Università
Ca'Foscari
Venezia

Master's Degree

in Computer Science
Software Dependability and Cyber Security

Final Thesis

# Password Cracking and Tools: Analysis of Automatic Cracking Tools and Password Guessing

**Supervisor**
Ch. Prof. Riccardo Focardi
Ch. Prof. Flaminia Luccio

**Graduand**
Siviero Enrico
Matricolation number
865112

**Academic Year**
2021 / 2022

# Acknowledgement

Firstly, I would like to express my deepest thanks to my family for their support throughout my entire academic life.

My sincere thanks to my supervisors Riccardo Focardi and Flaminia Luccio for their help.

Finally, I would like to thank all my friends.

# Abstract

Passwords are very important and are necessary for most of our activities. They are the primary authentication system, they are stored in the form of one-way hashes. People usually select passwords that are easy to remember, and an attacker can hash a large quantity of easy passwords until a match is found with the target hash. Some password cracking tools such as hashcat can be used to efficiently crack the passwords using dictionaries and appropriate rules. Recent works on password guessing show how the tools can be used to efficiently crack passwords using some trained dictionaries and rules. Through previous analysis and dataset previously trained we tried to increment the success rate of the attack. We tried to increase the number of password cracked or get down the time necessary using a new type of approach based on iterations. To achieve this goal, we carry out an analysis of previous work done and we modify the algorithm [2] used to try to achieve a better success rate. Our result shows that, with this new approach, we can reach the previous result and overcome it.

# Contents

# Introduction

Nowadays passwords are very important and are necessary for most of our activities. They are the primary authentication system, although other types also exist. An example is the Biometric authentication, but when a problem arises, the user then usually needs to insert a password. So passwords are always a must in every authentication system. Password authentication has numerous advantages but also disadvantages that must be taken into account. One advantage of passwords is the easiness with which they are created, but at the same time they could be more vulnerable to attacks. A password to be less vulnerable must follow some guidelines. Users usually do not follow them and they choose very simple passwords, which can lead to many problems.

The importance of choosing a well formed password is a difficult task for a lot of people, that tend to choose simple passwords to easily remember them. Using the guidelines over the years, users have begun to use increasingly complex passwords, usually formed from combinations of known words, also using numbers and special characters. These passwords, however, are easily predictable despite these guidelines.

The password chosen by the user is not stored in clear in the database but using a hash function that can encrypt it and hide the real password. Over the years some vulnerabilities have discovered in the application and in the database that contains the password and these vulnerabilities can be used to leak them. There are numerous datasets of leaked passwords created over the years. An example is the service *haveibeenpwned.com*, created specifically to check whether a password used by a user has been leaked and has become vulnerable. The leaked passwords are a very useful resource because represent a real set of passwords used by the people. Using them is possible to create some specific attacks to try to guess the passwords that are encrypted in the database. Even considering that users very often use the same password on multiple accounts, these leaks can lead to serious damage.

Two of the main tools used to recover passwords are *hashcat* and *johntheripper*. These two tools enable the recovery of hashed passwords using various techniques to reduce the number of attempts and they carry out a more targeted attack. *Hashcat* and *johntheripper* are highly optimized to make maximum use of the parallelism given by multicore CPUs and GPUs.

Many techniques have explored as attempt to crack or guess passwords, recent studies have shown that it is possible to use some combinations of attacks to obtain strong results. Attempting to crack a password requires a lot of resources and time.

In this thesis we focused on the analysis of the previous work done with these tools in attempt to crack the greatest number of passwords. We present a new approach that can improve the work done by Di Campi et al. [2]. We carry on an analysis about the various cracking strategies, and we propose a new approach using iterations. We test this new approach with the same dataset used in previous work to have a comparison of the results obtained. This allows us to achieve a higher success rate (number of password cracked). The thesis is structured as follows: The first chapter describes how an authentication mechanism works, which types of them exist and why the password authentication mechanism is very used.

The second chapter describes the principle oh hash function, which crypthographic hash functions are the most popular and the one we use.

The third chapter describes the dataset that we used and an analysis of them. After the analysis of the dataset in chapter Fourth we talk about the meaning behind the password cracking and the typologies of attacks.

In the fifth chapter, we explain which results we want to achieve and what we get from the experiment done.

In the last chapter, we conclude our study and we discuss possible future work.

# Chapter 1

# Authentication methods

This chapter is an introduction to User Authentication: Section 1.1 explains the basis of User Authentication, Section 1.2 extends to an Electronic User Authentication works, Section 1.3 discuses how password authentication works. Finally, some alternative authentication methods are briefly explained.

## 1.1 User Authentication

User authentication is the principal line of defense used in security. It consist of two phases [26]:

- Identification phase: Presenting an identifier to the security system
- Verification phase: Presenting or generating information that corroborates the binding between the entity and the identifier

Identification provides an identity to the user that wants to authenticate to a system. NIST SP 800-63-2 (Electronic Authentication Guideline, August 2013) [5] defines electronic user authentication as the process of establishing confidence in user identities that are presented electronically to an information system [26]. The typical authentication access is formed by:

- **user ID**: user identity;
- **password**: usually kept secret and encrypted.

## 1.2 Electronic User Authentication

NIST SP 800-63-2 defines a general model for user authentication that involves a number of entities and procedures. The main requirement for performing user authentication is that the user must be already registered with the system. A typical sequence for registration can be described as follow. An applicant applies to a *registration authority (RA)* to become a subscriber of a *credential service provider (CSP)*. In this template, the RA is a trusted entity that establishes and vouches for the identity of an applicant to a CSP. The CSP then engages in an exchange with the subscriber. Depending on the details of the overall authentication system, the CSP issues some sort of electronic credential to the subscriber. The **credential** is a data structure that authoritatively combines an identity and additional attributes to a token possessed by a subscriber, and can be checked when presented to the verifier in an authentication transaction. The token could be an encryption key or an encrypted password that identifies the subscriber. The token may be issued by the CSP, generated directly by the subscriber, or provided by a third party. The token and credential may be used in subsequent authentication events.

Once the user is registered as subscriber the authentication process is done between the subscriber and the system that perform the authentication process. We distinguish the two different parties:

- **claimant:** the user that needs to be authenticated, by proving the possession and control of a token;
- **verifier:** verify that the claimant is the subscriber named in the corresponding credential.

When the verifier confirms that the claimant is subscribed to the system the user can have access to the system. The verifier passes on an assertion about the identity of the subscriber to the **relying party (RP**)[18]. The relying party (RP) is an entity that relies upon the subscriber's authenticator(s) and credentials or a verifier's assertion of a claimant's identity, typically to process a transaction or grant access to information or a system. That assertion includes identity information about a subscriber, such as the subscriber name, an identifier assigned at registration, or other subscriber attributes that were verified (e.g. the username) in the registration process. The RP can use the authenticated information provided by the verifier to make access control or authorization decisions [26]. There are some classes of identification schemes described as follow:

- **Something known:** Check the knowledge of a secret. Example includes password, passphrases, Personal Identification Numbers (PINs), cryptographic keys.

- **Something possessed:** Check the possession of a device. Examples include electronic smart cards, physical keys and keycards. This type of authenticator is referred to as a token.

- **Something the individual is (static biometrics):** Examples include recognition by fingerprint, retina, and face.

- **Something the individual does (dynamic biometrics):** Examples include recognition by voice pattern, handwriting characteristics, and typing rhythm.

An attacker could be stealing or guessing the password. As for biometrics authenticators there are a lot of problems, including false positives and false negatives, user acceptance, cost, and convenience.
We focus on the first classes of identification: *Something Known.*

## 1.3 Password Authentication

Passwords have been used starting from ancient times, up to nowadays with the advent of the first computer. Passwords are the most used ways to authenticate a user. Password authentication is used in online services, personal email, banking account, military services and the idea is to protect from unauthorized access to these services.
All systems that use password authentication require that you not only provide an identifier (ID) but also a password. The system compares the obtained password with a previously saved password for that ID, stored in a password file. The password is used to authenticate the individual's ID. The

ID is used to determine whether a person is allowed to access and the user's privileges.

### 1.3.1 Vulnerabilities

A password-based authentication can have different vulnerabilities that can be performed to attack and some countermeasures that can be applied. Usually passwords are stored using a one-way hash function. The attack strategies and the countermeasures that can be applied are:
**Offline dictionary attack:** Bypass strong controls and gain access direct to the file. The attacker obtains the system password file and compares the hashed password with the hash of commonly used password. If a match is found, the attacker can gain access using ID/password combinations. An effective countermeasure may be to use a complicated password and change it often.
**Specific account attack:** The attacker targets specific accounts or users until he guesses the correct password. An efficient countermeasure is a lock-out mechanism that locks account after a finite number of tries.
**Popular password attack:** A variation of the specific account attack. Uses a popular password for different IDS and tries to guess it. This is very easy attack because a lot of people choose passwords that are easy to remember. A countermeasure is to force the user to choose a not common password.
**Password guessing against single user:** An attacker tries to gain knowledge about the account holder and system password policies and uses that knowledge to guess the password. Countermeasures can be to enforce the password policies.

**Workstation hijacking:** An attacker gains control of a workstation already logged-in. Countermeasures can be to force the user to logout after a defined length of time.

**Exploiting user mistakes:** The system assigns a default password to the user. Users can share this password with other colleagues or write down the password on paper. An attacker can get the password using social engineering by tricking the user into revealing the password. Default passwords assigned to the users are easily guessed because usually have a defined pattern. A countermeasures include user training, and simpler passwords with another authentication mechanism.

**Exploiting multiple password use:** Users can use the same single password for different devices. An attacker can take advantage and damage multiple devices or gain control. An effective countermeasure could be to force the use of different passwords for each device and for each account.

**Electronic monitoring:** Passwords are transmitted over the network in order to access remote services, they can be intercepted. Encrypting a password does not solve the problem because they can be observed and re-encrypting the same password will create the same ciphertext, the ciphertext will become the password.

### 1.3.2 Guidelines for creating a password

Despite the many problems of user authentication technique, it remains the most used method to authenticate a user.

*NIST SP 800-63 Digital Identity Guidelines* [5] suggests some guidelines for a creation of a password:

- Al least 8 character when a human sets it;
- 6 character minimum when set by a system/service;
- Should support at least 64 characters maximum length;
- All ASCII character (including space) must be supported;
- Unicode character must be supported as well;
- Passwords must be composed by uppercase and lowercase letter, a symbol, and a digit;
- Check if a password inserted by a user is a commonly used one.

A similar suggestion of well-formed password is given by the author of *rockyou2021* [17]: An actual an well-formed password should be 6-20 characters long and only include printable ASCII (American Standard Code for Information Interchange) characters except space, tab and other control characters. All the previous strategies and recommendations are based on four basic technique:

**User education:** Users need to known the importance of using hard-to-guess passwords and may be recommended some guidelines to create a strong password. Many users ignore the guidelines and think that an reversed password or capitalizing last letter can give a strong password.

**Computer-generated password:** Automatically generated passwords can be difficult to remember and a user may be tempted to write the password on paper. To avoid this kind of problem,the *Federal Information Processing Standard (FIPS) 181* describes one of the best Automated random password generators, which describes a standard process for converting random bits (from a hardware random number generator) into somewhat pronounceable "words" suitable for a passphrase. [19].

**Reactive password checking:** This strategy periodically checks with a password cracker if there are weak passwords in the system.

**Complex password policy** or **proactive password checker:** The user creates the password and the system checks if the password is acceptable and, if it is not, rejects it.

### Password policies

A password policy is a set of rules used to increase password security by encouraging users to use more complicated passwords and use them in the right way. There are many types of password policies created by governments or companies. Some of the most widely used guidelines are those recommended by the United States Department of Commerce's National Institute of Standards and Technology (NIST).

## 1.4 Alternative Authentication methods

Authentication has changed over the years, new alternatives in addition to password authentication were created. This alternative methods of authentication are very useful because allow users to authenticate without a passwords. In this section we will briefly describe them.

## Token-based Authentication

Token-based authentication is an authentication method where a single token is generated for a user and is used to verify the user's identity in future requests to the system [26]. In token-authentication, a user sends a request to a server with a valid username and password. The server authenticates the request, generates the token, and sends it to the user. The user can then use the generated token to authenticate and make requests to the system.

The main advantage of token-based authentication over traditional username and password authentication is that the token can be encrypted and stored securely on the client side. This allows for a stateless authentication process, where the server does not have to store the user's session information. Tokens can be invalidated or revoked if necessary, making authentication more secure than a traditional one with username and password.

Token-based authentication is commonly used in web and mobile applications, and can be used in combination with other authentication methods, such as two-factor authentication, for added security. Tokens must be saved on the client side and sent securely over the network .

Token authentication cane be implemented by different devices such as memory cards, smart cards.

## Biometric Authentication

Biometric authentication system authenticate an individual based on his or her unique physical characteristics [26]. These include static characteristics, such as fingerprints, hand geometry, facial characteristics, and retinal and iris patterns; and dynamic characteristics, such as voiceprint and signature. In essence, biometrics is based on pattern recognition. Biometric authentication can be used to replace or supplement traditional forms of authentication, such as passwords and security tokens. The main advantage of biometric authentication over traditional authentication methods is that biometric data is unique and cannot be forgotten, or lost as can occur with passwords or security tokens. Biometric authentication also provides a more convenient and user-friendly experience for users, as they do not need to remember a password or security token.

Biometric authentication also presents some challenges, including privacy concerns about the storage and use of biometric data, the possibility of false rejection or false acceptance, and the need for expensive hardware and software systems Compared to passwords and tokens, biometric authentication is both technically more complex and expensive.

## Multi-Factor Authentication

Multi-Factor Authentication (MFA) or Two-factor authentication (2FA) is a security process that requires a user to provide two different authentication factors to verify their identity. These factors are usually something the user knows (such as a password or PIN) and something the user has (such as a physical token).

The purpose of 2FA is to increase the security of authentication by requiring a second form of verification that an attacker would not have access to. If an attacker were to steal or guess a user's password, they would not be able to gain access to the user's account without also possessing the second factor.

Multi-factor authentications can be implemented using different strategies:

- SMS or voice call verification: A code is sent to the user's mobile phone via SMS or a voice call, and the user enters the code to authenticate.
- Time-based One-Time Password (TOTP) authentication: The user installs an app on their mobile device that generates a new code every few seconds, which they enter to authenticate.
- Physical tokens: A hardware device is used to generate a new code every few seconds, which the user enters to authenticate.
- Biometric authentication: The user's biometric information, such as their fingerprint or face, is used to authenticate.

Two-Factor authentications is widely used by many online services and is an important security measure for protecting sensitive information and accounts.

### Single Sign-On Authentication

Single sign-on (SSO) is an authentication mechanism that allows users to use one set of login credentials (e.g. username and password) to access multiple applications. When a user is logged in using a SSO, can gain access to multiple applications without the necessity to enter their credentials. SSO is very useful because improve usability of applications and reduce the number of passwords that the users needs to remember. There are different types of SSO protocol, such as OAuth [12].

# Chapter 2

# Cryptography

This chapter is an introduction to Hash Function: Section 2.1 shows the basis of an Hash Function, Section 2.2 explain the most used crypthographic hash functions used in modern cryptography and the one we used.

## 2.1 Hash Function

A cryptographic hash function is a hash algorithm that map a binary string to a another binary string with a size fixed of $n$ bits.
A hash function is used also to detect modifications of transmitted message called **data integrity**.

**Definition 1.** [21] A **hash function** $h : X \to Z$ is a function taking an arbitrarily long message x and giving a digest z of fixed length.

A hash function must respect three properties to be considered safe.

**Definition 2.** [21] Given a hash function $h : X \to Y$ and an element $x \in X$

and $z \in Y$. $H$ is **preimage resistant** (or one-way) if given $z$ it is infiseable to compute $x$ such that $h(x) = z$.

**Example** We consider a hash function h(x) that splits message x into blocks $x_1, x_2, ..., x_n$ of a fixed size k and computes $h(x) = x_1 \oplus x_2 \oplus ... \oplus x_n$, the bit-wise xor of all blocks. The hash function is not preimage resistant, given a digest $z$ we can find preimages.

**Definition 3.** [21] Given a hash function $h : X \to Y$ and $x_1 \in X$ and $x_2 \in Y$. $h$ is **second preimage resistant** if given $x_1$ it is infeasible to compute $x_2$ such that $h(x_1) = h(x_2)$.

**Example** An attacker creates two messages that look the same but one ($x_2$) gives more advantage than the other one ($x_1$), such as in two contracts with different prices. The attacker can convinces the other party to sign $x_1$ which looks appropriate but obtains a signature on $x_2$.

**Definition 4.** [21] Given a hash function $h : X \to Y$ and an element $x_1 \wedge x_2 \in X$ it is **collision resistan**t if it is infesible to compute different $x_1$ and $x_2$ such that $h(x_1) = h(x_2)$.

**Example** Taking in consideration the same hash function of Example 2.1. Given $x = x_1, x_2, ..., x_n$ we have that $x_2, x_1, ..., x_n$ has the same digest. All the blocks can be swapped because xor is commutative.

If a Hash function is not collision resistant is vulnerable to **birtday attack**, the name comes from the birthday paradox, which states that in a group of just 23 people, there is a greater than 50% chance that two of them share the same birthday. This is because there are 365 possible birthdays, and with just 23 people, there are 253 unique pairs of birthdays that could be shared. Similarly, in a cryptographic hash function with a sufficiently large output space, it may seem unlikely that two messages would produce the same hash value. However, due to the birthday paradox, the probability of a collision actually increases more rapidly than the number of messages being hashed [21].

### 2.1.1 Cryptographic hash functions

There are many cryptography hash algorithms, in this section we will explain the most used and which we used to perform the tests. A birthday attack is a type of cryptographic attack that exploits the mathematics of probability to find a collision in a hash function.

#### Secure Hash Algorithm

SHA (Secure Hash Algorithm) is a set of cryptographic hash functions designed by the US National Security Agency (NSA) and published by the US NIST as a Federal Information Processing Standard. There are several versions of SHA, including SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512, with the number indicating the length of the hash value produced. These functions are widely used to verify the integrity of data, such as the contents of a file, by comparing the calculated hash value with a known and expected value. The SHA-1 is based on principles similar to those used by Professor Ronald L. Rivest of MIT when designing the MD4 message digest algorithm and is modeled after that algorithm [4].

### MD5 Message-Digest Algorithm

MD5 (Message-Digest algorithm 5) is a widely-used cryptographic hash function that produces a 128-bit (16-byte) hash value, typically expressed in text format as a 32-digit hexadecimal number. It is commonly used to verify the integrity of data, such as the contents of a file, by comparing the calculated hash value with a known and expected value. MD5 was designed as secure replacement for MD4. However, MD5 has been shown to be vulnerable to collision attacks, where two different inputs can produce the same hash value, which weakens its effectiveness for secure applications [15].

### NTLM

NTLM (Network Lan Manager) is an authentication protocol used in windows for authentications between clients and server. NTLM is used to authenticate remote users. NTLM is a challenge-response authentication protocol. The authentication mechanism in this protocol works as follow: the server sends a challenge to the client, the clients sends back a response that is a function of the challenge, the user's password, and other information. Computing the right response require knowledge of the user's password. The server can validate the response by checking the database. NTLM password are considered weak because they can be brute forced, in fact uses one of two one-way functions, depending on the NTLM version: NT LanMan and NTLM version 1 use the DES-based LanMan one-way function (LMOWF), NTLMv2 uses the NT MD4 based one-way function (NTOWF) [9].

**MD4**

The MD4 Message-Digest Algorithm is a cryptographic hash function developed by Ronald Rivest in 1990. The algorithm takes as input an input message of arbitrary length and produces as output a 128-bit "fingerprint" or "message digest" of the input [14]. The security of this algorithm has been compromised, the first collision attack were shown by Dobbertin in 1996 [3] and newer attacks have been published since then. MD4 is used also to compute NTLM password-derived key on Microsoft Windows.

### 2.1.2 Enforce password security

A technique used to enforce password security is the use of hashed passwords and a salt value. A salt is a non-secret value used in cryptographic process used as an additional input to a one-way function [5]. When we need to store a new password in the system the password is combined with a fixed-length *salt* value. The hash algorithm is designed to be slow to make it harder for an attacker to brute-force the password. The hashed password is then stored, together with a plain-text copy of the salt, in the password file for the corresponding user ID.

If a user tries to log in he provides a ID and a password. The system uses the ID to search for the password of the user and retrieves the plain-text salt and the encrypted password. The salt and the password supplied by the user is given as input to the hash function. If the result of hash matches the stored value, the password is accepted.

Salt can be used to:

- Prevents duplicates of password;
- Increase the difficulty of dictionary attacks;
- Impossible to identify a person with password used in more devices.

In our work we will not cover thoroughly how to strengthen the security of a password.

# Chapter 3

# Password Datasets

This chapter illustrates the password datasets used in our work: Section 3.1 gives a brief introduction, Section 3.2 shows and explains the datasets used and finally Section 3.3 shows a brief analysis of the datasets and explains how we will use them.

## 3.1 Existing password datasets

Numerous datasets of leaked passwords have been created over the years. In our work we used these datasets to perform some attacks, in which these datasets are used. One of the main problems is selecting data-sets of real passwords useful for our study. This work is particularly complicated because you have to have access real password data-sets that have been leaked from Internet. A lot of leaks of real password are available for a limited amount of time and every data-set could be biased in various ways, e.g., to specify policies adopted. Another problem is that using public data-sets of real passwords to perform searches is not fully ethical and could lead to legal issues.

In our work we have considered several datasets: *RockYou* [16], *RockYou2021* [17], *Have i been pwned?* [6]. We chose these datasets because they are the largest datasets of real passwords available online. However, there are numerous other password datasets.

## Rockyou

Rockyou [16] is a well-known password data-set that was obtained in a data breach in 2009. The company used an unencrypted database to store milions of users account data, including plaintext passwords for its services, as well as password to connected accounts such as Facebook and MySpace and other services. RockYou wuold also sent the unencrypted password via email when it was necessary to retrieve an account. They also did not have a password policies and did not allow using special characters in the passwords. The databreach was possible by exploiting an old SQL vulnerability that had not been patched. The dataset contains approximately 14.341.564 unique passwords, used in 32.603.388 accounts that were leaked from the social networking site *RockYou.com.*

The Rockyou dataset is frequently used by security researchers and ethical hackers to study password trends and to test the strength and effectiveness of password cracking tools and techniques. The dataset provides insight into common password choices, such as simple and easily guessable passwords, as well as the prevalence of certain password patterns, such as numbers at the end of passwords.

Some common findings from analysis of the *RockYou* dataset include [24]:

- Password reuse: A significant number of users in the *RockYou* dataset were found to reuse the same password across multiple accounts;
- Weak passwords: The *RockYou* dataset contains many easily guessable passwords, such as "123456" and "password";
- Password length: The majority of passwords in the *RockYou* dataset are relatively short, with the majority being less than 10 characters in length;
- Character sets: The *RockYou* dataset contains passwords that use a variety of character sets, including alphanumeric, special characters, and non-English characters;
- Password complexity: The *RockYou* dataset contains many passwords that are easily guessable.

The *Rockyou* dataset has also been used to demonstrate the importance of using strong and unique passwords, as well as the need for implementing security measures such as password salting and hashing to protect sensitive information. Rockyou dataset is also included in Kali Linux, and since its launch in 2013.

## Rockyou2021

*Rockyou2021* [17] dataset password list is typically dictionary of real used password used for password cracking attacks. This data-set is a combination of all password from different sources: CrackStation's Password Cracking Dictionary, Hack3r.com's Wikipedia Wordlist , Daniel Meissler's SecLists/-Passwords, berzerk0's Probable Wordlists, Passwords from Weakpass and COMB (Compilation of Many Breaches): 3.2 million previously shared passwords. All passwords are 6-20 characters long, all lines with non-ASCII characters or white space or tab are removed resulting in 82 billion unique entries. The list does not contains any username or other personal information associated with the password. The advantages compared to RockYou is the large size of it that makes it very suitable for attacks. This dataset however, is very recent and it is not protected by hashes.

## Have i been pwned?

*Have I Been Pwned* [6] is a website that allows Internet users to check if their personal data has been compromised by data branches. The service has hundred database dumps and pastes containing information about billions of leaked accounts.

The *HIBP* dataset is compiled from various sources, including publicly available breach data, data contributed by breach victims and organizations, and data obtained through partnerships with other organizations. The general idea behind was to provide the general public with a means to check if their private information has been leaked or compromised. In late 2013, security expert Troy Hunt with this idea was analyzing different data breaches for trends and patterns and realized that users might not be aware that their data had been compromised. The first data breaches included was the Adobe System security breach that affected 153 million accounts in October 2013.

Since his launch the dataset is continuously updated as new breaches are reported, and includes information such as usernames, email addresses, passwords, and other personal information.
What's alarming is that a significant number of users in the HIBP dataset were found to reuse the same password across multiple accounts, making them vulnerable to attacks that target a single account. Have I Been Pwned? dataset contains 306M passwords wichich could be accessible via a web search or downloadable.

## 3.2 Dataset Analysis

There are three different database analyzed as mentioned:

- *RockYou*
- *RockYou2021*
- *haveyoubeenpwned* called *hbp*

Datasets were checked and manipulated to create a new dataset containing all unique passwords. Creating a unique dataset with all the unique passwords extracted from the datasets allows us to have a sample for our tests large enough. The datasets were compared using hashcat, a dictionary attack was performed with rockyou2021 as the dictionary and rockyou as the target. With this simple check it was possible to verify if a dataset is contained in another dataset [2]. The first check was made to check if rockyou is contained in rockyou2021. The analysis showed that 98.73% of passwords contained in rockyou is contained in rockyou2021, some passwords are not contained because not well-formed. The second test was done to check if rockyou2021 is contained in hbp, the analysis shows that 85.43%, $723M$, rockyou passwords are contained in hbp.

Hbp provides very interesting information about hash frequency, has been recomputed the hash of the 723M password, was searched in the hbp the corresponding hash to associate the frequency to the corresponding plaintext password. This allowed to build a plaintext of 723M well-formed real password with associated frequency. The new dataset has been called $D$.

For purely cognitive purposes in the Table 3.1 are shown the 10 most used passwords extracted from the newly created dataset of 732M passwords.

| **Password** | **Frequency** |
|---|---|
| 12456 | 373591195 |
| 123456789 | 16629796 |
| qwerty | 10556095 |
| password | 9545824 |
| 12345678 | 5119355 |
| 111111 | 4833228 |
| qwerty123 | 4759446 |
| 1q2w3e | 4456640 |
| 1234567 | 4043126 |
| abc123 | 3891152 |

Table 3.1: Most frequent password of 732M new dataset $D$ [2]

# Chapter 4

# Password cracking

This chapter explains what a Password cracking is: Section 4.1 gives an introduction to the topic, Section 4.2 shows the previous work done on Password cracking, the tools used and finally Section 4.3 shows the different types of attacks.

## 4.1 Password cracking definition

In a password based authentication system when a user enters a password, a hash of the entered password is generated and compared with a stored hash of the user's actual password. If the hashes match, the user is authenticated. Password cracking is the process of recovering passwords from password hashes stored in a computer system or transmitted over networks. It is usually performed during assessments to identify accounts with weak passwords. Password cracking is performed on hashes that are either intercepted by a network sniffer while being transmitted across a network, or retrieved from the target system, which generally requires administrative level access on, or physical access to, the target system [22]. Once these hashes are obtained, an automated password cracker rapidly generates additional hashes until a match is found or the assessor halts the cracking attempt .

Password cracking is also used to identify and evaluate password security policy and take countermeasures to increase protection. There are different approaches, such as trying to guess the password and checking if it is a hash of a known password.

There are different types of attacks that can be carried out to crack a password.

**Online guessing attack:** A type of password guessing attack where the attacker tries to guess the password for a target account by sending repeated request to the target system. In this attack, the attacker use such as *offline guessing attack* a list of commonly used password, or a combinations of letters to generate potential password. The advantage of online guessing attack is that the attacker can make use of information obtained from the target system during the attack, such message error obtained from the system or delay in the response that can be used to determine when a guess is correct or not. Online guessing attack can be seen as a legitimate attempt by the system and can be ignored. Online guessing attack also have limitations. Such as the risk of detection and mitigation by the target system and the requirement that the attacker have the necessary information to access the target account, such as the username or email address. The online guessing attack can be very slow because the process can be slow for long and complex passwords and we need to wait for the response from the targeted system.

**Offline guessing attack:** A type of password cracking which uses a re-computed dictionary of password already hashed, or computed on the way, to match with the targeted hash without having to make request to the target system. In this type of attack, the attacker obtain the hashes form breaches or social engineering, and use a password cracking tool, to match the hashes against a large dictionary of password.
Offline Guessing attack is faster and more efficient than an online attack, because the attacker doesn't need to wait for the response form the targeted system for each guess. The attacker can also perform attack on a separate system which reduce the risk of detection and mitigation. Offline guesses also have limitations, such as the necessity for a large database of hashes and the requirement that the hashes to be crackable, either because they use a weak hashing algorithm or because they lack proper security measures such as salting.

## 4.2 Previous work

Morris and Thompson in 1979 [10] proposed one of the most widely used attacks to this day: the dictionary attack which consists in using a dictionary of common words as the initial basis. The new dictionary attacks combine these popular words with mangling rules, i.e. rules that modify words and prodocued variations of passwords. A mangled wordlist attack consists of using both common word files usually from password leaks and rule files that are applied to generate new sets of possible passwords.

In [8] Liu et al. proposed a new technique to be used, based on Rules attacks using hashcat and johntheripper. The proposed technique could be used to verify the strength of a password by estimating the number of guesses needed to crack it. The tool is based on the module inversion rule that calculated the preimage set for a rule given a password. The paper studied how it was possible to reorder dictionaries and rules to get better results. The idea was to use more efficient words and rules to crack more passwords faster.

In [25] Weir et al. they discussed a method that, starting from the probability distribution of users' passwords, could generates password patterns in descending order of probability. They created a probabilistic context-free grammar (PCFG) based upon a training set of previously disclosed passwords which generates word mangling rules, and form them, password guesses to used in password cracking. This technique has been extended to compute password guessability, i.e. the number of guesses that is required by a cracking algorithm with particular training data to guess a password. The efficiency of this technique has been increased thanks to Komanduri [7] and modified so that it is easier to use by Narayanan and Shmatikov [11].

In [23] Segreti et al., compare password guessability models with password recovery tools. He founds that each approach was highly dependent on its configuration. He said that is better to run various algorithm and choose a minimum conservative value to measure password guessability. He resort expert to share the mangling rules.

## 4.3 Password cracking

Password cracking is done using some tools, the two most popular are *john the ripper* and *hashcat* [20]. These two tools can be used to recover a hashed password using various techniques, including brute force attack. In addition, these two tools are optimized and sophisticated enough to reduce the time it takes to crack a password.

**John the ripper** is a free and open-source password cracking tool that is commonly used for testing the strength of passwords and for cracking password hashes. It supports several operating systems including Linux, macOS, and Windows, and it can run on both CPUs and GPUs for faster cracking. John the Ripper supports multiple hash formats and cipher modes, including: Unix crypt(3) hashes, Microsoft LM hashes, MD5, SHA-1, and other hash algorithms.
It can use various attack modes, such as dictionary attacks, brute-force attacks, and combination attacks, and it can also incorporate rules that can been seen as functions used to modify, cut or extend wordlists (files containing words found in a dictionary or real passwords cracked before) to generate more combinations.

**Hashcat** is an open-source password cracking tool used by security professionals and ethical hackers to recover lost or forgotten passwords.
Hashcat can work on various platforms including Windows, Linux, and macOS. Hashcat can be used for simple attacks as a dictionary attack or brute force attack, but can also perform much better attacks such mask attack that

try all combinations in a specific key space or rule based attack which applies basic transformation to the word of a dictionary and creates more complex passwords.

Hashcat and johntheripper are highly optimized and take full advantages of the parallelism given by multicore processors and GPUs.

## 4.4 Cracking attacks

Hashcat and johntheripper can carry out different types of attacks on passwords, some more complex than others. In our study we focused on the use of **hashcat**. Below the attacks are explained from the point of view of hashcat. The different types of attacks can be carried out similarly on johtheripper.

### 4.4.1 Brute-Force attack

A *brute-force attack*, generates all possible passwords up to a certain length and their associated hashes. Since there are millions of possible combinations of characters, it can take months to crack a password. This can be a time-consuming process, but with the increasing power of computers, it is becoming easier to perform brute-force attacks on even strong passwords.

However in Brute-Force attack we can have a long time to crack a single password because we have that for every possible password of length $L$ the total number of combinations for which we have to iterate is:

$$N^L$$

where $N$ is the Number of Chars in char-set that contains all upper-case letters, all lower-case letters and all digits.

**Example** We want to crack the password: *password123*
The password length is 11, and the char-set is composed of 62 char. So we need to iterate for $62^{11}$ (520.3660.683.837.093.888) times. We crack with rate of $100M/s$, requires more than 5 years.

### Mask attacks

Mask attack try all combinations from a given keyspace. It is a particular case of brute-force attack. Mask attacks is used to reduce the password candidate keyspace to a more efficient one. Passwords usually use a common pattern. A number or year added at the end of a password is one of the simplest combinations, we can configure a mask to only try numbers at the end. For instance, if we have a password composed by 6 lower characters and 2 digits at the end, we can use a mask that places any combinations of 2 digits after 5 lowercase characters, i.e., only $26^6 \cdot 10^2$ possible combinations of passwords. Without password policy adopted a password frequently used consists only of lowercase letters or numbers (i.e. PIN). If instead a policy is adopted we can have different combinations with some stronger than others. An example where masking can be applied is that if default passwords schemes use a known set of characters.

**Example** We want to crack the following password example using a mask: *password123*
It has length of 11 characters and for each one, it could be upper-case (26

potential characters), lower-case (26 potential characters), a symbol (33 potential characters) or a number (10 potentials characters, we had to try $95^1$1 combinations.
If we suppose we known the last three characters are numbers. This would drastically reduce the potential keyspace as no passwords with any letter or symbol in the last three spaces would need to be tried.

**Generation of mask** A password mask is a string that displays the position of character candidates in a password.
To generate a mask for a password we want to crack we need specify which character set you want to use in which position. The specify characters is described by a placeholder which is a sequence of symbols, for example, the "?" placeholder can be used to represent any single character. The placeholder number to be entered is the length of the password taken into account.

- A mask is a simple string that configures the keyspace of the password candidate engine using placeholders;
- A placeholder can be either a custom charset variable, a built-in charset variable or a static letter;
- A variable is indicated by the ? letter followed by one of the built-in charset $(l, u, d, s, a)$ or one of the custom charset variable names $(1, 2, 3, 4)$;
- A static letter is not indicated by a letter.

**Example** We want to generate the mask to crack the password: *password123* We known that the password is formed by 11 characters. Let's assume we also know that the last three characters of the password are 3 numbers. Knowing these two information we can build a mask with charsets $?l$(lowercase-letter) and $?d$ (digits) as shown:

?l?l?l?l?l?l?l?l?d?d?d

This mask will allow us to try all combinations from *aaaaaaaa*000 to *zzzzzzzz*999 excluding all combinations that do not respect the mask.

A typical mask attack is made when we have an idea of which guideline a password follow. For example, we can use a mask attack if we know the length a password should have and if there are numbers entered at the end of the password. Using this knowledge we can build several masks that can, through brute-force, crack passwords that we could not find. Another possible attack could be to generate masks based on cracked passwords from a previous attack. In this way we can use the generated masks to try to crack passwords that we could not find in the previous attack. Using masks allows us to exclude many character combinations and speed up cracking.

### 4.4.2 Dictionary attacks

In a *dictionary attack*, a pre-computed list of common passwords and words is used to crack the password. This method can be much faster than a brute-force attack, it only tries passwords that are known to be commonly used. All that is needed is to read line by line from a textfile (aka “dictionary” or “wordlist”), generate the hash of the read line and try each line as a password candidate [20].

Several leaked passwords datasets are used as dictionaries. An example of a dictionary used in our research is *RockYou* (see Chapter 3).

### 4.4.3 Rule-based attacks

The rule-based attack is one of the most complicated of all the attack modes. The rule-based attack is like a programming language designed for password candidates generation. It has functions to modify, cut or extend words and has conditional operators to skip some, etc. That makes it the most flexible, accurate and efficient attack.

**Example** Two example of Rules that are used can be the lowercase all letters $l$ and reverse the entire word $r$. Suppose that we have the password: *Password123* We want to lowercase all the characters with rule $l$ and we obtain the new candidate password:

*password123*

Using the rule $r$ we obtain the password:

*321drowssap*

The rule-engine in Hashcat was written so that all functions that share the same letter-name are 100% compatible to John the Ripper and PasswordsPro rules and vice versa.

There are different functions that can be used in the Rule-based attack. To generate rules from a dataset is possible to use *maskprocessor*. Another option is generate rules randomly.

There are some popular off-the-shelf rules used in hashcat, best64 and generated2 are the most common and used.

**Probabilistic context-free grammar rules** Wier et al. in [25] describe a method to create a probabilistic context-free grammar based upon a training set of previously disclosed passwords. This grammar then allows us to generate word-mangling rules (which are used to modify or "mangle" words producing other likely passwords), and from them, password guesses to be used in password cracking. PCFGs are very useful for password cracking and has several advantages. Firstly, they can be used to model the structure and patterns of passwords in a more accurate way, and allows more efficient and effective password cracking for complex and long password. PCFGs can also be used to generate passwords that are more likely to be accepted by the system because they are based on the system's password rules. This is very useful when the system requires and apply some restriction on password length and character sets. PCFGs also have some limitation, firstly are computationally intensive to be generated and may not always reflect the underlying structure and patterns of passwords, leading to generation of invalid password and by consequence less effective cracking.
A typical mangling rules involve such as capitalizing the first letter, replacing characters or more complex operations.

### Precomputed rules

**Best64** [20] is a set of rules used by Hashcat to generate password candidates for cracking hashes. The "best64" rule set is considered one of the most

effective rule sets in Hashcat, and is designed to generate a large number of potential passwords while still being relatively fast to execute.
The best64 rule set uses a combination of simple and complex transformations, such as character substitutions, case changes, and word concatenation, to generate potential passwords. The rules are designed to cover a wide range of common password patterns and to account for common mistakes and typos that users may make when selecting passwords. Length of the rule set: 77.

**OneRuleToRuleThemAll** [1] is a password cracking rule set. The goal of this rule set is to generate as many potential passwords as possible, based on the idea that "one rule to rule them all." This rule set applies a large number of different transformations and manipulations to the input wordlist, generating a large number of potential password candidates. The "OneRuleToRuleThemAll" rule set is often used as a starting point for password cracking attempts, as it provides a comprehensive coverage of different password patterns and structures. Length of the rule set: 52.014.

**Pantagrule** [13] is a series of rules for the hashcat password cracker generated from large amounts of real-world password compromise data. While Pantagrule rule files can be large, the rules are both tunable and perform better than many existing rule sets. The rules were generated based on the number of occurrences and then filtered using various test data sets, discarding the rules that were not used for cracking or taking the best performing ones. This has led to the creation of different sets of rules *popular.rule*, *random.rule*, *hybrid.rule*, *one.rule.*

### TRule

The idea form previous work [2] was to train a rule set starting from an existing one. The new rules have been ordered in order of frequency to increase effectiveness by selecting the most used. Generated rules have been combined with a large dictionary in order to speed up the cracking of the password.
The starting point of the rule is to begin with the rule sets provided by hashcat distribution "OneRuleToRuleThemAll" and the best of "pantagrule". The initial set was formed by about 779K unique rules. Before the rules training it was taken the leaked password dataset (formed by $732M$ passwords) and was divided into two separate parts of different size. The largest part (about 80%) was called $Train_D$. In order to train rule is used *TrainD* as larger dictionary divided in two sets of equal size $TrainDic_D$ and $TrainPwd_D$ and the initial 779K rules. The attack on this dictionary was stopped at $10^{12}$ guesses. Using hashcat debug function the rules are sorted by decreasing frequencies and unused rules have been deleted, reaching 641K rules. From the 641K rules 52014 were taken, which is the number of rules in OneRuleToRuleThemAll. By comparing generated and existing rules, the new rules lead to better results.
The best tradeoff of rules needed is calculated by:

$$G = D \cdot R$$

where G is the target guesses, D is the size of the dictionary and R is the size of the rule set.

To look for a point of balance between the number of rules and the size of the dictionary to take, to stay in the $10^{12}$ guesses, the whole set of rules of TRule formed by 640K rules was taken and a small subset of the initial dictionary. It was then noticed that success rates increased as the rules decreased and the size of the dictionary increased.

The size of rules after some experiment show that there is not a tradeoff and the best result is obtained by taking the complete dictionary and compute the size of R:

$$R = \lfloor G/D \rfloor$$

Starting from the 3726 best rules of TRule, which was the biggest R for $TrainDic_D$ a rule training was redone. This new set of rules was called TRule2. Trying to re-crack passwords using TRule2 as the new rule set, a success rate of 71.11% was achieved using only 1728 rules.

# Chapter 5

# Experimental results on password cracking

This chapter explains the test performed and the data obtained from the tests: Section 5.1 from a small introduced to the topic, Section 5.2 explains the result obtained in the previous work, Section 5.3 explains the basic idea and Section 5.4 presents the tests performed and analyzes the results.

## 5.1 Test setting

Password cracking is the process of attempting to gain unauthorized access to a computer system or network by guessing or cracking passwords. This is typically done by using brute-force or dictionary attack methods. The results we are presenting were obtained with the *hashcat* distribution (non-legacy) but we are confident that similar results can be reproduced using *johntheripper*.

In our tests we focused on attacks that combine a dictionary attack with rules attack.

In previous experiments [2] Di Campi et al. it was shown that with $10^{12}$ of guesses it was possible to reach a 71.11% of cracked password. This result was obtained by training the rules on a $732M$ password dataset, obtaining rules, called $TRule2$, very optimized. It was also shown that using the largest possible dictionary and fewer rules could lead to a greater result.

To carry out the tests we used two different datasets of passwords both obtained from $D$ and separated from each other.

**Dictionary and Target uset:** In *The revenge of Password crackers: Automated Training of Password Cracking Tools* [2] the data-set formed by 732M unique password is split in two different disjoint subset, one used for the training and one used for the test as represented in the following table.

| **Subset Name** | **Dimension** | **Size** |
|---|---|---|
| $Train_D$ | 579.008.917 | 80% |
| $Test_D$ | 144.740.239 | 20% |

In our approach we did the same to compare the results, we used $Train_D$ as dictionary and $Test_D$ is used as target.

**Rules:** In our experiments we used the rules created in the previous work *TRule* and *TRule2* [2] created with the procedure described in the Section 4.4.3. TRule was created by training, using all the rule sets provided in the hashcat distribution plus OneRuleToRuleThemAll and popular.rule of pantagrule, sorted by descending frequencies. $TRules2$ is a set of rules trained using the 3726 best rules of $TRule$ and $TrainDic_D$. TRule was taken because $TRule2$ in some tests required a number of rules bigger than the one in TRule2.

In the following table we have a summary of what was used.

| **Name** | **Description** | **Dimension** |
|---|---|---|
| $Test_D$ | Target of the attack | 144.740.239 |
| $Train_D$ | Dictionary | 579.008.917 |
| $TRule$ | All rules used | 640.939 |
| $TRule2$ | Subset of TRule trained | 3726 |

Table 5.1: Summary of what we have used

The passwords in the target are stored encrypted with the *NTLM protocol*, which use MD4 cryptographic hash function.

*Experimental setup.* Our experiments have been done on a server equipped with an Intel(R) Xeon(R) E5-2699 v4 @ 2.20GHz with 88 CPUs and 256G of RAM. No GPUs used. Time for each experiment varied from 40 min to 3 hours.

## 5.2 Previous experiment

The target set in previous works [2] was to reach the maximum possible cracked passwords in $10^{12}$ guesses. This objective was set because it allowed to do experiments on large amounts of passwords in a reasonable time of 2-3 hours. In the previous work to achieve a high level of cracking rate a detailed analysis was made of the main strategies used to crack passwords. An initial analysis focused on off-the-shelf rules and a 732M dictionary of real passwords. From an accurate analysis of the strategies it has been decided to perform a training of the rules based on the dataset of 732M of passwords. By training the rules it was possible to obtain a very large set of rules, called

TRule, with a success cracking rate of 51.87% was achieved outperforming the rules off-the-shelf. From these rules was extracted the 3726 best rules, which was the biggest numbers of rules to be taken to reach $10^12$ guesses. These rules have been trained on cracked passwords to try to get a bigger result and renamed $TRule2$. In the end a final result of 71.11% was achieved using these rules (TRule2). In the previous work it was shown that using all the 3456 rules from TRule2 with half of the dictionary is not worth and gives only 65.73% of cracked password. Instead using the TRule2 with only 1728 rules and the full dictionary in $10^{12}$ gives better result return 71.11% of cracked password (see Figure 1).

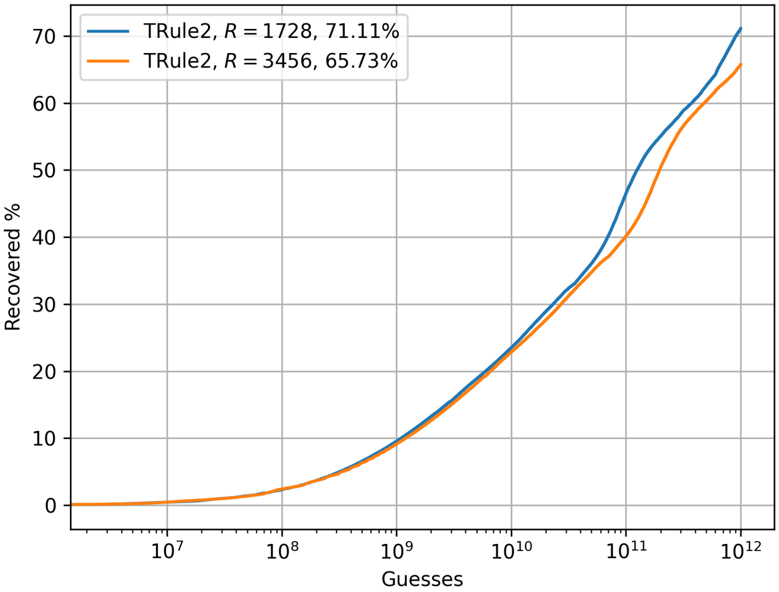

Figure 1: TRule2 reach 71.11% success rate [2]

The results of 71.11% cracked passwords was impressive because it exceeded all the results obtained with the different sets of rules already generated and compared. It was also done a mask training on cracked passwords always using hashcat. It has been calculated that dividing a part of guesses between masks (36.87% of guesses) and rules with dictionaries (63.13% of guesses) was possible to circumvent a guesses rate of 72.67%.

## 5.3 Our new algorithm

Our idea was to start from the result of [2] and to understand if it is possible to achieve an even greater cracking rate on terms of cracked password or a decrease in time using different techniques. As a starting point we set $10^{12}$ as a target of guesses in order to compare the results existing with the ones. The initial idea was to check if after a password cracking, it was possible to use cracked passwords as a new dictionary also using the rules from previous work [2], to try to get a higher percentage of password cracked. As the target of the new attempt we used passwords not cracked from the previous stage. For simplicity, each step will be called Iteration.

**Algorithm 1** Algorithm "Improve cracking"

```
$target \leftarrow 10^{12}$
for $G \leq target$ or $|D| = 0$ do
    CP = PasswordCracking(T,D,R)
    $D_1 = CP$
    $NC = T - CP$
    $T = NC$
    $D = D_1$
    $G+ = |D| \times |R|$
end for
```

We denote as G the number of guesses, D is the dictionary, and T the target of the attack. $PasswordCracking(T, D, R)$ is a function which start $Hashcat$ with the target $T$, the dictionary $D$ and the set of rules $R$. The algorithm works in the following way: Perform a password cracking on a target $T$ with a dictionary $D$ and some rules $R$, save the cracked password $CP$. Create a new dictionary called $D_1$ with the password cracked from the step before. Extract the password not cracked $NC$ from the initial set $T$ by subtract from the target the password already cracked. Use the new data-set of password not cracked $NC$ as target, $D_1$ as new dictionary and the rules $R$ form previous iterations.

The number of rules to be taken for all iterations is not necessarily the same. This is due to the fact of the relationship $G = R \cdot D$ and we want to fix $G$ to $10^{12}$.

In general we have several strategies to choose the number of rules to be taken for the next iteration.

- Keep the same rules for each iteration: this is the simplest strategy. The problem with this strategy is that by iterating you get fewer and fewer cracked passwords and the number of guesses made for that iteration will decrease. In fact, the relationship $G = R \cdot D$ will start to become smaller and smaller because of the smaller size of the dictionary, which we remember are only cracked passwords from the previous iteration.

- Increasing the number of rules: this is a more complex strategy that allows each iteration to keep a target of guesses. However, it requires each time to recalculate the number of rule needed and to initially set a number of guesses per iteration. In some cases it also requires you to completely change the rules.

Using one of these strategies we have to verify if it is possible to obtain a number of cracked passwords in the defined target $10^{12}$.
With our idea we are trying to arrive at two hypothesis:

1. Get the same number of password cracked (71.11%) from previous work [2] but with less guesses done, we tried to use less than $10^{12}$ guesses;

2. Get more cracked passwords (more than 71.11%) from previous work [2], we expect $10-20\%$ of more password cracked, with the same number of guess $10^{12}$ or less.

In this thesis we tried to achieve at least one of these two hypothesis.

## 5.4 Experimental Results

In the previous chapter we described what our approach can be. We takes two sets of rules called TRule and TRule2 from previous work [2] and the same dictionary is used. We divided the experiments into two parts because it allows us to consider different rules and how to use them. From the previous described algorithm we can consider our work as a "Dictionary Training", because we use what we get from the previous Iteration to perform an attack aimed at not cracked passwords.

### 5.4.1 Experiment 1

The first test was to see if our idea could be applied. To verify this hypothesis we left out the target guesses that we had set. Recall the relation of the target guesses

$$G = D \cdot R$$

where G are the number of guesses, D the length of the dictionary and R the length of the rules. In the previous work with the target of $10^{12}$ the result of 71.11% was obtained by taking TRule2 and extracting a certain number of rules, the rules were obtained by taking the whole dictionary (579.008.917 words) and calculating the number of rules to get to the target.

$$R = \frac{10^{12}}{579008917} = 1728$$

We perform two iterations using the same set of rules (1728) extracted from the TRule2.. With the first Iteration we get 71.11% (102.918.340) of recovered password. In the second iteration we get 5.61% (2.344.725) of recovered passwords. The total percentage of password cracked must be calculated by adding the number of previous password cracked with the password cracked in this iteration and divide by the total number of password in the dataset. Adding up the two results we get the final success rate of 72.73% of cracked password. This result is a increase of password cracked of 1.62% form the initial 71.11% (see Figure 2).

The result shows that iterating sightly improve existing results. Remember that this is not a real representation because with two single iterations we surpass the $10^{12}$ guesses and we reach $1.17 \times 10^{12}$. With this experiment we were able to show a cracked password increment of 1.62% compared to the previous work [2].

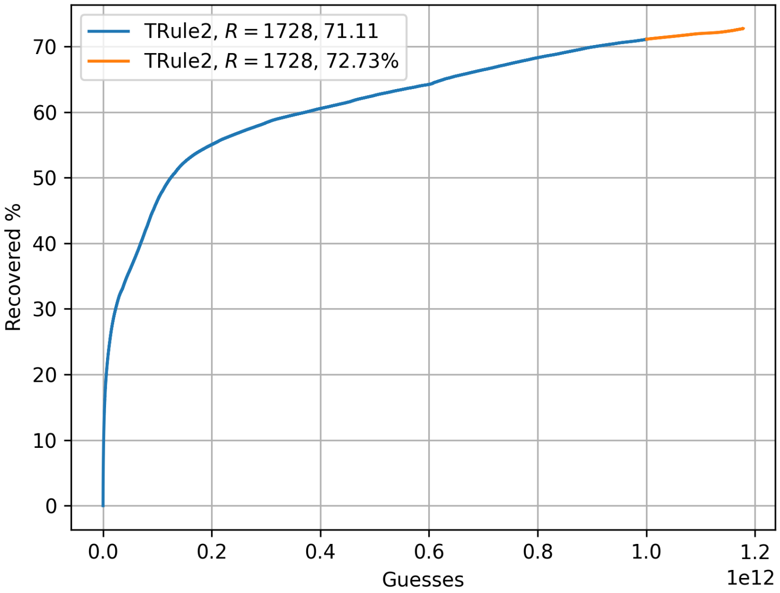

Figure 2: TRule2 with Iteration reach 72.73% of success rate

### 5.4.2 Experiment 2

In the previous test we had not set a target of guesses. Now let us set a target of $10^{12}$ guesses to run different iterations to try to crack as many passwords as possible always remembering the rule $G = R \cdot D$. First we need to set the initial target for our first iteration. The selection of the first iteration is one of the crucial parts because it allows us to focus on the type of iteration to be performed later. Selecting a target too high or too low for the first iteration could lead to not optimal results. We recall in particular a statement "We could conclude that with bigger dictionaries (and smaller rule sets) the success rate would increase even more" [2]. If this statement is correct it would mean that at the first iteration we have to try

to crack as many passwords as possible, because at the second iteration we need a dictionary as large as possible. In the first iteration we must take into account the ratio of cracked passwords and number of guesses, because having a large number of guesses at the second iteration allows us to try with more rules.
Once the starting rule is decided, it is necessary to decide whether to keep the same number of rules for all iterations or increase them.

### Fixed Rules

The first solution tested was to keep the number of rules fixed. Remembering the rule $G = D \cdot R$, we set our first iteration to $10^{11}$ which is about 1/10 of the target guesses prefixed. Using the previous report we calculate the number of rules needed.

$$R = \frac{10^{11}}{579008917} = 172.7 \approx 173$$

So let's take the first 173 rules from TRule2, $Test_D$ target and $Train_D$ dictionary.

We modify the algorithm so that it allows us to iterate with fewer rules.

**Algorithm 2** Algorithm "Fixed Rules"

```
$target \leftarrow 10^{12}$
$R \leftarrow TRule2\_173$
for $G \leq target$ or $|D| = 0$ do
    CP = PasswordCracking(T,D,R)
    $D_1 = CP$
    $NC = T - CP$
    $T = NC$
    $D = D_1$
    $G+ = |D| \times |R|$
end for
```

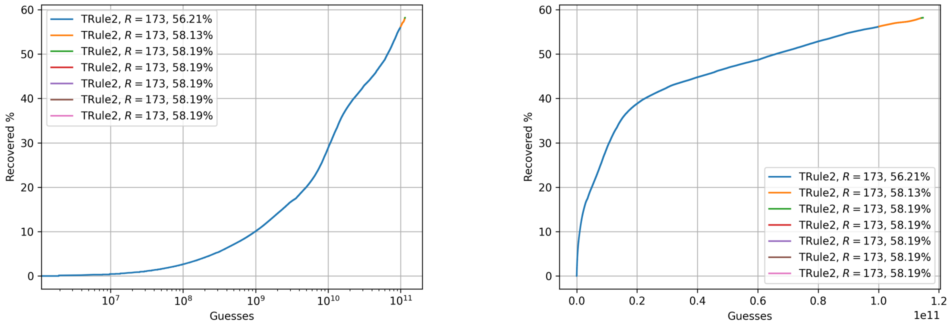

Figure 3: Fixed rule iterations with 173 rules from TRule2

In the Figure 3 we can see the result of the new algorithm with 173 rules taken from $TRule2$. At each iteration we noticed that the number of cracked passwords decreases more and more, this is due to the decrease in the size of the dictionary for each iteration and the constant number of rules. After 7 iterations the percentage of cracked passwords becomes 0%, it is therefore shown that using the same number of rules and iterating does not lead to satisfactory results. With these iterations we can not reach the target of $10^{12}$

guesses and we stop there. We get with 173 rules as final result 58.19% of password cracked which is not a great result and does not reach the target of guesses prefixed.

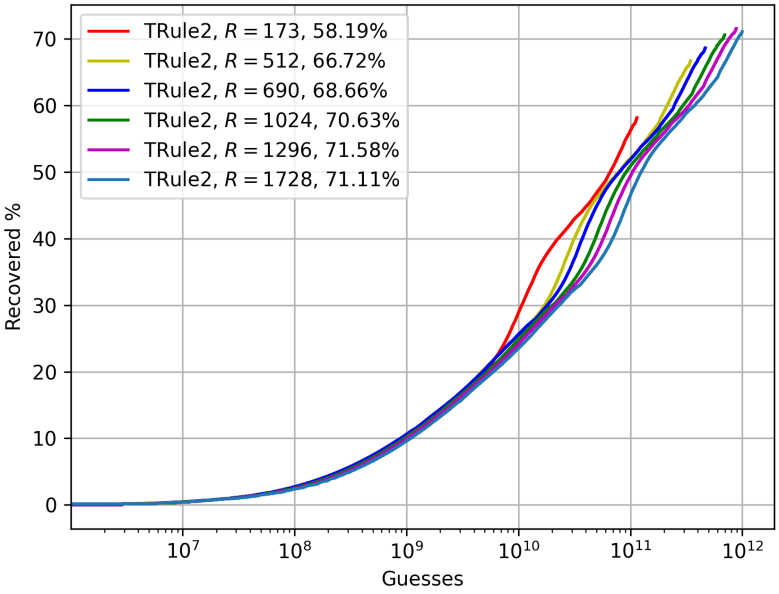

Figure 4: Success rate logarithmic for fixed rule with iteration

We also tried a different number of rules (see Figure 4) to see if the previous statement was correct. The choice of the various dimensions of the rules was made observing the work done previously $(512 - 1024)$ and dividing the number of maximum rules (1728) in order to get several steps. In the case of the 1296 rules the first three-quarters of rules were taken from TRule2. By comparison it turns out that with the different sets of fixed rules we get a maximum result of 71.58% with the number of rules $R = 1296$. The is the best success rate achieved so far, we managed with fewer rules to get a higher result than the previous one of 71.11% [2] and we can confirm hypothesis (1). We also made a lot less guesses than before with $0.9 \times 10^{12}$ guesses and we achieve hypothesis (2).

The Figure 4 show the result, in a logarithmic form, of our iterations with the sets of rules that we have choose. We have shown also the result in a linear scale in Figure 5 because the result are more understandable and show better the number of guesses.

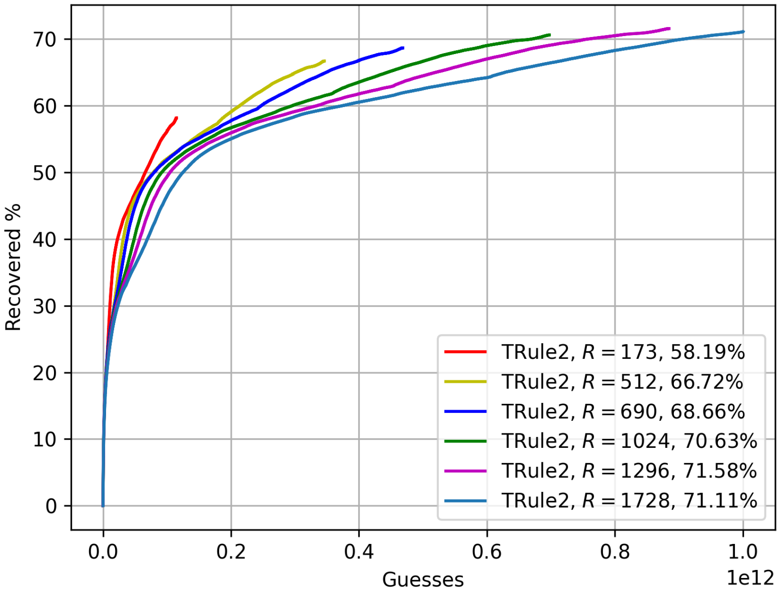

Figure 5: Success rate linear for fixed rule with iteration

### Dynamic Rules

Using this approach after calculating the first set of rules and performing the first iteration, we decided to recalculate the number of rules to get to $10^{12}$ guesses. The reason for recalculating the rules is due to two main reasons: the number of guesses we want to make at each iteration and the size of the dictionary of each iteration that decreases. If we want to keep the number of guesses fixed for each iteration we have to recalculate each iteration the number of rules needed. Also in this approach we had to make use not only of $TRule2$ but also of $TRule$, because decreasing the dictionary we need more and more rules. Stopping at 3726 rules of $TRule2$ would lead to the same situation as described before, after some iteration. So we had to modify the algorithm in the following way to recalculate at each iteration the number of rules needed.

**Algorithm 3** Algorithm "Dynamic Rules"

```
$target \leftarrow 10^{12}$
$TR \leftarrow TRule2$ or $TRule$
$TG \leftarrow 10^{n}$
for $G \leq target$ or $|D| = 0$ do
    R = extractFrom($TR, TG/|D|$)
    CP = PasswordCracking(T,D,R)
    $D_1 = CP$
    $NC = T - CP$
    $T = NC$
    $D = D_1$
    $G+ = |D| \times |R|$
end for
```

Where $TG$ is the number of guesses per iteration to be done, $TR$ is the TRule or TRule2 and *extractFrom* is a function that extract the first $n$ rules from

a set of rules. To calculate the number of rules to use for each iteration we must divide the target of guesses prefixed by the size of the dictionary (the cracked passwords from the previous iteration).
For instance an example is when we decide to keep 1/10 (or $10^{11}$) for iteration, until we reach $10^{12}$ guesses. The first iteration will require 173 rules, the second iteration needed to perform $10^{11}$ guesses, calculating with the usual formula we get that need 1129 rules. To perform this test after the first iteration we use the TRule, because TRule2 was not big enough. The Figure 6 show the result of this approach.

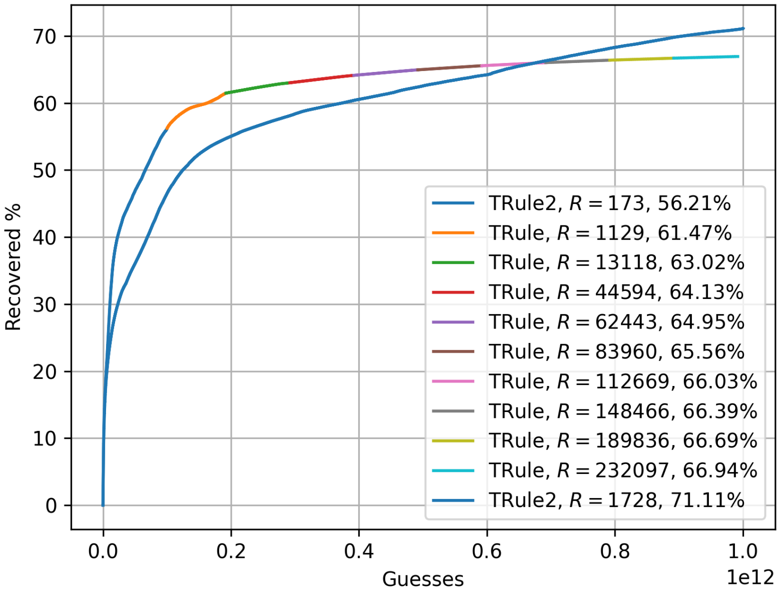

Figure 6: Success rate after 10 iteration until we reach $10^{12}$ guesses

At the end of the 10 Iterations we obtain a success rate of 66.94% which gives us a decrease of success rate compared to the previous work 71.11% [2].

Another approach was to divide the number of guesses into two iterations of different sizes. The reason for this choice was to check if it was possible to obtain a greater or lesser result by cracking more or less passwords in the first iteration.
To run the new test we have to change the algorithm as follows.

**Algorithm 4** Algorithm "Dynamic Rules Modified"

```
$target \leftarrow 10^{12}$
$TR \leftarrow TRule2$ or $TRule$
$TG \leftarrow 10^n$
for $G \leq 10^{12}$ or $|D| = 0$ do
    R = extractFrom($TR, TG$)
    CP = PasswordCracking(T,D,R)
    $D_1 = CP$
    $NC = T - CP$
    $T = NC$
    $D = D_1$
    $G+ = |D| \times |R|$
    $TG = G - TG$
end for
```

Assuming that with bigger dictionaries (and small rule sets) the success rate would increase even more [2], we can try to crack the biggest number of password in the first Iteration to get a bigger dictionary to be used in the second iteration. Gives this assumption we can take half the guesses $10^{12}/2$ for the first Iteration and the remaining guess for the Second Iteration. Starting always from the assumption that the best result that we can get is from 1728 rules form $TRule2$ and the entire size of the dictionary. We took 864 rules (half of 1728, $(10^{12})/2 \cdot 579008917 = 864$) and we perform the First Iteration. We get a result of 68.05% of cracked password.

Taking the previous result as starting point with the new dictionary the cracked password we need to recompute the number of rules to reach the $10^{12}$ guesses. The rules needed are $R = (10^{12}/2)/98501818 = 5076$. In the algorithm the calculation is done in the $TG$ variable. Performing the new Iteration we can reach a result of 71.83%, (see Figure 7) that show a small increments compared to 71.11% obtained in the previous work [2].

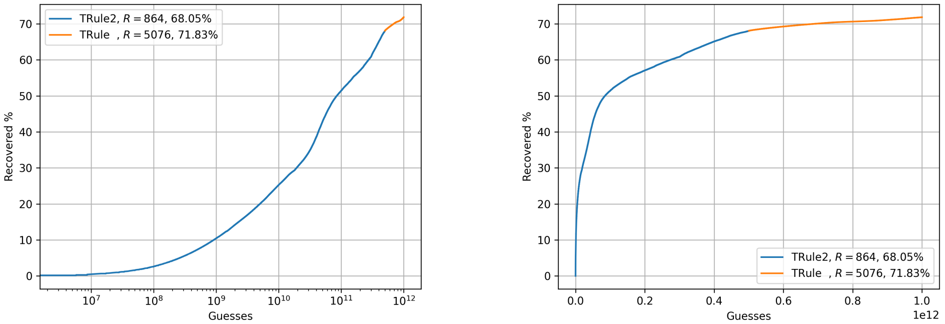

Figure 7: Test with dynamic rule with two iterations with half guesses each

As a second attempt we tried to get even more cracked passwords in the first iteration. We therefore decided to divide the number of guesses in the following way: 3/4 of guesses in the first iteration, 1/4 in the second. The results in the Figure 8 show 72.13% of password cracked with $10^{12}$ guesses, this result confirm the 1° hypothesis.

This is one of the best results achieved so far with 72.13% of total of cracked password, an increments of +1.02%.

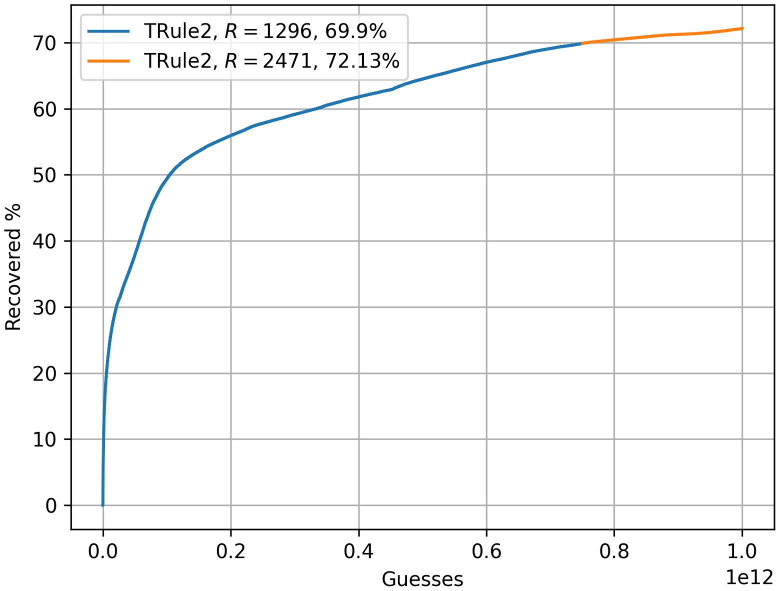

Figure 8: Test with dynamic rule with two iterations and TRule2

As last attempt we decided to make 1/4 of guesses on the first iteration and 3/4 on the second iteration. The final result show 70.5% of password cracked (see Figure 9) which is not optimal.

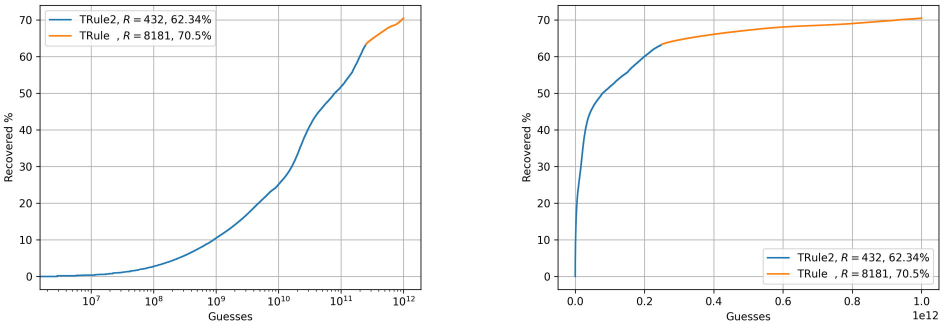

Figure 9: Test with dynamic rule using TRule2 and TRule

The statement "We could conclude that with bigger dictionaries (and smaller rule sets) the success rate would increase even more"[2] still proved true, in fact we noticed that the best results are obtained with a dictionary as large as possible, which is obtained by cracking as many passwords as possible in the first iteration.

One of the possible reasons for such poor results may be due to the use of unplanned rules. In the case of TRule the rules are taken in order of frequency from the highest to the lowest and are not necessarily optimized for the tests carried out. The TRule2 rules, on the other hand, are optimized but are not enough by themselves to be used. The best result is still 72.13% cracked passwords which is a slight increase compared to the previous work (71.11%), and has been achieved by focusing only on the optimized rules called TRule2. This result was also obtained with only two iterations dividing by the first iteration the 75% of total guesses and the second the remaining 25%, proving once again that having a high number of cracked passwords at the first iteration helps a lot in the second iteration.

Figure 10 show our best results obtained compared with those obtained in the previous work. We decided to compare the previous result 71.11% (green line),the result obtained using the algorithm and 1296 rules from TRule2 71.58% (red line) and the result obtained with the algorithm with the rules 1296+2471 from TRule2 72.13% (blue line). The results show that all our best success rates have been achieved using the TRule2 rule set.

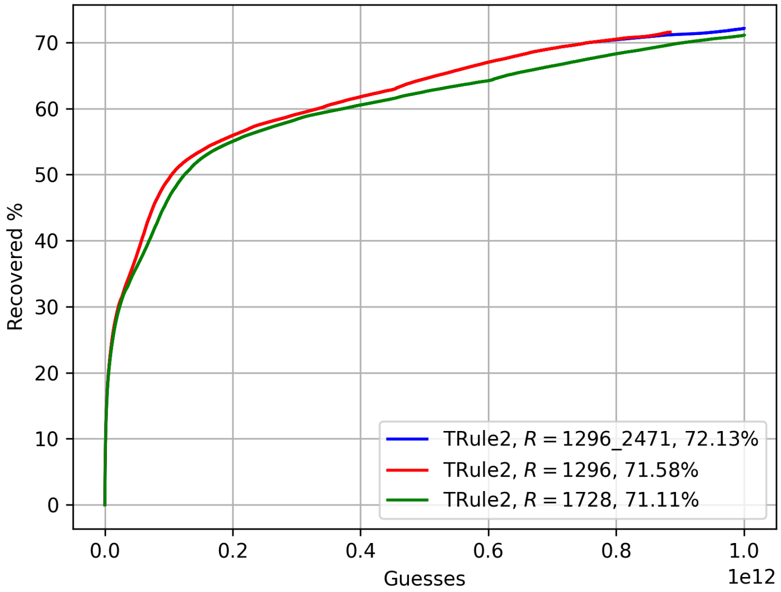

Figure 10: Comparison between previous work and our result

# Conclusion

In this thesis we studied the password cracking problem analysing whether it was possible to iteratively decrypt multiple passwords. We decided to study different approaches. The first approach allowed us to verify if our hypothesis was feasible and then the second approach helped us to check the hypotheses by setting a precise target.

We decided to run two different experiments in order to test our assumptions. The first experiment was to iterate using always the same rules for each iteration. This experiment proved to be correct by showing that with a fairly large number of rules and the complete dictionary it was possible to obtain a success rate of 71.58% (an increase of +0.47%), greater than 71.11% from the previous work, with less guesses $0.88 \times 10^{12}$ made compared to the previous work $10^{12}$ guesses [2].

The second experiment was to increase the number of rules at each iteration, in order to keep a well-defined number of guesses per iteration. This allowed us to achieve our best cracking rate of 72.13% (an increase of +1.02%) with the same number of $10^{12}$ guesses compared to the previous work [2].

These two results enabled us to achieve an encouraging result.

We came to the conclusion that the statement" bigger Dictionaries and

smaller rule sets increase the success rate" is a valid statement in the purposes of our research. Despite the results obtained in our work with an increase in cracked passwords, we did not achieve the hoped result of cracking a $10-20\%$ more password. A reason for this failure could be that the applied rules were not optimized to be used more than once.
At the end of our experiments, we showed that it is possible to achieve better results by using fewer rules and iterating, rather than using a large number of rules.

As a future work we first want to investigate if it is possible to find an iterable rule set, because we realized that the rules used (TRule and TRule2), if used more than once, lead to poor results. To achieve a better outcome it is necessary to study how the rules are composed and how they can be improved, finding new rules to add or deleting those not reusable. We also could try for each iteration to completely change the rule set to see if the results change.
Another study we could do would be to check if passwords that have not been cracked have a particular shape. An example could be to check the length of the remaining passwords. Another idea would be to check if these passwords were generated randomly and therefore they do not respect any particular characteristic and they can not be cracked.

# Bibliography

[2] Alessia Michela Di Campi, Riccardo Focardi, and Flaminia L. Luccio. "The Revenge Of Password Crackers: Automated Training Of Password Cracking Tools". In: *Computer Security – ESORICS 2022: 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26–30, 2022, Proceedings, Part II.* Copenhagen, Denmark: Springer-Verlag, 2022, pp. 317–336. URL: `https://doi.org/10.1007/978-3-031-17146-8_16`.

[3] Hans Dobbertin. "Cryptanalysis of MD4". In: *Proceedings of the Third International Workshop on Fast Software Encryption.* Berlin, Heidelberg: Springer-Verlag, 1996, pp. 53–69. ISBN: 3540608656.

[4] D. Eastlake. *US Secure Hash Algorithm 1 (SHA1).* `https://www.rfc-editor.org/rfc/rfc3174.html`. Accessed: 2-2023.

[5] Paul A. Grassi et al. *NIST Special Publication 800-63B, Digital Identity Guidelines, Authentication and Lifecycle Management.* Accessed: 20-1-2023.

[7] Saranga Komanduri. "Modeling the Adversary to Evaluate Password Strength With Limited Samples". In: (Mar. 2016). DOI: `10.1184/R1/`

6720701.v1. URL: `https://kilthub.cmu.edu/articles/thesis/Modeling_the_Adversary_to_Evaluate_Password_Strength_With_Limited_Samples/6720701`.

[8] Enze Liu et al. "Reasoning Analytically about Password-Cracking Software". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 380–397. DOI: `10.1109/SP.2019.00070`.

[10] Robert Morris and Ken Thompson. *Password Security: A Case History ACM(22)11.* `https://rist.tech.cornell.edu/6431papers/MorrisThompson1979.pdf`.

[11] Arvind Narayanan and Vitaly Shmatikov. "Fast Dictionary Attacks on Passwords Using Time-Space Tradeoff". In: *Proceedings of the 12th ACM Conference on Computer and Communications Security*. CCS '05. Alexandria, VA, USA: Association for Computing Machinery, 2005, pp. 364–372. ISBN: 1595932267. DOI: `10.1145/1102120.1102168`. URL: `https://doi.org/10.1145/1102120.1102168`.

[14] R. Rivest. *The MD4 Message Digest Algorithm.* `https://www.rfc-editor.org/pdfrfc/rfc1186.txt.pdf`. Accessed: 2-2023.

[15] R. Rivest. *The MD5 Message Digest Algorithm.* `https://www.rfc-editor.org/pdfrfc/rfc1321.txt.pdf`. Accessed: 2-2023.

[19] National Institute of Standards and Technology. *Automated Password Generator (APG) - FIPS 181.* `https://csrc.nist.gov/publications/detail/fips/181/archive/1993-10-05`. Accessed: 3-2-2023.

[21] D. R. Stinson. *Cryptography, Theory and Practice Fourth Edition.* CRC Press., 2019.

[22] *Technical Guide to Information Security Testing and Assessment.* `https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-115.pdf`. Accessed: 4-2-2023.

[23] Blase Ur et al. "Measuring Real-World Accuracies and Biases in Modeling Password Guessability". In: *Proceedings of the 24th USENIX Conference on Security Symposium.* SEC'15. Washington, D.C.: USENIX Association, 2015, pp. 463–481. ISBN: 9781931971232.

[24] Rafael Veras, Julie Thorpe, and Christopher Collins. "Visualizing Semantics in Passwords: The Role of Dates". In: *Proceedings of the Ninth International Symposium on Visualization for Cyber Security.* VizSec '12. Seattle, Washington, USA: Association for Computing Machinery, 2012, pp. 88–95. ISBN: 9781450314138. DOI: `10.1145/2379690.2379702`. URL: `https://doi.org/10.1145/2379690.2379702`.

[25] Matt Weir et al. "Password Cracking Using Probabilistic Context-Free Grammars". In: *2009 30th IEEE Symposium on Security and Privacy.* 2009, pp. 391–405. DOI: `10.1109/SP.2009.8`.

[26] Lawrie Brown. William Stallings. *Computer Security Principles and Practice (Fourth Edition).* Pearson Edu., 2018.

# Sitography

[1] Anant Shrivastava a.k.a anantshri. *One Rule to Rule Them All.* `https://notsosecure.com/one-rule-to-rule-them-all`. Accessed: 20-1-2023.

[6] Troy Adam Hunt. *Have I Been Pwned?* `https://haveibeenpwned.com/Passwords`. Accessed: 3-2-2023.

[9] Microsoft. *NT LAN Manager (NTLM) Authentication Protocol.* `https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-nlmp/a4f28e01-3df1-4fd1-80b2-df1fbc183f21`. Accessed: 2-2023.

[12] *OAuth 2.0.* `https://oauth.net/2/`. Accessed: 2-2023.

[13] rarecoil. *Pantagrule.* `https://github.com/rarecoil/pantagrule`. Accessed: 20-1-2023.

[16] *RockYou.* `https://github.com/praetorian-inc/Hob0Rules`. Accessed: 20-1-2023.

[17] *RockYou 2021.* `https://github.com/ohmybahgosh/RockYou2021.txt`. Accessed: 20-1-2023.

[18] U.S. NIST Nastion Institute Of Standard and Technology. *relying party*. `https://csrc.nist.gov/glossary/term/relying_party`.

[20] Jens Steube and Gabriele Gristina. *hashcat*. `https://hashcat.net/hashcat/`. Accessed: 20-1-2023.

# Appendix A

# Experimental Result

In this section we show for each experiments, mentioned during the chapter, the details of the results.

## Experiment 1

| Iteration | Target | Target size | Dictionary | Dict. size | Rules | Rule size | Guesses | Result |
|---|---|---|---|---|---|---|---|---|
| 1 | $Test_D$ | 144.740.239 | $Train_D$ | 579.008.917 | TRule2 | 1728 | $10^{12}$ | 71.11% |
| 2 | $Test_DNONCracked$ | 41.821.899 | $CrackedIteration1$ | 102.918.340 | TRule2 | 1728 | $1.7 \times 10^{11}$ | 72.73% |

## Experiment 2

### Fixed Rule

| Rules | N° of Rules | Guesses | Result | N° of Iterations | N° of password recovered |
|---|---|---|---|---|---|
| TRule2 | 173 | $0,1 \times 10^{12}$ | 58.19% | 6 | 84.221.544 |
| TRule2 | 512 | $3,4 \times 10^{11}$ | 66.72% | 4 | 96.573.034 |
| TRule2 | 690 | $4,6 \times 10^{11}$ | 68.66% | 4 | 99.383.441 |
| TRule2 | 1024 | $6,9 \times 10^{11}$ | 70.63% | 4 | 102.228.852 |
| TRule2 | 1296 | $8,8 \times 10^{11}$ | 71.58% | 4 | 103.609.749 |

## Dynamic Rule

### 10 Iteration

10 Iteration scaling up the rules, starting from 173 rules from TRule2.

| Iteration N° | Rules | N° of Rules | Guesses | N° of password recovered | Total Result |
|---|---|---|---|---|---|
| 1 | TRule2 | 173 | $10^{11}$ | 84.221.544 | 58.19% |
| 2 | TRule | 1129 | $10^{11}$ | 7.623.087 | 61.47% |
| 3 | TRule | 13118 | $10^{11}$ | 2.242.462 | 63.02% |
| 4 | TRule | 44594 | $10^{11}$ | 1.601.449 | 64.13% |
| 5 | TRule | 62443 | $10^{11}$ | 1.191.030 | 64.95% |
| 6 | TRule | 83960 | $10^{11}$ | 887.552 | 65.56% |
| 7 | TRule | 112669 | $10^{11}$ | 673.557 | 66.03% |
| 8 | TRule | 148466 | $10^{11}$ | 526.771 | 66.39% |
| 9 | TRule | 189836 | $10^{11}$ | 430.854 | 66.69% |
| 10 | TRule | 232097 | $10^{11}$ | 363.087 | 66.64% |

### 2 Iteration

### First Iteration 1/4 of guesses, Second Iteration 3/4 guesses

| Iteration N° | Rules | N° of Rules | Guesses | N° of password recovered | Total Result |
|---|---|---|---|---|---|
| 1 | TRule2 | 432 | $2,5 \times 10^{11}$ | 91.678.642 | 63.34% |
| 2 | TRule | 1129 | $7,5 \times 10^{11}$ | 10.368.650 | 70.5% |

### Half guesses First Iteration, Half Guesses Second Iteration

| Iteration N° | Rules | N° of Rules | Guesses | N° of password recovered | Total Result |
|---|---|---|---|---|---|
| 1 | TRule2 | 1296 | $5 \times 10^{11}$ | 98.501.818 | 68.05% |
| 2 | TRule | 5076 | $5 \times 10^{11}$ | 5.471.181 | 70.5% |

### First Iteration 3/4 guesses, Second Iteration 1/4 guesses

| Iteration N° | Rules | N° of Rules | Guesses | N° of password recovered | Total Result |
|---|---|---|---|---|---|
| 1 | TRule2 | 1296 | $7,5 \times 10^{11}$ | 101.169.048 | 69.9% |
| 2 | TRule2 | 2471 | $2,5 \times 10^{11}$ | 3.225.003 | 72.13% |

# Appendix B

# Image not used

In this section we show the unused images.

## Iteration with fixed rules

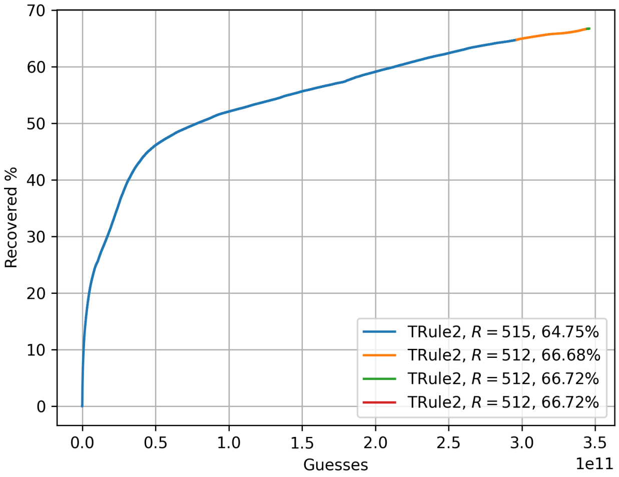

Figure 11: Success rate for fixed R=512 from TRule2 in scale linear

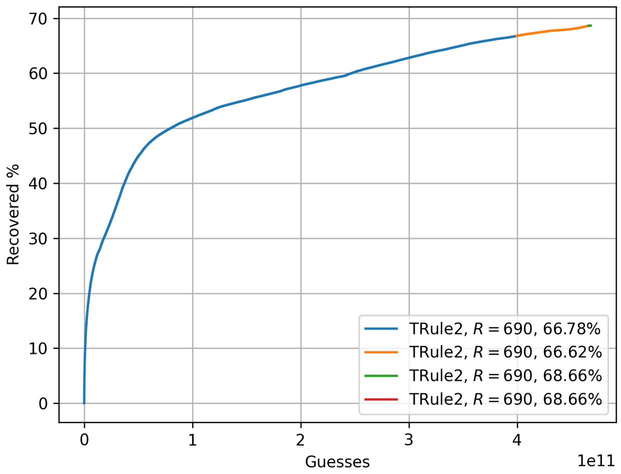

Figure 12: Success rate for fixed R=690 from TRule2 in scale linear

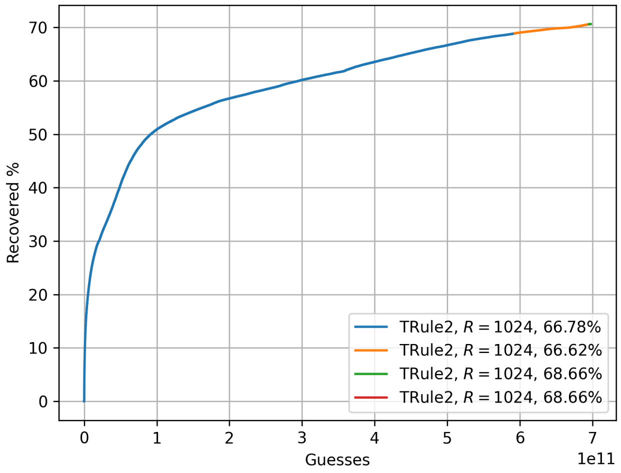

Figure 13: Success rate for fixed R=1024 from TRule2 in scale linear

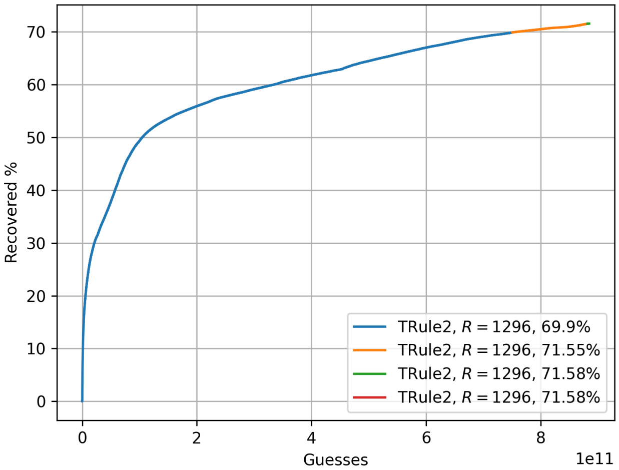

Figure 14: Success rate for fixed R=1296 from TRule2 in scale linear

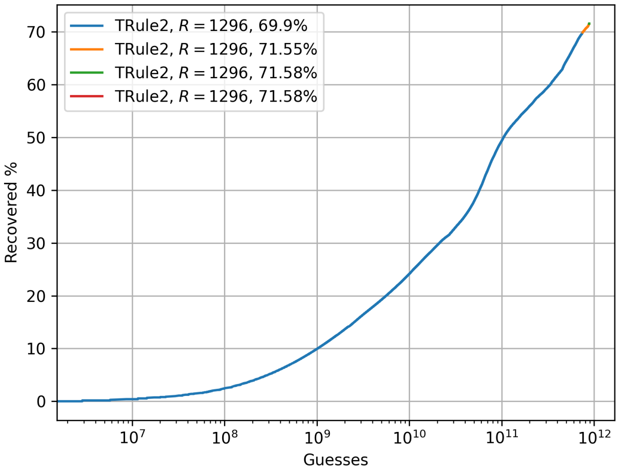

Figure 15: Success rate for fixed R=1296 from TRule2 in scale logarithmic