash rate in our cloud computing experiments.

5.4 Time estimates

On top of what we discussed in Section 5.3, the reader can get an idea of how long their attacks are going to take from Table 5.1 (brute-force attacks) and Table 5.2 (dictionary attacks).

Simply look up how long an attack with some specific parameters would take and then divide that time frame by how many times the used hardware is faster. For example, if an attack took eight years at 1 MH/s and our machine was able to compute 4 MH/s, the attack would only take $8/4 = 2$ years. If the hashes are salted, multiply the prior result by the amount of unique salts in the dataset.

		Time @ hash rate			
MT	**ML**	**1 GH/s**	**1 MH/s**	**1 kH/s**	**1 H/s**
25	3	< 1 second	< 1 second	16.3 seconds	4.5 hours
25	4	< 1 second	< 1 second	6.8 minutes	4.7 days
25	5	< 1 second	10.2 seconds	2.8 hours	3.9 months
25	6	< 1 second	4.2 minutes	2.9 days	8.1 years
25	7	6.4 seconds	1.8 hours	2.5 months	201.6 years
25	8	2.7 minutes	1.8 days	5 years	-
25	9	1.1 hours	1.5 months	126 years	-
25	10	1.2 days	3.2 years	-	-
25	11	28.7 days	78.8 years	-	-
25	12	10.2 months	-	-	-
25	13	8.4 years	-	-	-
25	14	84 years	-	-	-
50	3	< 1 second	< 1 second	1.1 minutes	18.5 hours
50	4	< 1 second	2.4 seconds	40 minutes	27.8 days
50	5	< 1 second	1.4 minutes	1 day	2.7 years
50	6	3.1 seconds	51.8 minutes	1.2 months	98.6 years
50	7	1.9 minutes	1.3 days	3.6 years	-
50	8	1.3 hours	1.8 months	145.1 years	-
50	9	2.2 days	6 years	-	-
50	10	3 months	243.8 years	-	-
50	11	10 years	-	-	-
50	12	107.5 years	-	-	-

Table 5.1: Estimated duration of brute-force attacks based on Markov threshold (MT), maximum password length (ML) and hash rate

	Time @ hash rate			
DS	**1 GH/s**	**1 MH/s**	**1 kH/s**	**1 H/s**
10^2	< 1 second	< 1 second	< 1 second	1.7 minutes
10^3	< 1 second	< 1 second	1 second	16.7 minutes
10^4	< 1 second	< 1 second	10 seconds	2.8 hours
10^5	< 1 second	< 1 second	1.7 minutes	1.2 days
10^6	< 1 second	1 second	16.7 minutes	11.6 days
10^7	< 1 second	10 seconds	2.8 hours	3.9 months
10^8	< 1 second	1.7 minutes	1.2 days	3.2 years
10^9	1 second	16.7 minutes	11.6 days	31.7 years
10^{10}	10 seconds	2.8 hours	3.9 months	317.1 years
10^{11}	1.7 minutes	1.2 days	3.2 years	-
10^{12}	16.7 minutes	11.6 days	31.7 years	-
10^{13}	2.8 hours	3.9 months	317.1 years	-
10^{14}	1.2 days	3.2 years	-	-
10^{15}	11.6 days	31.7 years	-	-
10^{16}	3.9 months	317.1 years	-	-
10^{17}	3.2 years	-	-	-
10^{18}	31.7 years	-	-	-
10^{19}	317.1 years	-	-	-

Table 5.2: Estimated duration of dictionary attacks based on dictionary size after applying mangling rules (DS) and hash rate

Conclusion

The main goal of this thesis was to create a set of recommendations to show penetration testers which approaches to password cracking work best in various scenarios. Leading up to this goal, there were several other subtasks. All of the goals were fulfilled.

We first explored and compared authentication methods commonly used today. Then, we talked about the history as well as the present of password authentication and also discussed what its future may hold. After that, we performed an analysis of hardware, tools and methods used for password cracking and decided which approaches we were going to use for our implementation.

In the implementation, we experimented with Hashcat's brute-force and dictionary attacks accelerated with Markov models and word mangling rules. We also showed how the size of the keyspace searched in a brute-force attack can be reduced by specifying a Markov threshold value. After running experiments on the author's machine, we explored how our attacks could be accelerated with cloud computing and performed similar expriments on a VM with multiple high-performance GPUs provided by Microsoft Azure.

Finally, in the last chapter, we summarized how hardware, dataset size and used hash funtion can all affect the speed of password cracking. On top of that, we laid out some recommendations for password cracking and created two tables, which estimate how long brute-force and dictionary attacks are going to take based on the attack's parameters and hash rate.

Bibliography

1. Merriam-Webster. *Authentication* [online]. [N.d.] [visited on 2021-03-03]. Available from: `https://www.merriam-webster.com/dictionary/authentication`.
2. YADRON, Danny. Man Behind the First Computer Password: It's Become a Nightmare. *The Wall Street Journal* [online]. 2013 [visited on 2021-03-03]. Available from: `https://www.wsj.com/articles/BL-DGB-35227`.
3. SCHNEIDER, Fred B. *Something You Know, Have, or Are* [online]. [N.d.] [visited on 2021-03-03]. Available from: `https://www.cs.cornell.edu/courses/cs513/2005fa/NNLauthPeople.html`.
4. LORD, Nate. Uncovering Password Habits: Are Users' Password Security Habits Improving? (Infographic). *Digital Guardian* [online]. 2020 [visited on 2021-03-06]. Available from: `https://digitalguardian.com/blog/uncovering-password-habits-are-users-password-security-habits-improving-infographic`.
5. PasswordManager.com. *65 % of people don't trust password managers despite 60 % experiencing a data breach* [online]. [N.d.] [visited on 2021-03-27]. Available from: `https://www.passwordmanager.com/password-manager-trust-survey`.
6. 1Password. *1Password Security Design* [online]. 2019. Available also from: `https://1password.com/files/1Password-White-Paper.pdf`. [visited on 2021-03-27, release v0.2.10].
7. 1Password. *How to add your 1Password account to the apps* [online]. [N.d.] [visited on 2021-03-27]. Available from: `https://support.1password.com/add-account`.
8. AREŽINA, Luka. Save Your Data with These Empowering Password Statistics. *DataProt* [online]. 2021 [visited on 2021-03-06]. Available from: `https://dataprot.net/statistics/password-statistics`.

9. Proton Team. The pros and cons of biometric authentication. *ProtonMail Blog* [online]. 2020 [visited on 2021-03-07]. Available from: https://protonmail.com/blog/biometric-authentication.

10. Visa. *Consumers ready to switch from passwords to biometrics, study shows* [online]. [N.d.] [visited on 2021-03-27]. Available from: https://usa.visa.com/visa-everywhere/security/how-fingerprint-authentication-works.html.

11. TSAI, Peter. What are Digital Certificates? *Spiceworks* [online]. 2018 [visited on 2021-03-27]. Available from: https://community.spiceworks.com/security/articles/2952-data-snapshot-biometrics-in-the-workplace-commonplace-but-are-they-secure.

12. SHERIDAN, K. Researchers Fool Biometric Scanners with 3D-Printed Fingerprints. *Dark Reading* [online]. 2020 [visited on 2021-03-07]. Available from: https://www.darkreading.com/endpoint/researchers-fool-biometric-scanners-with-3d-printed-fingerprints/d/d-id/1337522.

13. BBC. Face ID iPhone X 'hack' demoed live with mask by Bkav [online]. 2017 [visited on 2021-03-07]. Available from: https://www.bbc.com/news/av/technology-41992610.

14. Bkav Corp. *Bkav's New Mask Beats Face ID in "Twin Way": Do not Use Face ID in Business Transactions* [YouTube video]. 2017 [visited on 2021-03-07]. Available from: https://www.youtube.com/watch?v=rhiSBc061JU.

15. RANKL, Wolfgang; EFFING, Wolfgang. *Smart Card Handbook*. 3rd ed. Milton, Queensland, Australia: Wiley, 2004. ISBN 9780470856680.

16. SSH Communications Security. *SSH Command* [online]. [N.d.] [visited on 2021-03-14]. Available from: https://www.ssh.com/ssh/command.

17. CHACON, Scott; STRAUB, Ben. *Pro Git*. 2nd ed. Berkeley, California, United States of America: Apress, 2014. ISBN 9781484200773. Available also from: https://git-scm.com/book/en/v2.

18. SSH Communications Security. *Public Key authentication for SSH* [online]. [N.d.] [visited on 2021-03-28]. Available from: https://www.ssh.com/ssh/public-key-authentication.

19. SSH Communications Security. *Private Key* [online]. [N.d.] [visited on 2021-03-28]. Available from: https://www.ssh.com/cryptography/private-key.

20. SSH Communications Security. *Public Key* [online]. [N.d.] [visited on 2021-03-28]. Available from: https://www.ssh.com/cryptography/public-key.

21. SSH Communications Security. *PKI - Public Key Infrastructure* [online]. [N.d.] [visited on 2021-03-28]. Available from: https://www.ssh.com/pki.

22. HOUSLEY, Russ; POLK, William; FORD, Warwick; SOLO, David. *Internet X. 509 public key infrastructure certificate and certificate revocation list (CRL) profile* [Internet Requests for Comments]. RFC Editor, 2002-04. RFC, 3280. RFC Editor. ISSN 2070-1721. Available also from: https://www.rfc-editor.org/rfc/rfc3280.txt.

23. GIRY, Damien. *Cryptographic Key Length Recommendation* [online]. BlueKrypt, 2020 [visited on 2021-03-13]. Available from: https://www.keylength.com.

24. ENGLER, M. *State of the Auth: Experiences and Perceptions of Multi-Factor Authentication* [online]. 2019-12 [visited on 2021-03-14]. Tech. rep. Duo Security. Available from: https://duo.com/assets/ebooks/state-of-the-auth-2019.pdf.

25. M'RAIHI, D. et al. *HOTP: An HMAC-Based One-Time Password Algorithm* [Internet Requests for Comments]. RFC Editor, 2005-12. RFC, 4226. RFC Editor. ISSN 2070-1721. Available also from: https://www.rfc-editor.org/rfc/rfc4226.txt.

26. M'RAIHI, D. et al. *TOTP: Time-Based One-Time Password Algorithm* [Internet Requests for Comments]. RFC Editor, 2011-05. RFC, 6238. RFC Editor. ISSN 2070-1721. Available also from: https://www.rfc-editor.org/rfc/rfc6238.txt.

27. MCMILLAN, Robert. The World's First Computer Password? It Was Useless Too. *WIRED* [online]. 2012 [visited on 2021-04-10]. Available from: https://www.wired.com/2012/01/computer-password.

28. WALDEN, David; VAN VLECK, Tom. Compatible Time-Sharing System (1961–1973): Fiftieth Anniversary Commemorative Overview. *IEEE Computer Society*. 2011. Available also from: https://multicians.org/thvv/compatible-time-sharing-system.pdf.

29. MORRIS, Robert; THOMPSON, Ken. Password Security: A Case History. *Commun. ACM*. 1979, vol. 22, no. 11, pp. 594–597. ISSN 0001-0782. Available from DOI: 10.1145/359168.359172.

30. RITCHIE, D. M.; THOMPSON, K. The UNIX Time-Sharing System†. *Bell System Technical Journal*. 1978, vol. 57, no. 6, pp. 1905–1929. Available from DOI: 10.1002/j.1538-7305.1978.tb02136.x.

31. NAKOV, Svetlin. *Practical Cryptography for Developers*. Sofia, Bulgaria, 2018. ISBN 9786190008705. Available also from: https://cryptobook.nakov.com.

32. PROPOSED FEDERAL INFORMATION PROCESSING DATA ENCRYPTION STANDARD. *Cryptologia*. 1977, vol. 1, no. 3, pp. 292–306. Available from DOI: 10.1080/0161-117791833039.

33. LIU, Hongjun; XU, Yanqiu; MA, Chao. Chaos-based image hybrid encryption algorithm using key stretching and hash feedback. *Optik*. 2020, vol. 216, p. 164925. ISSN 0030-4026. Available from DOI: 10.1016/j.ijleo.2020.164925.

34. FELDMEIER, David C.; KARN, Philip R. UNIX Password Security - Ten Years Later. In: BRASSARD, Gilles (ed.). *Advances in Cryptology — CRYPTO' 89 Proceedings*. New York, NY: Springer New York, 1990, pp. 44–63. ISBN 978-0-387-34805-6.

35. DEBNATH, Santanu; CHATTOPADHYAY, Abir; DUTTA, Subhamoy. Brief review on journey of secured hash algorithms. In: *2017 4th International Conference on Opto-Electronics and Applied Optics (Optronix)*. 2017, pp. 1–5. Available from DOI: 10.1109/OPTRONIX.2017.8349971.

36. PESLYAK, Alexander; MARECHAL, Simon. Password security: past, present, future. *Openwall* [online]. 2012 [visited on 2021-04-16]. Available from: https://www.openwall.com/presentations/Passwords12-The-Future-Of-Hashing/Passwords12-The-Future-Of-Hashing.pdf.

37. TrustedSec Team. Of History & Hashes: A Brief History of Password Storage, Transmission, & Cracking. *TrustedSec Blog* [online]. 2015 [visited on 2021-04-11]. Available from: https://www.trustedsec.com/blog/passwordstorage.

38. WONG, David. SHA-3 vs the World. In: *OWASP London Chapter Meeting* [YouTube video]. London, England: OWASP London, 2017 [visited on 2021-04-12]. Available from: https://www.youtube.com/watch?v=2R3dTeRBhQw.

39. WALLEN, Jack. Git Transitioning Away from the Aging SHA-1 Hash. *The New Stack* [online]. 2020 [visited on 2021-04-12]. Available from: https://thenewstack.io/git-transitioning-away-from-the-aging-sha-1-hash.

40. BARKER, E.; CHEN, L.; DAVIS, R. *SP 800-56C Rev. 2, Recommendation for Key-Derivation Methods in Key-Establishment Schemes*. U.S. Department of Commerce, National Institute of Standards and Technology, 2020. Available from DOI: 10.6028/NIST.SP.800-56Cr2.

41. PROVOS, N.; MAZIERES, D. A Future-Adaptable Password Scheme. In: *USENIX ATC, FREENIX Track*. 1999, pp. 81–91. Available also from: https://www.usenix.org/legacy/events/usenix99/full_papers/provos/provos.pdf.

42. Google. *Firebase Authentication Password Hashing* [online]. [N.d.] [visited on 2021-04-15]. Available from: https://firebaseopensource.com/projects/firebase/scrypt.

43. WETZELS, Jos. Open Sesame: The Password Hashing Competition and Argon2. *CoRR*. 2016, vol. abs/1602.03097. Available from arXiv: 1602.03097.

44. AUMASSON, J-P. The impact of quantum computing on cryptography. *Computer Fraud & Security*. 2017, vol. 2017, no. 6, pp. 8–11. ISSN 1361-3723. Available from DOI: 10.1016/S1361-3723(17)30051-9.

45. LAVOR, C.; MANSSUR, L. R. U.; PORTUGAL, R. Grover's Algorithm: quantum database search. *arXiv e-prints*. 2003. Available from arXiv: quant-ph/0301079.

46. National Academies of Sciences, Engineering, and Medicine. *Quantum Computing: Progress and Prospects*. Ed. by HOROWITZ, Mark; GRUMBLING, Emily. Washington, DC, USA: The National Academies Press, 2019. ISBN 978-0-309-47969-1. Available from DOI: 10.17226/25196.

47. rarecoil. *hashcat v6.1.1 p4d.24xlarge AWS NVIDIA A100-SXM4-40GB benchmark* [GitHub gist]. 2020 [visited on 2021-04-24]. Available from: https://gist.github.com/rarecoil/2baf9f335faa7ad044a46281773ec5b3.

48. CUBRILOVIC, Nik. RockYou Hack: From Bad To Worse. *TechCrunch* [online]. 2009 [visited on 2021-04-25]. Available from: https://techcrunch.com/2009/12/14/rockyou-hack-security-myspace-facebook-passwords.

49. CHOU, Hsien-Cheng et al. Password cracking based on learned patterns from disclosed passwords. *IJICIC*. 2013, vol. 9, no. 2, pp. 821–839. ISSN 1349-4198.

50. BONNEAU, Joseph. The Science of Guessing: Analyzing an Anonymized Corpus of 70 Million Passwords. In: *2012 IEEE Symposium on Security and Privacy*. 2012, pp. 538–552. Available from DOI: 10.1109/SP.2012.49.

51. NARAYANAN, Arvind; SHMATIKOV, Vitaly. Fast Dictionary Attacks on Passwords Using Time-Space Tradeoff. In: *Proceedings of the 12th ACM Conference on Computer and Communications Security*. Alexandria, VA, USA: Association for Computing Machinery, 2005, pp. 364–372. CCS '05. ISBN 1595932267. Available from DOI: 10.1145/1102120.1102168.

52. MARECHAL, Simon. Advances in password cracking. *Journal in computer virology*. 2008, vol. 4, no. 1, pp. 73–81. Available from DOI: 10.1007/s11416-007-0064-y.

53. WEIR, Matt; AGGARWAL, Sudhir; MEDEIROS, Breno de; GLODEK, Bill. Password Cracking Using Probabilistic Context-Free Grammars. In: *2009 30th IEEE Symposium on Security and Privacy*. 2009, pp. 391–405. Available from DOI: 10.1109/SP.2009.8.

54. *hashcat* [online]. [N.d.] [visited on 2021-04-27]. Available from: https://hashcat.net/wiki/doku.php?id=hashcat.

55. *Rule-based Attack* [online]. [N.d.] [visited on 2021-04-27]. Available from: https://hashcat.net/wiki/doku.php?id=rule_based_attack.

56. *Combinator Attack* [online]. [N.d.] [visited on 2021-04-27]. Available from: https://hashcat.net/wiki/doku.php?id=combinator_attack.

57. *Mask Attack* [online]. [N.d.] [visited on 2021-04-27]. Available from: https://hashcat.net/wiki/doku.php?id=mask_attack.

58. *Hybrid Attack* [online]. [N.d.] [visited on 2021-04-27]. Available from: https://hashcat.net/wiki/doku.php?id=hybrid_attack.

59. Openwall. *John the Ripper's cracking modes* [online]. [N.d.] [visited on 2021-04-27]. Available from: https://github.com/openwall/john/blob/bleeding-jumbo/doc/MODES.

60. Openwall. *PRINCE mode* [online]. [N.d.] [visited on 2021-04-27]. Available from: https://github.com/openwall/john/blob/bleeding-jumbo/doc/PRINCE.

61. Openwall. *John the Ripper usage examples* [online]. [N.d.] [visited on 2021-04-27]. Available from: https://github.com/openwall/john/blob/bleeding-jumbo/doc/EXAMPLES.

62. Openwall. *Mask mode* [online]. [N.d.] [visited on 2021-04-27]. Available from: https://github.com/openwall/john/blob/bleeding-jumbo/doc/MASK.

63. Openwall. *Markov mode* [online]. [N.d.] [visited on 2021-04-28]. Available from: https://github.com/openwall/john/blob/bleeding-jumbo/doc/MARKOV.

64. Openwall. *SUBSETS mode* [online]. [N.d.] [visited on 2021-04-28]. Available from: https://github.com/openwall/john/blob/bleeding-jumbo/doc/SUBSETS.

65. The Go Authors. *bcrypt* [online]. [N.d.] [visited on 2021-05-02]. Available from: https://pkg.go.dev/golang.org/x/crypto/bcrypt.

66. The Go Authors. *scrypt* [online]. [N.d.] [visited on 2021-05-02]. Available from: https://pkg.go.dev/golang.org/x/crypto/scrypt.

67. VENNAM, Sai. Cloud Computing. *IBM Cloud Learn Hub* [online]. 2020 [visited on 2021-04-20]. Available from: https://www.ibm.com/cloud/learn/cloud-computing.

68. Amazon Web Services. *Amazon EC2 Instance Types* [online]. [N.d.] [visited on 2021-04-24]. Available from: https://aws.amazon.com/ec2/instance-types.

69. Amazon Web Services. *EC2 On-Demand Instance Pricing* [online]. [N.d.] [visited on 2021-04-24]. Available from: https://aws.amazon.com/ec2/pricing/on-demand.

70. Microsoft. *GPU optimized virtual machine sizes* [online]. [N.d.] [visited on 2021-05-03]. Available from: https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-gpu.

71. Google. *GPUs on Compute Engine* [online]. [N.d.] [visited on 2021-05-03]. Available from: https://cloud.google.com/compute/docs/gpus.

72. LIU, Peng; LI, Shunbin; DING, Qingyuan. An Energy-Efficient Accelerator Based on Hybrid CPU-FPGA Devices for Password Recovery. *IEEE Transactions on Computers*. 2019, vol. 68, no. 2, pp. 170–181. Available from DOI: 10.1109/TC.2018.2868191.

73. ZHANG, Zhendong; LIU, Peng. A Hybrid-CPU-FPGA-based Solution to the Recovery of Sha256crypt-hashed Passwords. *IACR Transactions on Cryptographic Hardware and Embedded Systems*. 2020, vol. 2020, no. 4, pp. 1–23. Available from DOI: 10.13154/tches.v2020.i4.1-23.

74. ZHANG, Zhendong; LIU, Peng; WANG, Weidong; LI, Shunbin; WANG, Peng; JIANG, Yingtao. High-Performance Password Recovery Hardware going from GPU to Hybrid CPU-FPGA Platform. *IEEE Consumer Electronics Magazine*. 2020. Available from DOI: 10.1109/MCE.2020.3042166.

75. Microsoft. *Linux Virtual Machines Pricing* [online]. [N.d.] [visited on 2021-05-03]. Available from: https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux.

76. FINKLE, Jim; SABA, Jennifer. LinkedIn suffers data breach. *Reuters* [online]. 2012 [visited on 2021-05-01]. Available from: https://www.reuters.com/article/net-us-linkedin-breach-idUSBRE85511820120606.

77. Amazon Web Services. *Amazon EC2 Pricing* [online]. [N.d.] [visited on 2021-05-09]. Available from: https://aws.amazon.com/ec2/pricing.

Appendix A

Acronyms

2FA Two-Factor Authentication. 9, 10

ASIC Application-Specific Integrated Circuit. 18, 19

AWS Amazon Web Services. 31, 32

CA Certificate Authority. 8

CFG Context-Free Grammar. 26

CPU Central Processing Unit. 19

CTSS Compatible Time-Sharing System. 13, 14

CUDA Compute Unified Device Architecture. 27, 30

DES Data Encryption Standard. 14–16

DHKE Diffie-Hellman Key Exchange. 20

DKIM DomainKeys Identified Mail. 17

DLP Discrete Logarithm Problem. 20

E2E End-to-End. 5

EC2 Elastic Compute Cloud. 31, 32

ECDLP Elliptic-Curve Discrete Logarithm Problem. 20

FPGA Field-Programmable Gate Array. 19

GECOS General Electric Comprehensive Operating Supervisor. 28

GPU Graphics Processing Unit. 18, 19, 23, 31

HMAC Hash-based Message Authentication Code. 18

HOTP HMAC-Based One-Time Password. 9

JtR John the Ripper. 26, 28–30, 32

KDF Key Derivation Function. 18–20

MD5 Message Digest 5. 17

MIT Massachusetts Institute of Technology. 3, 13, 17

NBS National Bureau of Standards. 14

NIST National Institute of Standards and Technology. 15, 17, 18, 21

NSA National Security Agency. 9

OpenCL Open Computing Language. 27, 30

OTP One-Time Password. 9

PBKDF2 Password-Based Key Derivation Function 2. 18

PC Personal Computer. 31

PCFG Probabilistic Context-Free Grammar. xv, 26, 27

PGP Pretty Good Privacy. 17

PHC Password Hashing Competition. 19

PIN Personal Identification Number. 3, 4

PKI Public Key Infrastructure. 5, 6, 8–12

QEC Quantum Error Correction. 20

QR Quick Response. 9

RSA Rivest–Shamir–Adleman. 8, 20, 21

SHA Secure Hash Algorithm. 17

SSH Secure Shell. 8, 12, 17

TLS Transport Layer Security. 17

TOTP Time-Based One-Time Password. 9–12

URI Uniform Resource Identifier. 9

VM Virtual Machine. 32, 43

Appendix B

Contents of enclosed SD card

- `dataset` ..... directory with data used in experiments
  - `dictionaries` ..... directory with used dictionaries
  - `hashes` ..... directory with used hashes
  - `rules` ..... directory with used rule sets
  - `dataset.plain` ..... password dataset in plaintext
- `src` ..... directory with source codes
  - `thesis` ..... directory with LaTeX sources of the thesis
- `text` ..... directory with compiled text of the thesis
  - `thesis.pdf` ..... text of the thesis in PDF
- `README.md` ..... description of contents