Filter Frenzy:

- Towards Assessing Active Directory Filters -

Masters Thesis

Group 1001
Mads Lykkeberg Møller - 20220431
Jonathan Hartvigsen Juncker - 20220325

Aalborg University
Electronics and IT

AALBORG UNIVERSITY
STUDENT REPORT

Electronics and IT
Aalborg University
http://www.aau.dk

Title:
Filter Frenzy: Towards Assessing Active Directory Filters

Theme:
Cyber Security

Project Period:
Spring Semester 2024

Project Group:
Group 1001

Participant(s):
Mads Lykkeberg Møller
Jonathan Hartvigsen Juncker

Supervisor(s):
Shreyas Srinivasa

Copies: 1

Page Numbers: 86

Date of Completion:
May 30, 2024

Abstract:

In recent years, several password filters for Microsoft's Active Directory have been developed. Historically, these filters have primarily been third-party implementations. However, Microsoft has introduced its own password filter in the form of Microsoft Entra Password Protection. Given that passwords remain the most prevalent form of access control and Microsoft's Active Directory is a cornerstone of Identity and Access Management, it is crucial to assess the effectiveness of these password filters. This raises the question of whether these filters enhance password security and how Microsoft's filter compares to third-party options.
This master's thesis presents an extensive literature review on password policies, password strength, and password guessing attacks. It also introduces a novel method for evaluating the effectiveness of password filters for Active Directory. The evaluation involves testing three different password filters using approximately 88 million passwords, state-of-the-art password strength meters, and various password guessing attacks.

AALBORG UNIVERSITET
STUDENTERRAPPORT

Elektronik og IT
Aalborg Universitet
http://www.aau.dk

Titel:
Filter Frenzy: Towards Assessing Active Directory Filters

Tema:
Cyber Security

Projektperiode:
Forårssemestret 2024

Projektgruppe:
Gruppe 1001

Deltager(e):
Mads Lykkeberg Møller
Jonathan Hartvigsen Juncker

Vejleder(e):
Shreyas Srinivasa

Oplagstal: 1

Sidetal: 86

Afleveringsdato:
30. maj 2024

Abstract:

I de senere år er der udviklet adskillige adgangskodefiltre til Microsofts Active Directory. Historisk set har disse filtre primært været tredjepartsimplementeringer. Microsoft har dog indført sit eget password-filter i form af Microsoft Entra Password Protection. Da adgangskoder fortsat er den mest udbredte form for adgangskontrol, og Microsofts Active Directory er fundamental i Identity and Access Management, er det afgørende at vurdere effektiviteten af disse adgangskodefiltre. Dette rejser spørgsmålet om, hvorvidt disse filtre forbedrer adgangskodesikkerheden, og hvordan Microsofts filter kan sammenlignes med tredjepartsmuligheder.
Dette kandidatspeciale præsenterer en omfattende litteratur søgning og review om adgangskodepolitikker, adgangskodestyrke og adgangskodegætteangreb. Den introducerer også en ny metode til at evaluere effektiviteten af adgangskodefiltre til Active Directory. Evalueringen involverer test af tre forskellige adgangskodefiltre ved hjælp af cirka 88 millioner adgangskoder, avancerede adgangskodestyrkemålere og forskellige adgangskodegætteangreb.

0.1 Reading Guide

This thesis contains a variety of technical abbreviations, the following reading guide will explain these abbreviations:

- KDC - Key Distribution Unit
- PSM - Password Strength Meter
- AD - Active Directory
- LSASS - Local Security Authority Subsystem Service
- LM - Lan Manager
- NTLM - New Technology Lan Manager
- LSA - Local Security Authority
- ASCII - American Standard Code for Information Interchange
- CFCS - Center for Cyber Sikkerhed
- NIST - National Institute of Standards and Technology
- GPU - Graphical Processing Unit
- CPU - Central Processing Unit
- PCP - Password Complexity Policy
- ADSI - Active Directory Service Interfaces
- COM - Component Object Model
- GAN - Generative Adversarial Network
- LDAP - Lightweight Directory Access Protocol
- HPC - High Performance Computing
- DLL - Dynamic Link Library
- OPF - OpenPasswordFilter

Contents

Preface

Aalborg University, May 30, 2024

[illegible]
Mads Lykkeberg Møller
mlmo22@student.aau.dk

[illegible]
Jonathan Hartvigsen Juncker
jjunck22@student.aau.dk

Chapter 1

Introduction

In the realm of enterprise network security Active Directory (AD) stands as a cornerstone for Identity and Access Management as the primary directory service for Windows domain networks. Active Directory plays a critical role in maintaining the security and integrity of organizational IT infrastructure.

Passwords remain the primary method for access control across organizations and technologies including Active Directory. Weak passwords have been a persistent source of security breaches and vulnerabilities for decades [40]. To enhance password security, various advancements has been made, one of the more prominent being Password Policies (PPs). These policies set specific requirements that passwords must adhere to, to be considered valid. Active Directory can also employ password policies in the form of password filters [35]. These filters have historically been third-party implementations, however to the best of the our knowledge Microsoft released the beginnings of what would become their own filter in 2018 [33], focused on blocking weak passwords from Active Directory. Today, both third-party implementations and Microsoft's Password Protection rely on blocklists to prevent weak passwords from being created and used, however research has shown that implementing blocklists while recommended [42][7] is notoriously hard to do accurately [55] [26]. To the best of our knowledge no research has been done into the effectiveness of password filters for Active Directory and the extent of the potential password security they provide.

This brings into question of what makes a weak or strong password, which has been a research topic for many years. Historically, password strength has been measured through information theoretic entropy [52], and has therefore resulted in password policies requiring using upper-, lower-case characters and special characters. This has been proven to be an insufficient measurement of password strength [63]. Modern standards regarding passwords also recommend minimal restrictions on password complexity [42] [7]. Most enterprise services now employ Password Strength Meters (PSMs) that inform users about the calculated strength of their password when creating it, and even forbids a low strength password from being created. There exists a plethora of different PSMs that can measure password strength in various ways such as guessability, password probability, and resistance to guessing attacks. Extensive research has been done to develop accurate and effective PSMs [63] [61] [15] [9]. We believe that these PSMs along with password guessing attacks can be leveraged to measure the effectiveness and strength of an Active Directory password filter.

1.0.1 Motivation

Considering the gap in research into Active Directory password filters it is the goal of this thesis to research:

- How does Microsoft's Password Protection solution compare to various open-source Active Directory password filters.
- Are Active Directory password filters effective in increasing password security.
- Develop and test a method to evaluate Active Directory password filters.

1.0.2 Contribution

This masters thesis consisted of researching Microsoft Active Directory password filters and developing a method for evaluating them. This thesis brings the following main contributions:

1. We provide an extensive literature search and review of contemporary research into password policies, password strength and password guessing attacks
2. We propose, to the best of our knowledge, a novel method for testing Active Directory password filters, leveraging four different password strength meters, several password guessing attacks, and roughly 88 million passwords.
3. We systematically test our proposed method on three different password filters for Microsoft Active Directory, and provide an evaluation of the effectiveness of the tested Active Directory password filters.

Chapter 2

Background

2.1 Password History

Password security is not a new research area, Morris et al. [40] explored password vulnerabilities in 1979. Furthermore, passwords have been used for authentication since the early days of computing, such as MIT's time-sharing system in the early 1960s [4].

While password security has witnessed many vulnerabilities and changes through the times, the security that passwords provide is ultimately dependent on the implemented system in question. As to improve these systems, various password policies have been proposed through the times [4].

2.2 Directory Services

A Directory, is a hierarchical structure used to store data on a network. The type of data stored is varying depending on the directory, but commonly information about users, computers, servers, etc. is stored in a directory [30].

Directory Services has evolved over the time, but the primary reason it became a popular solution is the same reason for it being used so widely today. That reason is that it provides a central solution to meet various business needs, such as support for network and enterprise management, security, messaging, employee and client identity management [53]. Directory Services can also provide supportive services to an enterprise, with the most fundamental service being storing and retrieving information from the Directory. Because of this service a centralized repository, efficient data management, security and scalability is possible [53].

2.2.1 Active Directory

Active Directory (AD) is a database and set of services that falls under the category of Directory Services. Active Directory uses a controller called the Domain Controller, which is used to control the services and directory itself. A Domain Controller is a Windows server and has the Active Directory Domain Services installed as a role [32].

The Microsoft Active Directory Domain Services (AD DS), provides methods for interacting with a Directory. These methods are used for storing and sharing data with users on a network. In Active Directory, user information such as names, passwords, usernames are stored in the AD DS, and other authorized users on the same network can through the AD DS access this information [30].

Authentication in Active Directory, is done through Windows Single Sign-on (SSO) architecture. This means that users on the network, simply log on once to access resources in the Directory. Authentication in Active Directory has previously used the LM and NTLM (New-Technology LAN Manager) protocols, but today follow the Kerberos protocol. The NTLM protocol is still employed by various systems, particularly legacy services that have not been updated [34]. One significant vulnerability of NTLM authentication is the PassTheHash attack [6]. This attack is possible due to how NTLM authentication involves the transmission of a password hash. Since this hash is used for authentication, it practically functions as the password.

For the Keberos protocol, instead of passing the credentials across the network, a session token is created. This session token contains what access rights the user have [49]. The Kerberos authentication in Active Directory, is done by converting the entered password into a hash and storing it in memory, which is all done by the Local Security Authority Subsystem Service (LSASS) process. The hash is then used against the Key Distribution Center (KDC), typically the Domain Controller [39].

Due to backwards compatability, the Keberos protocol can create RC4-HMAC-MD5-encrypted Kerberos tokens based on NTML hashes. However, this opens up the authentication to an attack known as OverpassTheHash [1], which operates under a similar principle to PassTheHash but instead facilitates the acquisition of a Kerberos ticket.

In Active Directory a password policy that enforces certain password requirements can be enabled. These requirements are enforced when a password is created or changed. The different requirements that can be enforced are [37] [38]:

- Maximum password age.
- Minimum password age.
- Minimum password length.
- Password must meet complexity requirements:
  - Passwords must not contain the user's entire samAccountName (Account Name) value or entire displayName (Full Name) value. Both checks are not case sensitive
  - Passwords must contain characters from three of the following five categories:
    - Uppercase characters of European languages (A through Z, with diacritic marks, Greek and Cyrillic characters).
    - Lowercase characters of European languages (a through z, sharp-s, with diacritic marks, Greek and Cyrillic characters).

    * Base 10 digits (0 through 9).
    * Nonalphanumeric characters: ~ ! @ # $ % ^ & * _ - + = `\ | ( ) { } [ ] : ; ” ‘’ < > , . ? /
    * Any Unicode character that is categorized as an alphabetic character but is not uppercase or lowercase. This includes Unicode characters from Asian languages.

2.2.2 Active Directory Password Filters

In Active Directory it is possible to enable custom password filters, that has custom requirements for password creation and password changes. When a password change request is made, the Local Security Authority (LSA), will call all password filters on the system, to validate the new password. It is possible to have multiple password filters enabled, such as both the standard password requirements in Active Directory, and custom password filters. [36]

These custom filters can be created to enforce much more strict and many more requirements for password creation. There exists many open-source and commercial password filters, that can be tweaked to fit all needs. Such tools include, but is not limited to:

- Lithnet: Password Protection for Active Directory [28]
- ImprosecPasswordFilter [22]
- safepass.me [46]
- Specops Password Policy [47]
- Enzoic for Active Directory [14]
- OpenPasswordFilter [54]
- PassFiltEx [45]
- Microsoft Entra Password Protection [33]

2.3 Authentication

Authentication is the process of verifying the identity of a user or system when accessing a resource or service. The methods which the user uses to substantiate its authenticity can be divided into different categories [59]:

- The user/system knows something: Passwords, PINs, shared secrets.
- The user/system owns something: Cards, certificates.
- The user/system has some inviolable characteristics: Biometric data.

The widely used method of authentication is passwords along with some multi factor-authentication [59].

2.4 Password Entropy

Passwords are still the most common method of authentication. Generally humans are bad at creating and memorizing complex arbitrary passwords, resulting in passwords that are easily guessed. To force users to create more complex passwords, organizations enforce password policies, that often state that passwords require upper and lower-case letters, numbers, symbols, and must be at least 8-characters long. The introduction of these requirements aim at increasing the entropy of the password.

Information theoretic entropy [52] has often been used to quantify the complexity of passwords. When researching ways to calculate password entropy a simple formula appears on several password entropy calculators:

$$E = log_2(R^L)$$

Where E is the entropy in bits, R is the possible characters within the password, meaning if the password is *"password"*, R is 26 considering the standard English alphabet, and L is the length of the password. This results in an entropy of $log_2(26^8) = 37.6$.

Upon further research, the simple formula can only correctly calculate entropy if the password is completely randomly generated, which most passwords are not [43].

Considering new guidelines for password safety from Center for Cybersikkehed (CFCS) [7], and NIST [42], the most important part of a passwords security is the length of the password, determining that the length should be at least 15 characters long. This is mostly to protect against bruteforce attacks. Both organizations state that most passwords are vulnerable to phishing and general social engineering attacks.

The guidelines also state that the usual requirements for passwords that companies or organizations tend to use to increase password complexity would still allow for a password like *Password123456!*, and doing the entropy calculation states that the password has an entropy of 98 bits, which is seen as very strong. But looking up the password on `https://haveibeenpwned.com/Passwords` [21] we see that the password has been seen in 120 different data breaches, making it a poor choice of password.

2.5 Leaked Credentials

Account credentials, is a information tuple that includes username and password combinations. These credentials are used for authenticating to gain access to a service. If credentials gets stolen or leaked, then any adversaries can potentially obtain the credentials and use them to gain unauthorized access to the relevant services.

If a user decides to re-use a password for different services, then the number of services that can be accessed by adversaries if they obtain the password also increases. This poses a security risk, due to the nature of users often re-using passwords [29].

The challenge of identifying whether your passwords have been compromised is significant, due to massive scale of credentials getting leaked or stolen, and due to the secret nature of leaked credentials. There is no guarantee that a user will realise their credentials are leaked. Therefore, various services have been created to help users realise if their credentials are found leaked online. One of the leading platforms in this field, HaveIBeenPwned [21], now encompasses over 12 billion compromised credentials, showcasing the vast quantity of data accessible to potential attackers. These leaked credentials are often used in credential stuffing attacks, leading to a common security practice to block the use of previously leaked passwords.

2.6 Password Guessing Attacks

Password guessing attacks is a method used by attackers to gain unauthorized access to systems, user accounts, or sensitive information. These types of attacks usually rely on the fact that the passwords are either easy to guess or have been leaked in a data breach. There are two different strategies for password guessing attacks: Offline password guessing attacks and online password guessing attacks. [62]

2.6.1 Offline Password Guessing Attacks

In offline password guessing attacks, it is not difficult for attackers to utilize GPUs to perform billions of guesses [62]. Offline guessing can be generally categorized into three different groups: brute force, dictionary, and probability-based attacks. Brute force attacks consist of attackers performing an exhaustive search for a password over a given search space. This is inefficient, computationally expensive, and time-consuming. Dictionary attacks involve guessing a password based on wordlists and frequency lists, usually also with a ruleset for permutations. A probability attack relies on targeting the password distribution using parametric probability models such as the Markov model. Generally, in offline password guessing attacks, the attacker will utilize all three methods in order to successfully guess a password. Tools such as John the Ripper[1] and Hashcat[2] are powerful free tools for the purposes of offline password guessing, and supports the three different strategies. [62]

2.6.2 Online Password Guessing Attacks

In contrast to offline password guessing attacks, the online strategy, is limited by the amount of guesses an attacker is allowed to make. This means that an attacker cannot rely on brute force attacks, as those require a large amount of guesses to be successful. Instead the attacker will use probability based attacks like guessing with the most commonly used passwords. For this reason online password guessing attacks can be categorized by the type of attacker either a knowledgeable attacker or a general attacker. The difference here being the that the knowledgeable attacker has some knowledge about the password distribution and can make more calculated guesses, where as the general attacker utilizes generally common passwords usually from a leaked corpus of passwords. [62]

[1]https://github.com/openwall/john
[2]https://github.com/hashcat/hashcat

Chapter 3

Related Work

This chapter contains a systematic literature search and review of related works to password policies, password strength, and password guessing attacks. The purpose is to find ways of evaluating password policies that can be used to develop a method for evaluating Active Directory password filters.

3.1 Systematic Literature Search

The goal of this section is to find research that explores password policies, password strength, and password guessing attacks. For the purposes of searching and finding relevant research into the described areas. Papers found during the initial conception of the project was used as a start set for the snowballing method [64]. These papers can be seen in Table 3.1.

Authors	**Title**	**Year of publication**
Tan et al. [55]	Practical recommendations for stronger, more usable passwords combining minimum-strength, minimum-length, and blocklist requirements	2020
Nisenoff et al. [41]	A Two-Decade Retrospective Analysis of a University's Vulnerability to Attacks Exploiting Reused Passwords	2023
Golla et al. [18]	On the accuracy of password strength meters	2018
Bohuk et al. [3]	Gossamer: Securely Measuring Password-based Logins	2022

Table 3.1: Start set for snowballing.

Using the start set, two rounds of snowballing was performed. These rounds include both forward and backward snowballing. An overview of the snowballing process can be seen in Figure 3.1. A set of requirements for the research papers to be included in this project was made in order to filter the papers found. These requirements were:

- Paper must be in English.
- Paper must be no older than 6 years.
- Paper must pertain to either password policies, password strength, and password guessing attacks.

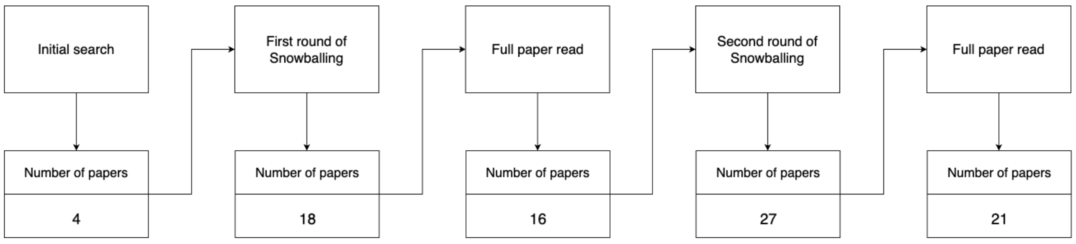


Figure 3.1: Overview of the literature search process.

The final list of papers found from the snowballing process, and to be reviewed for this project can seen in Table 3.2. Some papers where excluded from the filter, due to their relevancy for the project.

Authors	Title	Year of publication	Topic
Tan et al. [55]	Practical recommendations for stronger, more usable passwords combining minimum-strength, minimum-length, and blocklist requirements	2020	Password Strength
Habib et al. [19]	Password creation in the presence of blacklists	2017	Password Strength, Password Policy
Dong et al. [11]	RLS-PSM: a robust and accurate password strength meter based on reuse, Leet and separation	2021	Password Strength
Galbally et al. [15]	A new multimodal approach for password strength estimation—Part I: Theory and algorithms	2016	Password Strength
Galbally et al. [16]	A new multimodal approach for password strength estimation—Part II: Experimental evaluation	2017	Password Strength
Ji et al. [23]	Zero-sum password cracking game: a large-scale empirical study on the crackability, correlation, and security of passwords	2015	
Eckroth et al. [13]	OneRuleToFindThem: Efficient Automated Generation of Password Cracking Rules	2023	Password Attacks
Bohuk et al. [3]	Gossamer: Securely Measuring Password-based Logins	2022	Password Strength, Password Analysis
Golla et al. [18]	On the accuracy of password strength meters	2018	Password Strength
Nisenoff et al. [41]	A Two-Decade Retrospective Analysis of a University's Vulnerability to Attacks Exploiting Reused Passwords	2023	Password Strength, Password Policy
Gerlitz et al. [17]	Please do not use!? _ or your license plate number: analyzing password policies in german companies	2021	Password Policy
Hitaj et al. [20]	Passgan: A deep learning approach for password guessing	2019	Password Strength, Password Attacks
Johnson et al. [24]	Skeptic: Automatic, justified and privacy-preserving password composition policy selection	2020	Password Policy
Segreti et al. [50]	Diversify to survive: Making passwords stronger with adaptive policies	2017	Password Strength, Password Policy
Pereira et al. [44]	Evaluating the accuracy of password strength meters using off-the-shelf guessing attacks	2020	Password Strength, Password Attacks
Wang et al. [62]	No single silver bullet: Measuring the accuracy of password strength meters	2023	Password Strength, Password Analysis
Lee et al. [26]	Password policies of most top websites fail to follow best practices	2022	Password Policy
Lim et al. [27]	Evaluating Password Composition Policy and Password Meters of Popular Websites	2023	Password Policy
Wheeler [63]	zxcvbn:Low-Budget Password Strength Estimation	2016	Password Strength
Seitz et al. [51]	Do differences in password policies prevent password reuse?	2017	Password Policy
Dell'Amico et al. [9]	Monte Carlo strength evaluation: Fast and reliable password checking	2015	Password Strength

Table 3.2: Final set of papers for review.

3.2 Related Works Review

3.2.1 Password Strength

Galbally et al. [15] [16] proposes a new multimodal Password Strength Meter (PSM), that uses multiple types of models. The authors state that password strength is dependent on the attack strategy used against the password. And that there is no objective truth when it comes to password strength, and between 2 different strength scores deciding which is correct, is objectively hard. Their proposed method is comprised of both statistics and heuristics, fused from four individual heterogeneous strength scores, Attack-Based, Heuristic-Based, Adaptive Memory Markov Chain, and Hierarchical Markov Chain. The results of comparing their strength meter shows it outperforming strength meters from NIST-PSM, Yahoo-PSM, Google-PSM, and Dropbox-PSM.

Wheeler [63] showcase their PSM *zxcvbn* that estimates strength by how common a password is, based on various data sources, such as frequency lists, common names, and leaked passwords. Secondly, the strength estimator uses search heuristics for multipattern sequences. They compare the method to other estimators such as NIST and KeePass, aswell as the Password Guessability Service from Ur et al. [58]. They find that *zxcvbn* if trained on the same or similar data as attackers use, is able to predict todays best online guessing attacks.

Golla et al. [18] explores the accuracy of PSMs by proposing a set of properties that a PSM needs to have to maintain high accuracy. These properties are, *Tolerance to Monotonic Transformations*, *Tolerance to Quantization*, *Tolerance to Noise*, *Sensitivity to Large Errors*, and *Approximation Precision*. The authors performs a variety of tests on various PSMs, where an ideal reference is used as the comparison for the test cases. This means that they can test their properties and compare the results to their ideal reference. They find that using a weighted Spearman correlation for the comparisons gives the most accurate results, and also find that academic PSMs based on Markov models, PCFGs, and RNNs performs the best.

Bohuk et al. [3] creates a framework for recording statistics about login attempts. They use this framework to analyze the strength of passwords used at 2 unversities. Their method of measuring strength of passwords, is by calculating a strength score with the *zxcvbn* PSM [63], if the password appears in the top 5 thousand RockYou leaked passwords, if the password appears in the top 5 thousand generated passwords from Hashcat on RockYou with the best64 rule set, or if the password appears in their top 1000 common password leaked list.

Wang et al. [62] investigates the potential of having a single metric on which to evaluate the accuracy of PSMs. They find that no such metric exists. They develop a framework on which to measure PSM accuracy in various guessing scenarios and strategies. They do find academic PSMs are more accurate than commercial ones. Wang et al. include Chinese datasets since language is an important factor when it comes to measuring password strength. Wang et al. measures accuracy in two different scenarios, offline guessing attacks and online guessing attacks. They do so because under these two different scenarios, the factors determining whether a password is strong differ. In online guessing attacks the attacker is limited in the amount of guesses they can make, therefore they will try the most commonly used passwords. This means that what makes a password secure in this scenario is the frequency of the password. Wang et al. finds that for the online guessing scenario the fuzzyPSM[61] performs the best.

In the offline guessing attack scenario an attacker is not limited by the amount of guesses they can make and will use different strategies when trying to guess a password. In this scenario it is the guessability of the password that determines if the password is string. Wang et al. finds that for the offline guessing attack scenario the MultiPSM [15] performs the best.

Pereira et al. [44] evaluates the accuracy of 13 different PSMs, using off the shelf guessing attacks. The tools used to test the PSMs were Hashcat, JohnTheRipper, and the Password Guessability Service from Ur et al. [58]. The rules used for Hashcat and JohnTheRipper, was a rule set created by combining popular rule sets together. The method used, sampled $60,000$ passwords from leaked lists and queried the PSMs with these passwords. Their results show that passwords resistance to off the shelf guessing tools, relate to the passwords strength indicated by the PSMs tested. The study also shows that some passwords scored as stronger could still be guessed, indicating that the PSMs could be improved.

Dong et al. [11] propose a PSM called RLS-PSM based on Reuse, Leet and Separation. The PSM is seperated into three different modules a preprocesssing module, probability calculation module, and a strength feedback module. The preprocessing module calculates the use of special characters impact on password stength. They separate the password into substrings based on special characters, uppercase letters or leet transformations as separators. They use a dictionary to query the substrings and their corresponding probability. The probability calculation module calculates the probability of the basic string output from the preprocessing module, and calculates the probability of the overall structure of the password. The strength feedback module then communicates the strength of the password through an absolute strength which reflects the actual strength of the password. But also through relative strength by comparing it to a benchmark password. The proposed PSM outperforms many mainstream PSMs only slightly falling behind fuzzyPSM [61] in an offline guessing scenario.

Dell'Amico et al. [9] proposes a novel method based on Monte Carlo sampling to estimate the number of guesses an attacker would need to guess a password, which in turn can be used as a strength metric for passwords. The model uses Monte Carlo sampling on 3 probabilistic attack models, *n-grams*, *PCFG*, *backoff*. The authors prove that the Monte Carlo sampling on these 3 attack models can approximate the number of passwords more probable than a given password, thereby approximating the strength of a given password.

3.2.2 Password Policies

Lim et al. [27] and Lee et al. [26] provide general research into password policies on websites and seems to reach the same conclusion: a majority of online services still use outdated versions of password policies, that no longer adhere to the guidelines presented by NIST. Relying instead on old composition-based rules such as requiring special characters or capitalized letters.

Lee et al. [26] finds that more than half of the websites they investigate do not check passwords against the 40 most common passwords. Furthermore, they discover that only 23 of the investigated websites employ PSMs, and of those 23, 10 are inaccurate. The authors describes best practices for creating a password policy that encourages users to create more secure passwords, Lee et al. suggests using blocklists to prevent users from creating passwords common passwords. They note, however, that the blocklists should be configured carefully to avoid causing user frustration. Additionally, Lee et al. mentions the use of compromised credential checking, which involves verifying if a username-password combination has been exposed. Although this practice raises ethical concerns that inhibit further investigation.

Lee et al. [26] also asserts that minimum strength requirements and PSMs are effective and user-friendly. While measuring password strength poses challenges, it is a worthwhile pursuit as it encourages users to create stronger passwords. Finally, Lee et al. argues against the use of character-class Password Composition Policies (PCPs), which mandate the inclusion of special characters, numbers, and capitalized letters in passwords. According to them, this requirement is a relic of the era when password strength was measured solely based on entropy and serves only to decrease password memorability without significantly enhancing security.

Habib et al. [19] found that using a password blocklists in a password creation process, led to increased strength of passwords created. However, they also found that 51.4% of users who attempted to create a blocked password reused the blocked password as a part of their final password (e.g. "happyday" -> "happyday!"). Furhtermore, 18% reused the blocked password in a modified form as their final password (e.g. "stewart7" -> "s1t9e9w8art").

Gerlitz et al. [17] conducted a study on 83 German companies and their password composition policies (PCP). In their study, they compare the extracted PCPs to recommendations from industry frameworks NIST, BSI, and recommendations from researchers. Their study found that the PCPs examined have high heterogeneity and find it likely that this is due to clashing of guidelines from BSI and NIST, as well as the vague nature of BSIs recommendations. The password policies analysed in the study, are found to have minimum length requirements between 8-12 characters, with 8 being the most predominant requirement. Furthermore, requiring different character classes for complexity is also found to be either 3 or 4 classes required, with 3 classes being the most predominant. Finally, 50% of the companies where found to use some blocklist.

Research in expanding the responsibilities of password policies has been done. Seitz et al. [51] investigates if password policies prevent password reuse across websites. They find that this is not the case as a single password can be created that satisfies 99% of all policies, across the top 100 most used websites in Germany. They propose a dynamic policy that potentially could detect a likely reused password and encourage the user to change or alter their current password, to prevent the potential harms that comes from password reuse.

Johnson et al. [24] presents an evaluation of different PCP rules, using their developed software tool-chain SKEPTIC. To simulate user behaviour in face of different PCP rules they use leaked passwords lists to derive password probability distributions. By redistributing those probabilities they can simulate password re-selection behaviour that users exhibit. Johnson et al. ranks the different PCPs based on the values given by SKEPTIC, with PCPs that require longer passwords of 16-20 characters performing the best.

Tan et al. [55] conducts experiments to find practical recommendations for password policies. Their experiments tests the impact of differing combinations of composition requirements, blocklist requirements and minimum strength requirements. Their results showcases that character class requirements does not provide any substantial impact against password guess attacks. Their experiments also shows that the strength increase from increased length requirements is much higher than character class requirements. Regarding blocklists, the experiments shows that blocklists can improve the strength of passwords, but it is depending on the configuration of the blocklist. It also shows that a extensitve blocklist can impact the usability. They also find that fuzzy matching improves the strength more than full-string matching. Finally, they find that minimum strength requirements can help users create stronger passwords. The authors recommend minimum strength requirements and minimum length requirements, but without character class or blocklist requirements. The reasoning being that both minimum strength and blocklists make passwords stronger, however minimum strength is potentially easier to deploy, and password strength requirements does not have a noticeable negative impact on usability.

Segreti et al. [50] measure the impact on usability and security when adopting adaptive password policies. They find that with a structure based adaptive policy, meaning a policy that does not allow for passwords to have the same structure of uppercase, lowercase, numbers, and special characters, the quality of passwords increases significantly becoming more secure and with a minimal effect on usability.

3.2.2.1 Standards and Guidelines

In regards to industry frameworks and guidelines for password policies, in particular NIST [42] and the Danish CFCS [7]. We see that the recommendations for password security are mostly the same, and align with the research done in the area. Both NIST and CFCS recommend the use of blocklists, but there is a significant difference in the required length of the passswords recommended, where NIST only requires 8 character long password and CFCS requires 15 character long passwords. This is significant considering that the research shows that length is a determining factor in password strength.

3.2.3 Password Guessing Attacks

For the purpose of measuring and testing password strength, it is clear that password guessing attacks can and is used as a metric for testing password strength. Ji et al. [23] conduct a empirical study on the effectiveness of 6 state-of-the-art password guessing techniques testing on roughly 145 million passwords. The study conducts three tests: *training free*, where the guessing methods tested don't need any pre-training, *intra-site training* is tested where the guessing methods are trained on parts of a dataset, and testing is done on the remaining part of a dataset, *cross-site training* is where different datasets are used for training and testing. The different tests performed shows that for training free, JohnTheRipper-Bleeding-Jumbo Incremetal performs the best with a roughly 15% to 55% success rate depending on the dataset. The authors find that for both intra-site and cross-site training, no single guessing method is found to be the best overall and conclude that a hybrid strategy potentially could be promising.

Eckroth et al. [13] develops a new algorithm that can find successful rules for Hashcat password guessing. These rules are used to specify how password guess transformations should be done when password guessing. Their algorithm uses combinatorial generation of rules and emperical observations of these rules. Their algorithm is optimized as to avoid naïve brute-force technique pitfalls. They compare their algorithm to state of the art rule-sets such as Pantagrule, PACK, and dive. They find that a combination of the best performing rule set Pantagrule and their own improves the amount of passwords guessed by 1%.

Hitaj et al. [20] expands upon traditional rule based password guessing methods, and develops a deep learning Generative Adversarial Network (GAN), that learns from real password data, as to automatically generate password guesses. The model named PassGAN is compared to traditional guessing methods and a different neural network based password guesser. The results show that PassGAN was always able to produce the same amount of matches as other tools. However, it needs to produce a larger number of guesses than the other tools. Finally PassGAN was also found to be able to generate passwords that looks like human created passwords.

Nisenoff et al. [41] performs a two decade retrospective password analysis of a university. This is possible since the university has kept a list of all passwords used in the past. Their analysis encompasses both how strong the passwords of the university are and have been. The authors use 4 methods of password-tweaking for their password guessing, and they search for passwords in over 450 individual service breaches and 12 large breach compilations. The results show that 71 of the individual breaches and all compilations they tested resulted in at least 1 correct guess. In total the authors managed to guess $14,161$ passwords contained in the universities password history database. The results show that reused passwords are a far greater vulnerability than common passwords. Notably the authors find that password length requirements can have temporary protective effects against password reuse attacks.

3.3 Literature Review Summary

From the literature review, we have found different methods for calculating password strength. A common way to measure password strength is done with PSMs, that automatically determines the strength of a given password. From the literature review, we have found various studies of PSMs, such as Galbally et al., [15], [16], and Wheeler [63]. Additionally, our literature review revealed that PSMs can exhibit vastly different strengths. Some PSMs are more accurate for offline guessing attacks, while others perform better for online guessing attacks [62]. We also find that much of the research uses Ur et al. [58] Password Guessability Service to measure password strength.

The literature shows that in regards to password policies, blocklists are an effective measure in enhancing password security, though it may come at a cost to usability and is difficult to configure correctly [26] [19] [55]. It also showed that PCPs is an inefficient way of increasing security and harms usability [26].

The literature review also showed how different password guessing attacks can be used in various ways. Wang et al. [62] used both online and offline guessing attack scenarios to test various PSMs. Ji et al. [23] showcased the different strengths of various password guessing tools, and used password lists to do so, showcasing how the use of password lists for strength estimation. Additionally our literature review highlighted how settings for password guessing tools, such as mangling rules can have a significant impact on the success rate of the attacks [13].

The literature review also demonstrated the systematic use of password lists for strength estimation. Nisenoff et al. [41] illustrated how various password lists could be utilized to evaluate the strength of their university's passwords. The review also highlighted that much of the research relies on commonly used password lists, such as the LinkedIn leak and the RockYou leak lists. However, not a lot of the research uses a mix of password origins, Wang et al. [62] does this by using password lists of both English and Chinese origins. To expand upon just using public known password lists the authors Hitaj et al. [20] developed a Generative Adversarial Network that is capable of generating passwords.

Chapter 4

Methodology

In this chapter we will explain our approach for testing and evaluating the strength of Active Directory password filters. Our overall method can be described in the following way:
1: Gather a number of differing lists of passwords, that each bring their own aspect, such as an often academically used list, a list of common passwords, differing languages, and collections often used by hackers.
2: Test if all of these passwords are allowed by a given Active Directory password filter, whilst saving each password that was either accepted or rejected.
3: Test the strength of the accepted passwords of each filter, to determine the strength of the filter itself. Our method for testing the strength of passwords is based on previous research in this area, using methods such as password guessing attacks and password strength meters.

In Section 4.2 we explain our Active Directory experimental setup. We then in Section 4.3 go through our process for gathering password lists for our testing. The Active Directory password filters chosen for our thesis is explained in Section 4.4, and in Section 4.5 we outline the scripts developed and used to test if the passwords we have gathered are allowed by these filters. Then in Section 4.6 we showcase our chosen methods for measuring password strength, which is then used for the final score calculations described in Section 4.7.
A summary of the chapter can be found in Section 4.8.

4.1 Ethical Concerns

The ethical concerns surrounding this thesis revolves around the use of leaked passwords from various services. Though these passwords are publicly available and have been used in other research, none of the leaked password lists will be distributed by the authors, and any information and data gathered from them will only be presented in an aggregated form as to prevent exposure of any personable identifiable information.

Furthermore, any passwords that we have gathered for our experiments will be transferred between our devices in a secure encrypted manner. Finally, once our thesis has finished, we will purge all unnecessary password data from devices involved.

4.2 Experimental Setup

The setup used for our experiments involved a straightforward configuration using a VirtualBox virtual machine running the latest Windows Server 2022 build. We established a new Active Directory Domain on this server, and set the server as the Domain Controller. On this Domain Controller, we then created user objects, that could be used to simulate users "changing" their password. An overview of the virtual machine setup can be seen in Figure 4.1.

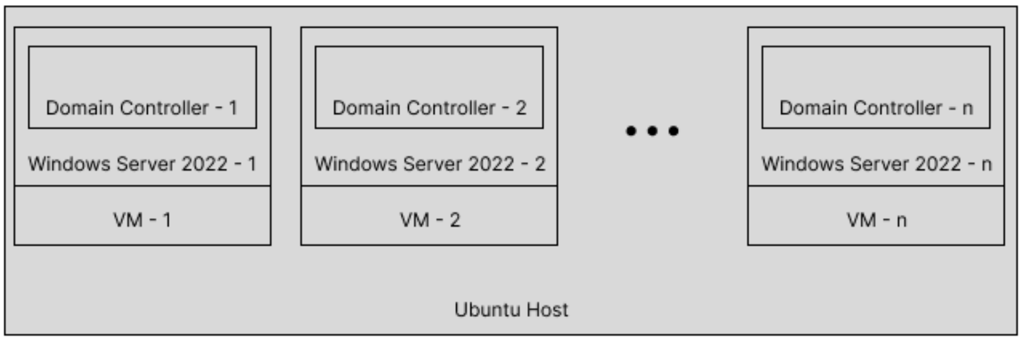


Figure 4.1: Overview of the virtual machine testing setup

4.3 Password Lists

For our testing we are using a collection of different password lists, that we have gathered for this specific purpose. Following the works of others such as Nisenoff et al. [41], we scoured various sources to find different password lists. The lists chosen come from sources such as website breaches, database leaks, breach compilations, word frequency lists, and public password cracking lists. Furthermore, we obtained password data from research contacts at our University. In our search for password lists, we did not sign up for any private leak forums, pay for any data, distribute or redistribute any data. An overview of the passwords lists can be seen in Table 4.1.

Name	Type	Total Passwords
SecLists Keyboard-Combinations	Keyboard Walk	9604
SecLists 10-million-password-list-top-1000000	Common Passwords	999,998
SecLists default-passwords	Common Passwords	1315
Xato-Net	Common password compilation	5,189,454
PassGAN	Generated passwords	10,000,000
Russian	Wordlist	2,500,000
Chinese	Wordlist	2,500,000
Greatest_books_of_all_time_original	Wordlist	5568
RockYou	Database leak	14,344,391
LinkedIn	Database leak	52,789,651
DanishTop5K	Wikipedia Frequency List	5000
EnglishTop1K	Wikipedia Frequency List	1000
Total		88,345,981

Table 4.1: Passwords Lists

For our data we have specifically chosen a list of Russian and Chinese words, to see how other languages impact the filters. However, these lists are not passwords in the same sense as some of the other lists.

4.3.1 Generating Passwords Using PassGAN

PassGAN developed by Hitaj et al. [20], is originally intended for performing password guessing attacks, by generating passwords based off of leaked password lists. But PassGAN offers a unique way to introduce new unseen passwords for the purposes of testing the Active Directory password filters. This is especially important as filters like Lithnet can use blocklists from services like HaveIBeenPwned [21], which likely will block the passwords from any leaked list of passwords we have chosen.

The PassGAN implemention used for this project [2], comes with a pretrained model, that has been trained on the RockYou dataset. The problem with using this model is the limitation set on the character length of the passwords that the model generates. The model only generates passwords with a length of up to 10 characters which eliminates the possibility of using PassGAN for testing longer passwords on the password filters. To address this, a new model was trained this time using the significantly larger list of leaked passwords, namely the LinkedIn password list mentioned in Table 4.1. Following the recommendations of the PassGAN authors Hitaj et al. [20], the model was trained over several days to reach the 200,000 recommended iterations with a new password character limit of 20 characters. A sample of the resulting passwords can be seen in Table 4.2

ms07111587	pacheans23	ba2wev96
55721350	stavardade	engxerpson
000818mi	jngioleedang	habriancy
tetruwd133	kieshorolu	107413edints
frewch07	dica2007	sumitaarainis
carvarez	5ehthj	99469214
miokye	tearradinuo2	010itav
712eflshtac	95phoves	aeenddadlosau
2478690	fshows0	2936tmma
dardaso	kasa2266	bmbice6
ctoobt123	estin3593	elsin286
cpk22000mjar	ealchmbicp1$1	aole01571

Table 4.2: Samples of the generated passwords using PassGAN

4.4 Password Filters to be Evaluated

Our proposed methods effectiveness will be assessed by testing and evaluating three different password filters for Active Directory, with the method. The password filters to be tested, are chosen from various criteria: Availability, cost, ease of use, and provider. While there exists different open source filters, we have chosen ones that supports different kinds of password filtering or different default blocklists. Furthermore, our method will be used to evaluate how effective Microsoft's own password filter is compared to the competing password filters on the market.

Microsoft Password Protection [33]. Microsoft have a password filter for Active Directory as a part of their Entra suite. It builds upon telemetry data and looks for terms that often are used as the basis for weak passwords. These weak terms are added to a *global banned password list* maintained by Microsoft. This list is not public, nor based on any online lists available. As stated by Microsoft *"To improve security, Microsoft doesn't publish the contents of the global banned password list."* [33]. The password filter also allows for some limited customization where up to 1000 terms can be added to the blocklist. This is recommended to be done with brand names, product names, company locations, and internal company-specific terms. The evaluation of passwords is done in 2 steps, first with normalization where all uppercase characters are changed to lowercase and common character substitutions are performed. In the second step, the password is examined for matching behaviour against the global banned list with fuzzy matching and substring matching. Then a score is calculated on the password and depending on this score the password is either rejected or accepted. [33]

OpenPasswordFilter (OPF) [54]. This open source Active Directory password filter is created to give a basic solution to reject common passwords based on a blocklist. The filter can do both partial matching and full matching at the same time. This can be done by configuring two distinct blocklists, one for partial matching and one for full matching. The recommended setup is to use partial matching on the most common passwords such as *'welcome'* and *'password'*. The filter does come with a pre-configured blocklist and recommendations for creating new blocklists with mangling rules from Hashcat. For our testing we will run the filter with the following settings as seen in Table 4.3.

Recommended-setup test:
OpenPasswordFilter password lists
HashCat password mangleded list
Partial matching enabled
Full matching enalbed

Table 4.3: OpenPasswordFilter test configuration

Lithnet Password Protection. Lithnet password protection, is an open source password filter for Active Directory, that can do various combinations of different checks on passwords. The filter can do direct blocking on passwords defined in a blocked password list, and has built in capabilities to use the HaveIBeenPwned [21] data for direct blocking. The filter can also block passwords with partial matching based on certain words, it does this with a normalization process where common character substitutions and weak obfuscation attempts are still blocked. The filter can also define complexity requirements based on length, where a threshold length defines if password complexity is required or not. The filter also supports custom regex matching, either for allowing or rejecting passwords. Finally, the filter has a points-based complexity where points are assigned for the use of certain characters and categories, and a minimum point-threshold is required for allowing passwords. The Lithnet setup we use can be seen in Table 4.4.

Recommended-setup test:
HaveIBeenPwned password lists
Partial matching enabled
Full matching enabled

Table 4.4: Lithnet test configuration

4.5 Automated Password Change Script

To perform our tests we developed a script to automatically change the password of a user in Active Directory. We developed the script using the Powershell scripting language, as to use the built-in Active Directory cmdlets [31].

4.5.1 Script Optimizations

Since we are attempting roughly 88 million password changes on three different Active Directory setups, we have implemented various methods to optimize the scripts used for automatically changing passwords in Active Directory.

4.5.1.1 Parallel Programming

To increase the amount of password changes we are able to do, we implemented parallel programming, to simultaneously attempt password changes. Since the script is reading passwords from password files line by line, we had to split the files to accommodate the parallel programming. To split the files we used the Powershell script created by Typhlosaurus [56], seen in Listing 1.

```
$from = "C:\Users\Administrator\Desktop\Passwords\linkedinclean.txt"
$rootName = "C:\Users\Administrator\Desktop\Passwords\Split\split"
$ext = "txt"
$limit = 30
$upperBound = (Get-Item $from).Length / $limit
$fromFile = [io.file]::OpenRead($from)
$buff = new-object byte[] $upperBound
$count = $idx = 0
try {
    do {
        $count = $fromFile.Read($buff, 0, $buff.Length)
        if ($count -gt 0) {
            $to = "{0}.{1}.{2}" -f ($rootName, $idx, $ext)
            $toFile = [io.file]::OpenWrite($to)
            try {
                #"Writing $count to $to"
                $tofile.Write($buff, 0, $count)
            } finally {
                $tofile.Close()
            }
        }
        $idx ++
    } while ($count -gt 0)
}
finally {
    $fromFile.Close()
}
```

Listing 1: Automated password script - Splitting the password lists

We ran multiple experiments to determine the optimal amount of processes for our tests, and during these experiments we realized that there is a upper limit to how often Active Directory can handle password changes. This means that we cannot infinitely increase the number of processes to further increase the speed. We also realized that this limit varied for the different filters. From our experiments we found 30 processes to be the optimal amount. More than 30 processes would throttle the filter, whereas no problems occurred at 30, and we still got a performance increase.

We also encountered a problem when the processes tried logging the results to the same file. To remedy this we opted to log all the password change results from each process to individual files. Once all password change attempts had been made, we would combine the individual files containing the results. For combining the result files we used the following Powershell code seen in Listing 2.

```powershell
foreach($list in $rejectedList) {
    $file = "C:\Users\Administrator\Desktop\Passwords\Split\rejected\" + ($list | Select-Object -ExpandProperty Name)

    $fromFile = [io.file]::OpenRead($file)
    $toFile = [io.file]::Open("C:\Users\Administrator\Desktop\Passwords\Split\rejected\combined.txt", [io.FileMode]::Append, [io.FileAccess]::Write)
    $buff = new-object byte[] $upperBound

    try {
        $count = $fromFile.Read($buff, 0, $buff.Length)

        $tofile.Write($buff, 0, $count)
    } finally {
        $tofile.Close()
    }
    $fromFile.Close()
}
```

Listing 2: Combining multiple result files

Since we were using parallel programming for the automatic password changes we had to create multiple users in our Active Directory setups. This was done to avoid a users password change attempt being locked by another process. Each process would attempt a password change for a separate user. For the creation of multiple users we used the Powershell script seen in Listing 3

```
$count = 30
for($i = 0; $i -le $count; $i += 1){
$splat = @{
    Name = 'test' + $i
    AccountPassword = (ConvertTo-SecureString -AsPlainText 'AccountPassword'
    ↪  -Force)
    Enabled = $true
}
New-ADUser @splat
}
```

Listing 3: Powershell script for creating AD users

4.5.1.2 Changing Passwords with Cmdlets and ADSI

We ran into a problem where using the Active Directory cmdlet `Set-ADAccountPassword` was being used by too many processes, and it would result in a transient error. This would happend if more than 20 processes was using the cmdlet, and depending on the filters speed, it could happen down to as few as 5 processes. We therefore implemented Active Directory Service Interfaces (ADSI), which is a set of COM interfaces that can be used to access Active Directory features through network providers. This meant we could call a password change through LDAP. For this we used modified code by ScottyDoo [48] seen in Listing 4.

```
$User = "test" + $number
$DomainDN = $(([adsisearcher]"").SearchRoot.path)
$Filter = "(&(objectCategory=person)(objectClass=user)(samaccountname=$User))"
$Searcher = New-Object System.DirectoryServices.DirectorySearcher
$Searcher.Filter = $Filter
$Searcher.SearchScope = "Subtree"
$Searcher.SearchRoot = New-Object
↪  System.DirectoryServices.DirectoryEntry('LDAP://CN=test' + $number +
↪  ',CN=Users,DC=passwordfilter,DC=local')
$objUser = $Searcher.FindOne().GetDirectoryEntry()
$objUser.PsBase.Invoke("SetPassword", $password)
$objUser.CommitChanges()
```

Listing 4: ADSI Password Change Script

While the change to ADSI solved the problem of limited cmdlet use, we found it to be slightly slower than the cmdlet in single use cases. Nevertheless, the speed improvement from parallel programming compensates for this.

4.5.1.3 Filters Impact On Speed

As briefly mentioned in Section 4.5.1.1, we found the different filters would have varying speeds with our scripts. An example of this is the OPF being the slowest at password changes. We ran experiments to find out if the parallel programming would increase the speed for a given password filter, and found that for some of the filters the parallel programming would actually slow down the password changing. The results in Table 4.5 are from running our scripts with and without parallel programming on 10.000 random passwords.

Filter	Parallel	Singular
OpenPasswordFilter	50 min.	22:30 min.
Improsec Filter	27 min.	14:40 min.
Entra Filter	2:36 min.	2:07 min.

Table 4.5: Password Changer script speed differences

We choose to look into why the filters behaved this way and while the results might seem surprising, after looking into just one of the filters we found that the different implementations of the filters can impact the speed signficantly. We investigated the slowest filter, OPF, and discovered that the installed DLL opens a network socket connection to a C# program that does the actual password filtering. This also explains why OPF becomes slower when using the script with parallel programming. The socket is likely getting overwhelmed with connections, thereby we indirectly are doing a Denial of Service attack on the filter. The OPF code responsible for the network socket can be seen in Listing 5:

```
// In this function, we establish a TCP connection to 127.0.0.1:5999 and
↪  determine
// whether the indicated password is acceptable according to the filter
↪  service.
// The service is a C# program also in this solution, titled "OPFService".
unsigned int __stdcall CreateSocket(void *v) {
        //the account object
        PasswordFilterAccount *pfAccount =
        ↪  static_cast<PasswordFilterAccount*>(v);
        SOCKET sock = INVALID_SOCKET;
        struct addrinfo *result = NULL;
        struct addrinfo *ptr = NULL;
        struct addrinfo hints;
        bPasswordOk = TRUE; // set fail open
        int i;
        ZeroMemory(&hints, sizeof(hints));
        hints.ai_family = AF_UNSPEC;
        hints.ai_socktype = SOCK_STREAM;
        hints.ai_protocol = IPPROTO_TCP;
        // This butt-ugly loop is straight out of Microsoft's reference example
        // for a TCP client.  It's not my style, but how can the reference be
        ↪  wrong? ;-)
        i = getaddrinfo("127.0.0.1", "5999", &hints, &result);
        if (i == 0) {
                for (ptr = result; ptr != NULL; ptr = ptr->ai_next) {
                        sock = socket(ptr->ai_family, ptr->ai_socktype,
                        ↪  ptr->ai_protocol);
                        if (sock == INVALID_SOCKET) {
                                break;
                        }
                        i = connect(sock, ptr->ai_addr, (int)ptr->ai_addrlen);
                        if (i == SOCKET_ERROR) {
                                closesocket(sock);
                                sock = INVALID_SOCKET;
                                continue;
                        }
                        break;
                }
                if (sock != INVALID_SOCKET) {
                        askServer(sock, pfAccount->AccountName,
                        ↪  pfAccount->Password);
                        closesocket(sock);
                }
        }
        return bPasswordOk;
}
```

Listing 5: OpenPasswordFilter DLL socket from dllmain.cpp [54]

4.5.2 Final Scripts

After all the improvements and testing, we ended up with two scripts for automatically calling password changes in Active Directory. A script that split files, uses ADSI and parallel programming. And a script that iterates through the password lists, using the Active Directory cmdlet to change passwords sequentially for each entry in the password list. The simple script using the cmdlet can be seen in Listing 6:

```powershell
param (
    [string]$filePath = "C:\Users\Administrator\Desktop\Passwords\linkedinclean.txt",
    [string]$allowedPasswords = "C:\Users\Administrator\Desktop\Passwords\RejectedAndAccepted\AllowedPasswords.txt",
    [string]$deniedPasswords = "C:\Users\Administrator\Desktop\Passwords\RejectedAndAccepted\DeniedPasswords.txt"
)

$stopWatch = [System.Diagnostics.Stopwatch]::StartNew()

$reader = New-Object -TypeName System.IO.StreamReader -ArgumentList $filePath

while ($password = $reader.ReadLine() ) {
#foreach ($password in $passwords) {
    try {
        Set-ADAccountPassword -Identity 'CN=test0,CN=Users,DC=passwordfilter,DC=local' -Reset -NewPassword (ConvertTo-SecureString -AsPlainText $password -Force)

        # The password change succeeded, so log the password in the allowed list
        Out-File -FilePath $allowedPasswords -Append -Encoding utf8 -InputObject $password
    }
    catch {
        # The password change failed, so log the password in the denied list
        Out-File -FilePath $deniedPasswords -Append -Encoding utf8 -InputObject ($password + " " + $_.Exception.Message)
    }
}
$stopWatch.stop()
Write-Output("Time: " + $stopWatch.Elapsed)
```

Listing 6: Simple script that uses cmdlets for changing passwords in Active Directory

A flowchart illustration of the simple scripts workflow can be seen in Figure 4.2. The figure shows how the script first initializes parameters and starts a stopwatch. Then by using a Streamreader, a list of passwords is looped over. For each password, a password change is attempted using a cmdlet and the results are logged. Once the password list has been looped over, the script ends.

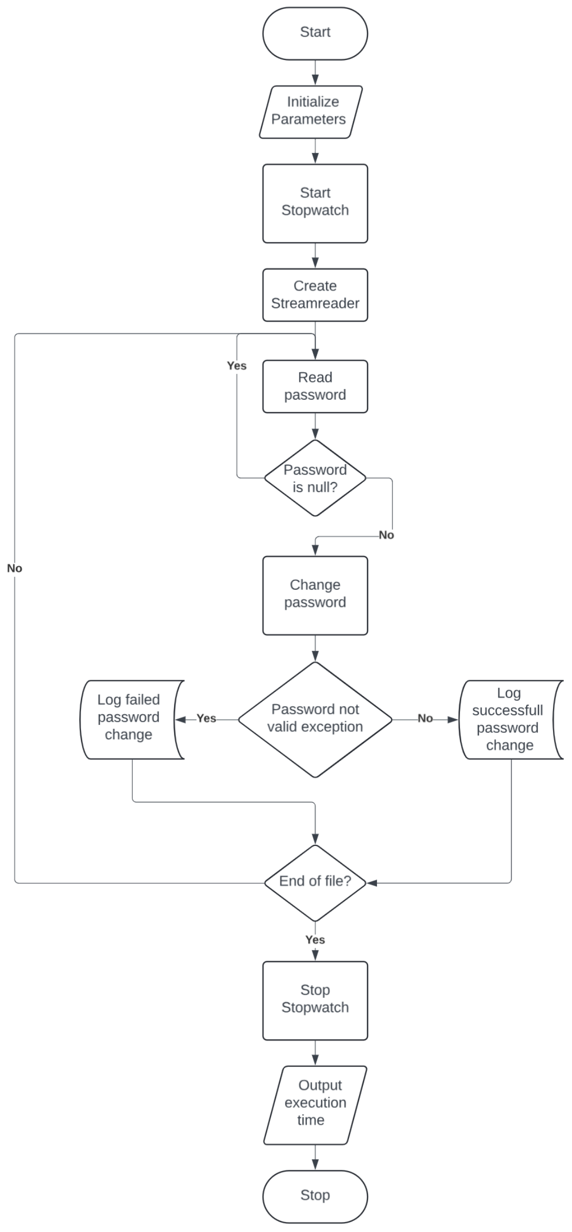


Figure 4.2: Flowchart depicting the flow of the simple script for changing passwords.

The full advanced script that utilizes file splitting and parallel programming, can be found in appendix A.1. The advanced script can be seen in Algorithm 1. The three main steps of the script is, splitting of files, the parallel password changing, and the combining of results. The password changing step, is following the same approach as the simple script from Listing 6, but uses ADSI for chagning passwords instead of a cmdlet.

Algorithm 1: Parallel Programming Script for Changing Passwords in Active Directory

```
Data: Source path, target path, split limit
Result: Password list file is split files, processed for password changes and logged,
        then the results are combined
1  Start timer
2  Define paths and limits
3  Calculate upperBound from file size and split limit
4  Open source file for reading
5  while bytes read > 0 do
6      Read from source file into buffer
7      if bytes read > 0 then
8          Open target split file for writing
           // Write buffer to target file and close it
9          Write buffer to target file
10         Close target file
11     end
12     Increment file index
13 end
14 Close source file
15 Enumerate all split files
16 foreach file in split list do
       // Parallel processing of files
17     foreach line in file as password do
           // Attempt to set password using ADSI and log result
18         Try to set password
19         if password change succeeded then
20             Log accepted password
21         else
22             Log rejected password
23         end
24     end
25 end
26 Function CombineFiles(rejectedList, acceptedList):
27     Combine files in rejectedList and acceptedList
28     Delete temporary split files
29 return
30 Stop timer and output elapsed time
```

A flowchart illustration of the advanced script's workflow can be seen in Figure 4.3, where three lanes show the three main steps of the script. The parallel programming is encompassed in the middle lane of the figure. Otherwise, the main difference compared to Figure 4.2, is the addition of lanes, that each handles the different steps of the script.

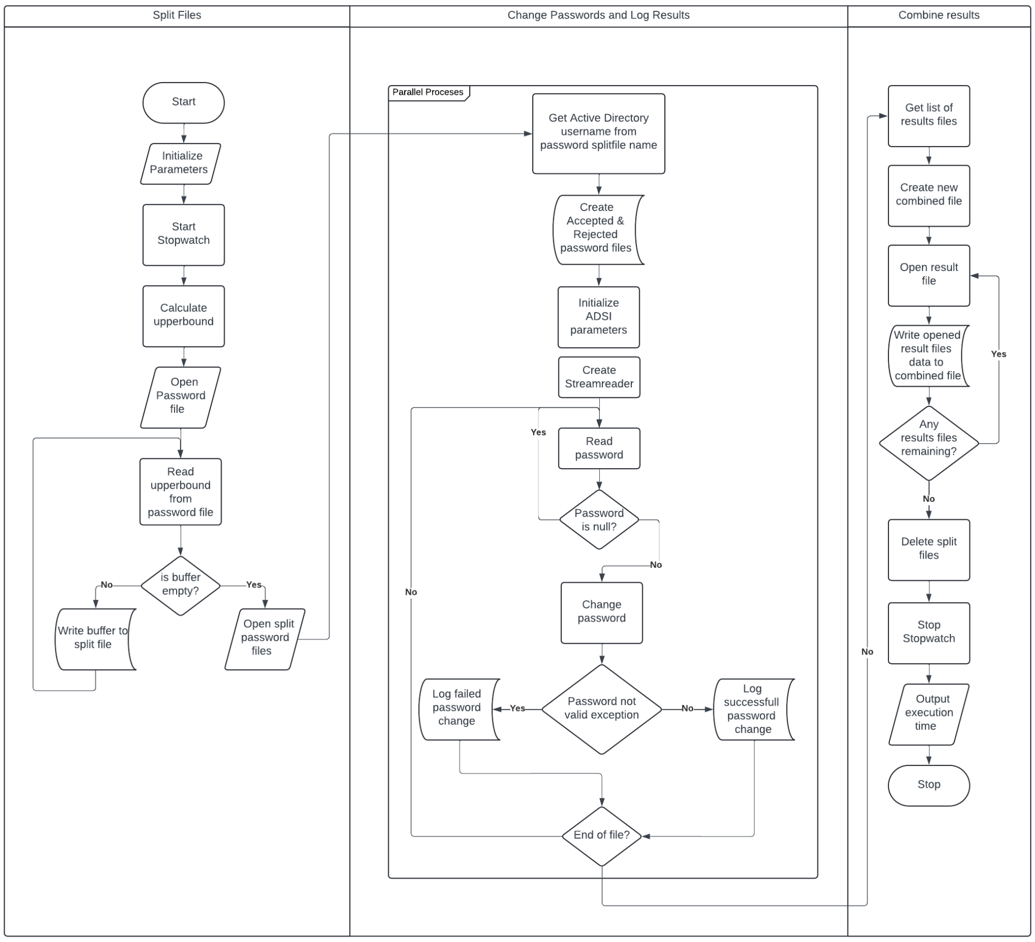

Split Files
Change Passwords and Log Results
Combine results
Start
Initialize Parameters
Start Stopwatch
Calculate upperbound
Open Password file
Read upperbound from password file
is buffer empty?
No
Yes
Write buffer to split file
Open split password files
Parallel Proceses
Get Active Directory username from password splitfile name
Create Accepted & Rejected password files
Initialize ADSI parameters
Create Streamreader
Read password
Yes
Password is null?
No
Change password
Password not valid exception
Yes
No
Log failed password change
Log successfull password change
End of file?
No
Get list of results files
Create new combined file
Open result file
Write opened result files data to combined file
Any results files remaining?
Yes
No
Delete split files
Stop Stopwatch
Output execution time
Stop
No

Figure 4.3: Flowchart depicting the flow of the password changing script which utilizes parallel programming.

4.6 Password Strength Measurement

Using the setup described in Section 4.2 with the scripts from Section 4.5, we obtain lists of accepted passwords for each password filter. To find the strength of the filters, we use the lists of accepted passwords together with a number of methods to analyze the strength of the individual passwords.

While the majority of the filters we tested are quite customizable, our method remains applicable to them. However, the scores we calculate is specific to the settings used for each filter. If the settings are changed for the filter, our method is still relevant, but it would require retesting to get a new accurate score for the changed settings.

The methods used to analyze the strength of the individual passwords is based on previous research, such as papers working with Password Strength Meters (PSM) [62], [61], [15] [16], [9]. We have chosen two overall methods to analyze the strength of the individual passwords. These two methods are password guessing attacks and PSMs.

4.6.1 Password Strength Meters

To calculate the overall strength of the password filters, our method uses four different PSMs, to calculate the strength of the individual passwords that were accepted by the Active Directory password filters. There exists a wide number of PSMs, so to chose the strength meters for our method we looked at research done relating to PSMs. Wang et al. [62] did an extensive evaluation of PSMs accuracy, and from their work we have chosen the following PSMs to incorporate in our method:

- zxcvbn [63] [65]
- fuzzyPSM [61] [60]
- MultiPSM [15] [16] [25]

To also include PSMs from other sources, we haven chosen the following PSM from the works of Dell'Amico et al., which calculates strength in terms of guesses needed to guess a given password:

- Monte Carlo PSM [9] [8]

The chosen PSMs are all different types, meaning that either their method of calculating strength or the feedback form is different. This will help give us a broader and potentially more correct evaluation of the passwords strengths. The PSMs can be seen in Table 4.6, where the PSMs feedback form, and type can be seen. Here the feedback form represents how the strength score is returned from the PSM, and the type refers to the underlying password strength evaluation method.

Name	Release Date	Type	Feedback form
zxcvbn	2017	Pattern Detection	Guess number
fuzzyPSM	2019	Attack Algorithm	Password probability
MultiPSM	2019	Multimodal	Fusion Score
Monte Carlo PSM	2017	Attack Algorithm	Guess number

Table 4.6: Final Password Strength Meters chosen

4.6.1.1 zxcvbn

The zxcvbn PSM presented by Wheeler [63], works as a low budget alternative for password strength estimation. It has since become a stable PSM in academic work relating to password strength. The PSM itself is very easy to implement and has been ported to numerous programming languages [65]. We have chosen to work with the Python implementation of zxcvbn [66]. In Listing 7 the Python script we have developed for calculating the zxcvbn scores can be seen, the script iterates over a list of passwords and returns the average strength of all the passwords. We use this script on the lists of accepted passwords for each filter.

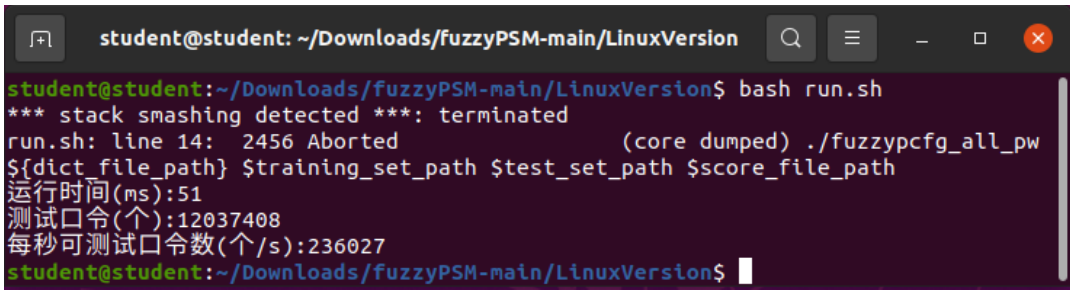
```python
from zxcvbn import zxcvbn
import json
import statistics
from tqdm import tqdm
results = []
passwords = []
with open('/home/student/Documents/Results/Lithnet/top1mil/rejected.txt', 'r')
↪  as file:
    for password in tqdm(file):
        password = password.rsplit(" ", 1)
        if str.isspace(password):
            continue
        x = zxcvbn(password.replace("\n", ")
        passwords.append(password)
        score = x["score"]
        results.append(score)

print("Mean: ", statistics.mean(results))
```

Listing 7: Python script for calculating mean zxcvbn score on a list of passwords

4.6.1.2 fuzzyPSM

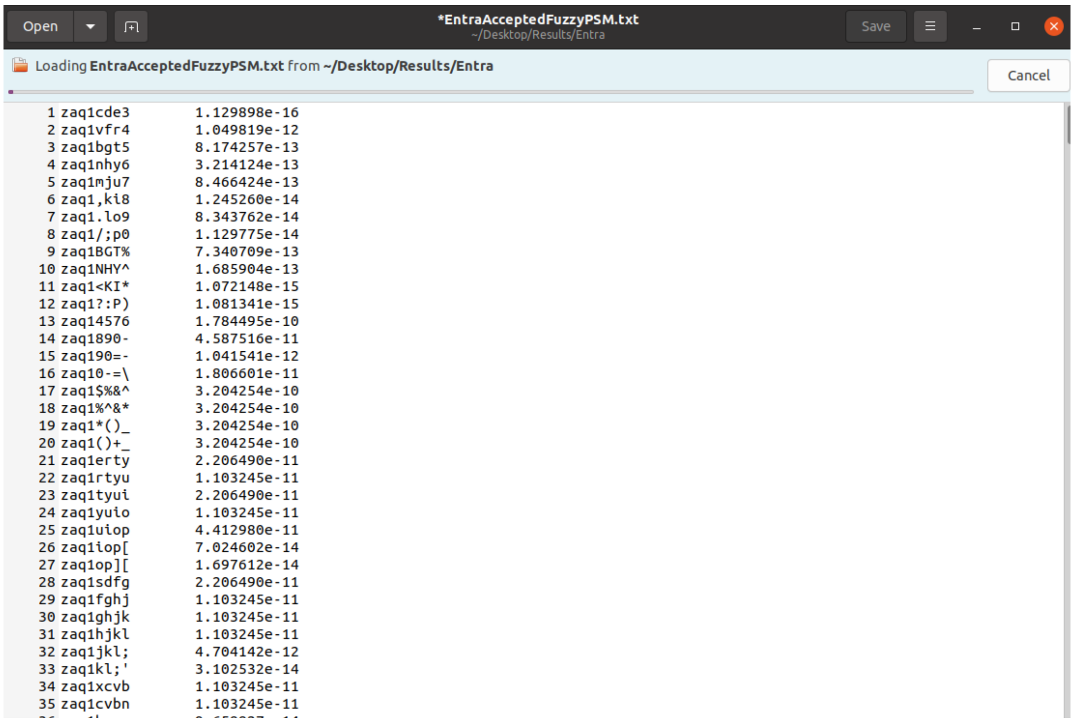
The fuzzyPSM was created by Wang et al. and is a PSM built on probabilistic context-free grammars. It is trained on leaked password data, as to capture the essence of users picking passwords based on previously used passwords. The authors have made fuzzyPSM public as an executable [60]. For our use we follow their instructions for running the PSM on Linux. In Figure 4.4, a terminal from a Linux system can be seen running the fuzzyPSM.



```
student@student:~/Downloads/fuzzyPSM-main/LinuxVersion$ bash run.sh
*** stack smashing detected ***: terminated
run.sh: line 14:  2456 Aborted                 (core dumped) ./fuzzypcfg_all_pw
${dict_file_path} $training_set_path $test_set_path $score_file_path
运行时间(ms):51
测试口令(个):12037408
每秒可测试口令数(个/s):236027
student@student:~/Downloads/fuzzyPSM-main/LinuxVersion$
```

Figure 4.4: fuzzyPSM running on a Linux system

In Figure 4.5 the results from running fuzzyPSM can be seen, the results are saved in a text file. The results are separated by each line, where the password can be seen followed by the score given by fuzzyPSM.


Open
*EntraAcceptedFuzzyPSM.txt
~/Desktop/Results/Entra
Save
Loading EntraAcceptedFuzzyPSM.txt from ~/Desktop/Results/Entra
Cancel

```
zaq1cde3        1.129898e-16
zaq1vfr4        1.049819e-12
zaq1bgt5        8.174257e-13
zaq1nhy6        3.214124e-13
zaq1mju7        8.466424e-13
zaq1,ki8        1.245260e-14
zaq1.lo9        8.343762e-14
zaq1/;p0        1.129775e-14
zaq1BGT%        7.340709e-13
zaq1NHY^        1.685904e-13
zaq1<KI*        1.072148e-15
zaq1?:P)        1.081341e-15
zaq14576        1.784495e-10
zaq1890-        4.587516e-11
zaq190=-        1.041541e-12
zaq10-=\        1.806601e-11
zaq1$%&^        3.204254e-10
zaq1%^&*        3.204254e-10
zaq1*()_        3.204254e-10
zaq1()+_        3.204254e-10
zaq1erty        2.206490e-11
zaq1rtyu        1.103245e-11
zaq1tyui        2.206490e-11
zaq1yuio        1.103245e-11
zaq1uiop        4.412980e-11
zaq1iop[        7.024602e-14
zaq1op][        1.697612e-14
zaq1sdfg        2.206490e-11
zaq1fghj        1.103245e-11
zaq1ghjk        1.103245e-11
zaq1hjkl        1.103245e-11
zaq1jkl;        4.704142e-12
zaq1kl;'        3.102532e-14
zaq1xcvb        1.103245e-11
zaq1cvbn        1.103245e-11
```

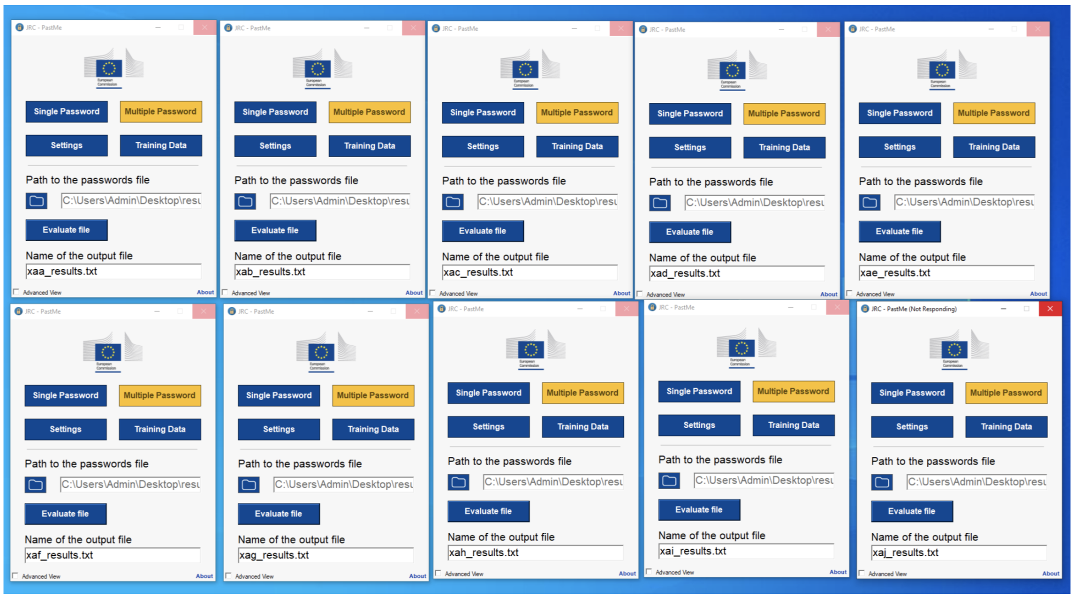
Figure 4.5: Result file from running the fuzzyPSM

4.6.1.3 MultiPSM

Galbally et al. [15] [16] developed a PSM that measures a passwords strength with multiple models, as mentioned in Section 3.2.1. Their PSM is publicly available as an executable program [25], which can handle both individual passwords or lists of passwords.

While using the provided executable, we noticed some problems. Some passwords would make the PSM crash, while it did not happen often, it would stop the program when evaluating a list of passwords. And since we are running the program on millions of passwords, it was happening more than once for each password list. To fix the crashing problem, we got the PSM's source code from GitHub and added exception handling to the evaluation of password files. This fixed the problem for us, and left us with an insignificant amount of passwords not getting properly evaluated.

In Figure 4.6, ten instances of the MultiPSM program can be seen running. We ran 10 instances due to the speed of the evaluations. This also meant that we had to split our passwords into multiple files for each instance of the MultiPSM program.



C:\Users\Admin\Desktop\resu

xaa_results.txt
xab_results.txt
xac_results.txt
xad_results.txt
xae_results.txt
xaf_results.txt
xag_results.txt
xah_results.txt
xai_results.txt
xaj_results.txt

Figure 4.6: 10 instances of MultiPSM running

4.6.1.4 Monte Carlo PSM

Dell'Amico et al. [9] created a PSM that uses Monte Carlo sampling on three probabilistic attack models as described in Section 3.2.1. The authors have made the code for their PSM public on GitHub [8]. In Listing 8, the Python code from the Monte Carlo GitHub and our own additions can be seen. The code calculates the Monte Carlo PSM scores for a given password list and then saves the password and the scores to a specified result file.

```python
# internal imports
import backoff
import model
import ngram_chain
import pcfg

with open("/home/student/Documents/StrengthEval/montecarlopwd/password.lst", 'rt') as f:
    training = [w.strip('\r\n') for w in f]
models = {'{}-gram'.format(i): ngram_chain.NGramModel(training, i)
          for i in range(2, 5 + 1)}
models['Backoff'] = backoff.BackoffModel(training, 10)
models['PCFG'] = pcfg.PCFG(training)
samples = {name: list(model.sample(10000))
           for name, model in models.items()}
estimators = {name: model.PosEstimator(sample)
              for name, sample in samples.items()}
modelnames = sorted(models)

passFile = "/home/student/Documents/Results/Entra/allRejectedremoved.txt"
outfile = "/home/student/Documents/Results/Entra/EntraAllRejectedMonte.txt"

with open(outfile, 'w') as fileOut:
    with open(passFile, 'r') as file:
        for password in tqdm(file):
            if str.isspace(password) == True:
                continue
            password = password.replace("\n", "")
            estimations = [estimators[name].position(models[name].logprob(password)) for name in modelnames]

            out = password + ", " + str(estimations[0]) + ", " + str(estimations[1]) + ", " + str(estimations[2]) + ", " + str(estimations[3]) + ", " + str(estimations[4]) + ", " + str(estimations[5])  + "\n"
            fileOut.write(out)
```

Listing 8: Python script for calculating the Monte Carlo PSM scores of a password list.

4.6.2 Password Cracking

Other than using PSMs, we also employ a number of guessing attack strategies for our password filter strength estimation method.

4.6.2.1 Offline Guessing Attacks

Offline guessing attacks represent the attacks where an adversary has seemingly unlimited attempts at guessing the password. Usually for this attack the adversaries have obtained the password hash. For the purposes of performing offline guessing attacks Hashcat will be used. The attacks will be done in steps, with increased complexity of the attacks in each step, thereby following the method developed by Galbally et al. [16]. This is to more accurately assess the strength of the passwords, considering that weaker passwords will be cracked with simpler attacks. An overview of the attack method can be seen in Figure 4.7. As illustrated in the figure, the lists of accepted passwords from the password filters undergo an attack session comprising of four distinct steps. The first step employs brute force to guess the passwords. The second step utilizes a dictionary attack. The third step involves a rule-based dictionary attack. Finally, the fourth step applies a rule-based attack using passwords generated by PassGAN from Section 4.3.1. Crucially, the passwords guessed in earlier steps are excluded from subsequent attacks, so that the filters can be accurately assessed based on what passwords were guessed in each step. The results are output to four files, detailing the number and specific passwords guessed at each step.

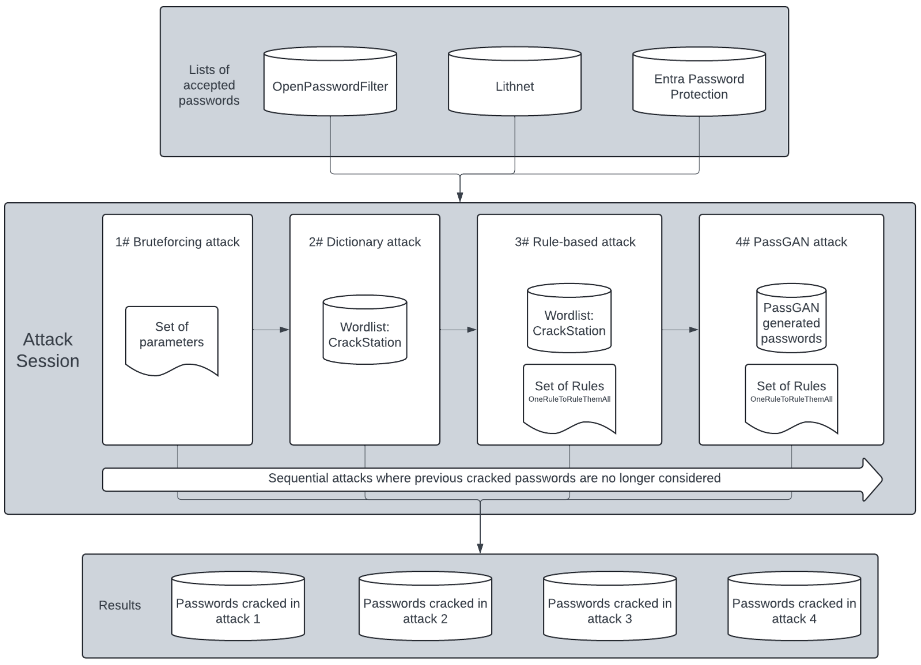

Lists of accepted passwords
OpenPasswordFilter
Lithnet
Entra Password Protection
Attack Session
1# Bruteforcing attack
Set of parameters
2# Dictionary attack
Wordlist: CrackStation
3# Rule-based attack
Wordlist: CrackStation
Set of Rules
OneRuleToRuleThemAll
4# PassGAN attack
PassGAN generated passwords
Set of Rules
OneRuleToRuleThemAll
Sequential attacks where previous cracked passwords are no longer considered
Results
Passwords cracked in attack 1
Passwords cracked in attack 2
Passwords cracked in attack 3
Passwords cracked in attack 4

Figure 4.7: Attack method for offline guessing attacks

In order to execute these attacks we utilize the GPUs available on CLAAUDIA [57] a cloud platform by Aalborg University, that specialize in High-Performance Computing (HPC) with GPUs. The entire Python script for offline password guessing attacks can be found in appendix A.2.

Brute Force Attacks In the first attack strategy, we try an exhaustive strategy at guessing the passwords. Various strategies have been proposed for this, and we follow the works of Galbally et al. [16] and Wang et al. [62]. The 5 methods presented by the authors that we will follow can be seen in Table 4.7. The methods are differentiated by two categories, the number of characters used, and max password length. The combination of these two categories gives the final number of guesses for each method.

Attack	Characters	Max Length	Number of guesses
1.A	10 digits	11	10^{11}
1.B	26 Lower Case	8	26^{8}
1.C	26 upper case	8	26^{8}
1.D	32 special characters	8	32^{8}
1.E	All 96 ASCII	6	96^{6}

Table 4.7: Brute force attacks

Listing 9 shows the Python function used to perform the 5 different guessing attacks from Table 4.7. Notably the script works by executing Hashcat commands through the subprocess module. The Hashcat commands used, have different flags set, and these flags define the settings for command being run. All the flags we use can be seen below:

- **-d 1,2** sets the devices used by Hashcat normally device 1 is the CPU but because of how CLAAUDIA[57] and singularity containers handle devices, when running the container device 1 and 2 is the GPUs.
- **-m 1000** is the type of hash we are working with 1000 is the hash number for NTML hashes.
- **–attack-mode 3** sets Hashcat in brute force mode.
- **–increment** tells Hashcat to start brute forcing with a password length of 1 and then increment the password length when Hashcat has exhausted its options.
- **–increment-max** defines the max password length Hashcat will try.
- The string of characters seperated by '?' like '**?d?d?d?d?**' is the mask hashcat uses for the attack. This defines the character set used for each character in the password.

```python
def brute(file_path):
    hash_cmd = f'hashcat -d 1,2 -m 1000 --attack-mode 3 --increment
    ↪  --increment-max=11 {file_path} "?d?d?d?d?d?d?d?d?d?d?d" -o
    ↪  hashcat_attack1.txt --outfile-format=2 --potfile-path=potfile.pot'
    subprocess.run([hash_cmd], shell=True)
    hash_cmd = f'hashcat -d 1,2 -m 1000 --attack-mode 3 --increment
    ↪  --increment-max=8 {file_path} "?l?l?l?l?l?l?l?l" -o hashcat_attack1.txt
    ↪  --outfile-format=2 --potfile-path=potfile.pot'
    subprocess.run([hash_cmd], shell=True)
    hash_cmd = f'hashcat -d 1,2 -m 1000 --attack-mode 3 --increment
    ↪  --increment-max=8 {file_path} "?u?u?u?u?u?u?u?u" -o hashcat_attack1.txt
    ↪  --outfile-format=2 --potfile-path=potfile.pot'
    subprocess.run([hash_cmd], shell=True)
    hash_cmd = f'hashcat -d 1,2 -m 1000 --attack-mode 3 --increment
    ↪  --increment-max=8 {file_path} "?s?s?s?s?s?s?s?s" -o hashcat_attack1.txt
    ↪  --outfile-format=2 --potfile-path=potfile.pot'
    subprocess.run([hash_cmd], shell=True)
    hash_cmd = f'hashcat -d 1,2 -m 1000 --attack-mode 3 --increment
    ↪  --increment-max=6 {file_path} "?a?a?a?a?a?a" -o hashcat_attack1.txt
    ↪  --outfile-format=2 --potfile-path=potfile.pot'
    subprocess.run([hash_cmd], shell=True)
```

Listing 9: Python function for conducting brute force attacks.

Dictionary Attack For the second attack we use a wordlist, where the passwords from the wordlist is compared against the passwords we're trying to guess. The total number of guesses for this attack will depend on the wordlist used. For this thesis a wordlist of $63,941,069$ passwords from Crackstation was used. Again we are using a Python function seen in Listing 10 to run the Hashcat command. The listing also shows the flags used, which has some overlap with the flags used for the brute force attack.

```python
def dictionary(file_path, wordlist):
    hash_cmd = f'hashcat -d 1,2 -m 1000 {file_path} {wordlist} -o
    ↪  hashcat_attack2.txt --outfile-format=2 --potfile-path=potfile.pot'
    subprocess.run([hash_cmd], shell=True)
```

Listing 10: Python function for conducting dictionary attack.

Dictionary Attack + Rules For the third attack we use a wordlist as in the dictionary attack, but will also include a ruleset for creating mangled versions of each password in the wordlist. The total number of guesses for this attack depends on the wordlist and the ruleset used for the attack. The wordlist will be the same wordlist as the dictionary attack, and the ruleset will be the ruleset presented by Eckroth et al. [13]. With the wordlist and ruleset combined the amount of guesses will total $3.32 \cdot 10^{12}$. In Listing 11 the Python function used to execute this attack can be seen. As with the previous attacks Python functions, the flags are overlapping, but this time the ruleset is also used in the command.

```python
def rule(file_path, wordlist, ruleset):
    hash_cmd = f'hashcat -d 1,2 -m 1000 {file_path} {wordlist} -r {ruleset} -o
    ↪  hashcat_attack3.txt --outfile-format=2 --potfile-path=potfile.pot'
    subprocess.run([hash_cmd], shell=True)
```

Listing 11: Python function for conducting Dictionary+Rules attacks.

Guessing with PassGAN In recent years machine learning has been leveraged to enhance password guessing attacks. Hitaj et al. [20] developed PassGAN in which they utillized Generative Adverserial Networks, to efficiently and accurately generate passwords based of leaked password lists. These generated passwords are then used to perform guessing attacks.

The code for the PassGAN implementation by Dorsey [12] is almost 7 years old and unfortunately quite deprecated. Fortunately, a GitHub repository by beta6 [2] contains an updated version of the PassGAN implementation with less deprecated dependencies. The implementation requires the use of Nvidia's CUDA, which means an Nvidia GPU is necessary to run the implementation. To meet this requirement we used Aalborg University's AI cloud CLAAUDIA [57], which contains compute nodes with Nvidia GPUs specialized for machine learning. As mentioned in Section 4.3.1, the PassGAN model used was trained for 200,000 iterations with the LinkedIn leaked password data mentioned in Table 4.1.

As per the recommendations of Hitaj et al. [20] the PassGAN generated passwords will be combined with a rule set for a rule-based attack. While Hitaj et al. recommends using Hashcats best64 ruleset. Eckroth et al. [13] has developed a ruleset that performs better, therefore the attack will be carried out using this ruleset. A list of *100,000,000* passwords was generated to perform this attack. The python script used to execute this attack can be seen in Listing 12

```python
def PassGAN(file_path, wordlist, ruleset):
    hash_cmd = f'hashcat -d 1 -m 1000 {file_path} {wordlist} -r {ruleset} -o
    ↪  hashcat_attack4.txt --outfile-format=2 --potfile-path=potfile.pot'
    subprocess.run([hash_cmd], shell=True)
```

Listing 12: Python function for conducting PassGAN attacks.

Building the Container To use CLAAUDIA [57] and its resources one has to run programs and processes inside a Singularity container. Singularity containers are specifically for High Performance Computing (HPC) such as the platform CLAAUDIA. To build a container that has the requirements to run Hashcat on CLAAUDIA, we use .def build files in order to construct the container. The .def build file we used can be seen in Listing 13, here it can be seen that we are using a docker image from dizcza [10].

```
Bootstrap: docker
From: dizcza/docker-hashcat

%post
    apt-get -y update
    apt install -y python3
```

Listing 13: The .def build file used to build the container for CLAAUDIA.

The build file allows us to use a prebuilt docker container with Hashcat installed and install Python in order to run the python script that will be used to perform the guessing attacks.

4.6.2.2 Online Guessing Attacks

As described in Section 2.6.2, online guessing attacks is limited by the number of guesses an attacker is allowed to make. This is usually due to a lockout threshold that defines how many incorrect login attempts can be made before the account gets locked. Wang et al. [62] explores how a threshold limit can be circumvented by delaying the guesses in the guessing attack. By limiting the number of guesses to less than the set lockout threshold during each lockout period, the threshold is effectively circumvented. For example, if the threshold of failed logins is 10 every hour, then an attacker can attempt up to 9 guesses each hour. Active Directories have a custom account threshold lockout policy, which is why we have decided to test different thresholds. We will also be acting as a general attacker without any knowledge of the password distribution. In Table 4.8, the 6 different threshold combinations we are using can be seen. The table shows that we use 3 distinct lockout thresholds, and 2 different lockout duration's, giving us a total of 6 different thresholds. The attempts per day in the table are made by the simple calculation:

$$attempts\ per\ day = lockout\ threshold - 1 \cdot (\frac{lockout\ duration}{60} \cdot 24)$$

and the attempts per month are made by multiplying the attempts per day with 30.

Lockout Threshold	Lockout Duration	Attempts per day	Attempts per month
5	15	384	11520
5	30	192	5760
10	15	864	25920
10	30	432	12960
20	15	1824	54720
20	30	912	27360

Table 4.8: Number of login attempts given different lockout thresholds and durations.

To determine if the passwords filters are effective against an online password guessing attack, we compare lists of the most common passwords against the lists of the filters accepted passwords. Utilizing multiple lists ensures thoroughness, allowing us to comprehensively evaluate the filters resistance to common password-based attacks. The password lists we use can be seen in Table 4.9, where we see lists with a length of 100000, 10000, and 200 passwords.

Source	Number of passwords
Seclists	100000
Wikipedia	10000
Nordpass	200

Table 4.9: Common password lists used for online guessing attack.

The Python script for the online password guessing attacks seen in Listing 14, it shows that the thresholds chosen are 50, 100, 200, 10000 and 54720, these are based of the thresholds from Table 4.8. The script also shows that we don't actually perform the guessing attack, since we already have the list of accepted passwords, it is simply a matter of checking the common passwords against this list.

```python
def count_matching_entries(file1, file2, num_entries):
    with open(file1, 'r') as f1:
        entries1 = set(f1.read().splitlines()[:num_entries])
    with open(file2, 'r') as f2:
        entries2 = set(f2.read().splitlines())
    matching_entries = entries1.intersection(entries2)
    return len(matching_entries)
file1_path = 'seclist100k.txt'
file2_path = 'allAcceptedOPF.txt'
guesses = [50, 100, 200, 10000, 54720]
for i in range(5):
    matching_count = count_matching_entries(file1_path, file2_path, guesses[i])
    print(f"Passwords guessed with {guesses[i]}:", matching_count)
```

Listing 14: Python script for conducting simulated online password guessing attacks.

4.7 Calculating Overall Score

To calculate the overall score for the filters, we use two different methods. We have chosen to calculate a final score and a weighted final score. The reason for calculating two final scores is to provide different perspectives on the data. A weighted score can account for PSMs or guessing attacks not performing as expected or intended. The weights can also be used to highlight scores that should have more impact on the final result, for example if most passwords accepted by a filter could be guessed using a simple dictionary attack, that would be indicative of a very bad filter and the score should reflect that. The unweighted final score is calculated with a simple average calculation:

$$x = \frac{\sum_{i=1}^{n} x_i}{n}$$

To calculate the weighted score, we follow the works of Galbally et al. [15], and use a combination algorithm that has been proven to work well, namely the *weighted sum* algorithm. This algorithm combines the different scores into a final score in a linear equation:

$$x_{weighted} = \sum_{k=1}^{K} w_k s_k$$

Where w_k is the weight given to each of the individual scores s_k, and with the weights following:

$$\sum_{k=1}^{K} w_k = 1$$

4.7.1 Normalizing Scores

Since the different methods we use to calculate password strengths returns results in different formats, we have to find a way to combine these results together in a meaningful manner. For this we use normalization, where the different strength scores are normalized to the same value range. This will give us comparable results from the different PSMs and guessing attacks. We have chosen to normalize to a value range of $0 - 1$.

To calculate the normalized values, we are using two methods of normalization. We will use both methods for all the PSM and guessing attack results, and compare how the different methods impact the final results. The first method of normalization is a simple min-max normalization:

$$Normalized\, Score = \frac{Score - Min.\, Score}{Max.\, Score - Min.\, Score}$$

The second method for normalization is using the logarithmic function, this method can be especially useful for data consisting of huge numbers. Since the Monte Carlo PSM returns a estimated guess number which can be quite high, this method might be better suited here:

$$Log\ Normalized\ Score = \frac{log(Score + 1)}{log(Max.\ Score)}$$

In this formula, we still use a maximum score. Since $log(0)$ is undefined, and log(1) is 0, we need to ensure that the max score is always above 1.

4.7.1.1 fuzzyPSM Normalisation

Since the fuzzyPSM [61], returns a number probability of the password where a lower probability is a better score. This is opposed to the other PSMs we are using for our method. To solve this issue, we simply multiply the score with 100 and then subtract the multiplied score from 100. Given that the score is a probability and always falls within the range of $0 - 1$, subtracting the multiplied score from 100 gives us a new score where a higher value indicates better performance.

4.7.1.2 Password Guessing Attacks Normalization

For the password guessing attacks, the score we calculate is the percentage of passwords we are able to guess. This again leaves us in a situation where a lower number is a better score. Again to solve this issue we follow the same approach as for the fuzzyPSM normalization in Section 4.7.1.1.

4.8 Method Summary

Our method consists of using Virtual Machines hosting an Active Directory, where a password filter has been installed. For our method we collected a number of password lists from various sources. We use our collection of password data, in a Powershell script that automatically tests if each password is accepted or rejected by the Active Directory password filter. The script does so by attempting to change a users password to each password from our collection.

Once we have attempted a password change for all passwords in our collection, we end up with a list of accepted and rejected passwords for each filter. We then use various methods to calculate the strength of all the accepted passwords, these methods are state of the art Password Strength Meters and a number of different password guessing attacks.

When we have the results from the Password Strength Meters and guessing attacks, we use a normalization formula to normalize our results, and then calculate the overall score based on these normalized results. This leaves us with a overall strength score for each Active Directory password filter.

Chapter 5

Results

This Chapter contains the results from the method described in Chapter 4. We first showcase in Section 5.1, the results of the automatic password changing scripts for each of the Active Directory password filters. These results showcase what passwords were accepted and rejected by each filter. We then in Section 5.2, showcase the results of running all the accepted and rejected passwords through our chosen Password Strength Meters. This section also includes an analysis on the PSM results, such as looking at the impact of password length, and distribution of scores. In Section 5.3 the results from all the password guessing attacks is shown, and contains an analysis on the results. Finally, in Section 5.4 we perform the normalization calculations, and then calculate the final scores for the three different Active Directory password filters.

5.1 Passwords Accepted by Filters

The results presented in this section are the passwords accepted and rejected by each of the Active Directory password filters. The results are presented for each filter individually.

5.1.1 Lithnet Filter

We can see in Table 5.1 that the Lithnet filter only allowed a total of 13.62% of all the passwords, where majority of the denied passwords come from the LinkedIn password list. As seen in the table, the filter blocked almost 90% of the LinkedIn passwords, but that still left more than half a million LinkedIn passwords that got accepted. Furthermore, the list with the most accepted passwords is the PassGAN generated passwords, then followed by the Chinese and Russian password lists.

List name	Accepted	Rejected	Percentage accepted	Total
SecLists Keyboard-Combinations	6,089	3,539	63.2%	9,628
SecLists 10-million-password-list-top-1000000	68,102	931,922	6,81%	999,998
SecLists default-passwords	199	1,140	14.9%	1,339
Xato-Net	703,595	4,485,885	13.5%	5,189,480
PassGAN Generated	6,072,327	3,927,702	60.7%	10,000,029
Russian Passwords	2,175,474	324,556	87.01%	2,500,030
Chinese Passwords	2,444,126	55,902	97.7%	2,500,028
Greatest_books_of_all_time_originals	3,556	2,041	63.5%	5,597
RockYou	18,027	14,326,393	0.12%	14,344,420
Linkedin	544,622	52,245,057	1,04%	52,789,651
DanishTop5K	1,290	3,734	25.6%	5,024
EnglishTop1K	1	1,021	0.09%	1,022
Summed	12,037,408	76,308,890	13.62%	88,346,298

Table 5.1: Passwords accepted by the Lithnet filter

5.1.2 OpenPasswordFilter Filter

In Table 5.2 we can see the results of the OpenPasswordFilter, and that the results are quite similar to the ones from the Lithnet filter. OPF only allowed 17.7% of the total passwords and we can see that only 0.006% of the RockYou list was accepted. The most accepted passwords also come from the PassGAN passwords, then the Chinese and Russian passwords, just as with the Lithnet filter. A big difference compared to the Lithnet filter, is from the SecLists Keyboard-Combinations, where OPF accepts more than 90% of the list.

List name	Accepted	Rejected	Percentage accepted	Total
SecLists Keyboard-Combinations	8,710	894	90.6%	9,604
SecLists 10-million-password-list-top-1000000	86,047	913,951	8.604%	999,998
SecLists default-passwords	362	953	27.5%	1,315
Xato-Net	958,078	4,231,376	18.4%	5,189,454
PassGAN Generated	8,990,578	1,009,442	89.9%	10,000,020
Russian Passwords	2,197,955	302,065	87.9%	2,500,020
Chinese Passwords	2,453,296	46,724	98.1%	2,500,020
Greatest_books_of_all_time_originals	4,799	769	86.1%	5,568
RockYou	981	14,343,431	0.006%	14,344,391
Linkedin	1,018,297	51,771,294	1.9%	52,789,591
DanishTop5K	1,292	3,708	25.8%	5,000
EnglishTop1K	1	999	0.1%	1,000
Summed	15,720,396	72,625,606	17.7%	88,346,002

Table 5.2: Passwords accepted by the OpenPasswordFilter filter

5.1.3 Entra Password Protection

Finally, the results of Microsoft Entra Password Protection filter can be seen in Table 5.3. These results are drastically different than the two other filters, with 88.92% of the total passwords accepted. Interestingly the only password list that is more blocked by Entra compared to OPF and Lithnet, is the Chinese Passwords list.

List name	Accepted	Rejected	Percentage accepted	Total
SecLists Keyboard-Combinations	9,024	580	93.9%	9,604
SecLists 10-million-password-list-top-1000000	751,636	248,362	75.1%	999,998
SecLists default-passwords	592	723	45.01%	1,315
Xato-Net	4,401,441	788,013	84.81%	5,189,454
PassGAN Generated	9,530,481	469,519	95.3%	10,000,000
Russian Passwords	2,291,248	208,752	91.64%	2,500,000
Chinese Passwords	2,088,822	411,178	83.55%	2,500,000
Greatest_books_of_all_time_originals	5,244	324	94.1%	5,568
RockYou	12,399,288	1,945,103	85.43%	14,344,391
Linkedin	47,081,197	5,708,455	89.1%	52,789,652
DanishTop5K	2,699	2,301	53.89%	5,000
EnglishTop1K	228	772	22.8%	1,000
Summed	78,561,900	9,784,082	88.92%	88,345,982

Table 5.3: Passwords accepted by the Entra Password Protection filter

5.1.4 Summarized

In Figure 5.1, all three password filters results can be seen in a column plot. Here the difference between Entra and the two other filters can clearly be seen visualized. It is clear that Lithnet and OPF are overall more selective in what passwords they let through.

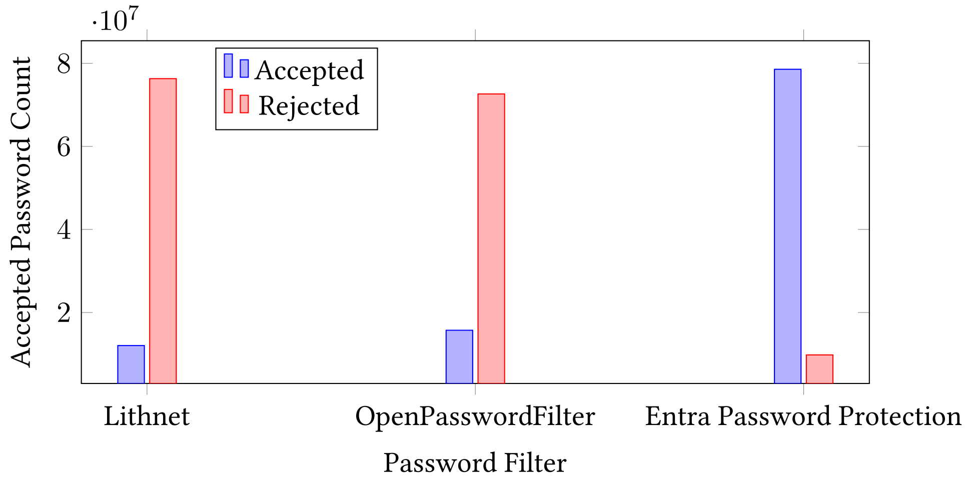


Figure 5.1: Column plot showcasing accepted and rejected passwords for each filter

We also looked at the different length distributions of the accepted passwords for each password filter. These findings can be seen in Figure 5.2, where we observe that Lithnet has the highest percentage of passwords with a length exceeding 10 characters, followed by

OPF, and then finally Entra. Interestingly, the opposite is true for passwords with a length between 5 and 10 characters. For passwords with a length less than 5 characters, we see that Lithnet and OPF are similar with respectively 4.06% and 3.19%, whereas Entra has 0%.

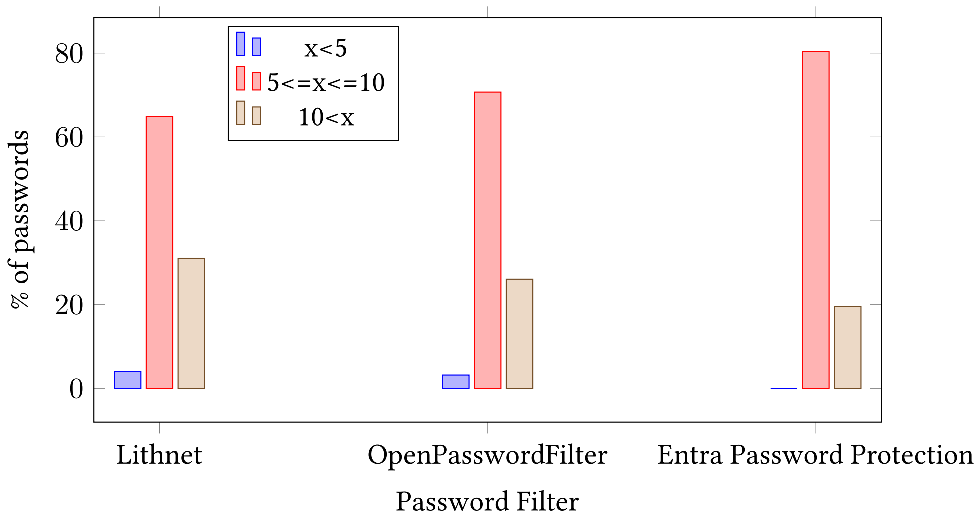


Figure 5.2: Column plot showcasing password length distribution for the three filters.

5.2 Password Strength Meter Results:

Here the results of the chosen PSMs can be seen. The results are presented for each PSM, where the password strength is calculated on all the combined accepted and rejected passwords.

5.2.1 zxcvbn

In Table 5.4 the scores from the zxcvbn PSM [63] can be seen. The results shows the average score for each filter, and we can see that Lithnet has the highest average score followed by OPF, and then Entra. While Lithnets average score is only 0.1 points higher than OPF, it is 0.44 points higher than Entra. For the rejected passwords we can see that Lithnet and OPF is very close in their scores, whereas Entras scores quite a bit lower. The reason for Entra to be scoring lower than both Lithnet and OPF in the rejected passwords, is most likely due to Entra not rejecting a lot of passwords in the first place.

	Accepted Passwords		**Rejected Passwords**	
Password Filter	**Number of passwords**	**Mean Score: 0-4**	**Number of passwords**	**Mean Score: 0-4**
Lithnet	12,037,408	2.667615708976309	76,308,890	2.038756946278184
OpenPasswordFilter	15,720,396	2.5606279103230247	72,625,555	2.030028066566927
Entra Password Protection	78,561,900	2.2247675555806	9,784,082	1.318926209864359

Table 5.4: zxcvbn PSM strength score results

Furthermore we have explored the impact of password length on the different PSM scores. In Figure 5.3 the impact of password length can be seen for the zxcvbn PSM [63] scores, and here we see that length has a significant impact on the scoring. Where for all three filters the highest scores are found for passwords with a length of at least 10 characters. Then passwords with a length between 5 and 10 characters, and finally passwords with a length less than 5 characters. We can see that Lithnet and OPF have very similar scores, whereas Entra is scoring slightly lower, and has no passwords with a length of less than 5 characters.

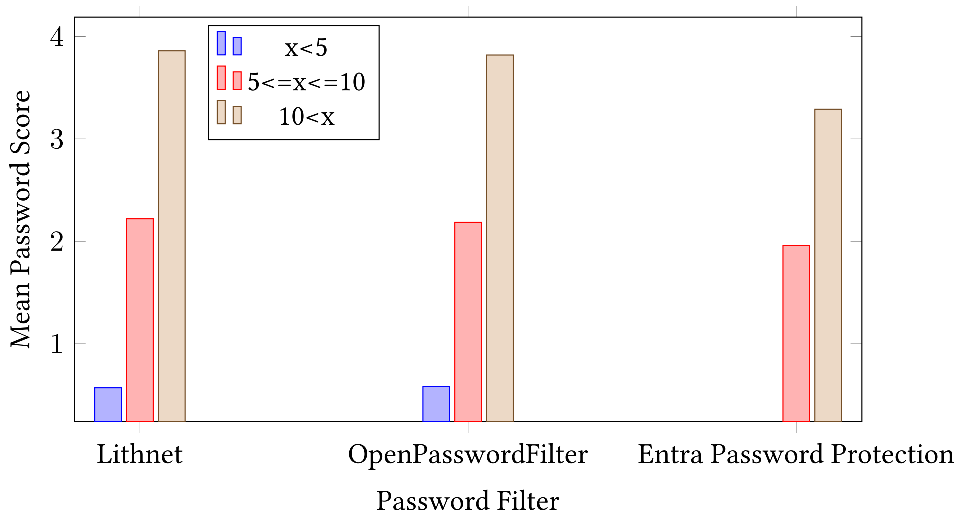


Figure 5.3: Column plot showcasing password length impact on zxcvbn PSM scores.

From the distribution of the zxcvbn scores on the three filters in Figures 5.4, 5.5, and 5.6, it can be seen that Lithnets scores are skewed towards scores of 2, 3, and 4. Where as OPF and Entra are more normally distributed with OPF leaning a bit more towards scores of 3 and 4, compared to Entra. Interestingly, no passwords from Entra are scored 0, which is likely is due to Entra having no passwords with less than 5 characters.

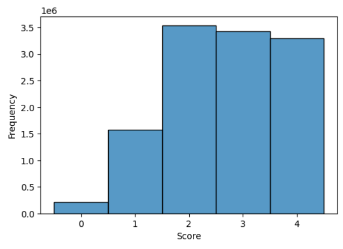


Figure 5.4: Histogram of zxcvbn scores on the accepted passwords from Lithnet.

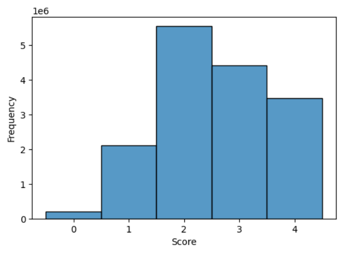


Figure 5.5: Histogram of zxcvbn scores on the accepted passwords from OPF.

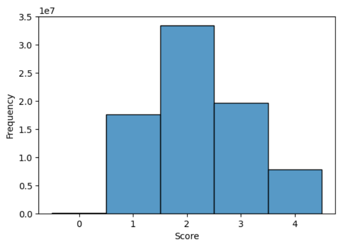


Figure 5.6: Histogram of zxcvbn scores on the accepted passwords from Entra.

5.2.2 MultiPSM

The results from the MultiPSM [15] can be seen in Table 5.5. The number of passwords used to calculate these scores is lower than the total amount of accepted passwords, this is due to MultiPSM not supporting non-ASCII characters, we have discarded these passwords for the MultiPSM scores. For the rejected OPF results, we only have 650,669 passwords. This is a random sample from all the passwords rejected by OPF. We used a sample here due to time constraints.

The results show that Lithnet have the highest Fusion Score, which is a fusion of the 4 other metrics: LIST, BF, AMM, and CFHMM. Although both OPF and Entra have a LIST score that is 0.01 higher than Lithnet, Lithnet still maintains a strong LIST score of 9.98. OPF leads in the BF score, but Lithnet has the highest score in the rest of the metrics, followed by OPF and then Entra.

Interestingly, for the rejected passwords, we can see that Lithnets rejected scores are still higher than Entras rejected scores. This is likely due to Lithnet being more selective with passwords in general, whereas Entra only rejects the very worst passwords.

	Accepted Passwords						**Rejected Passwords**					
Password filter	**Number of passwords**	**LIST Metric**	**BF Metric**	**AMM Metric**	**CFHMM Metric**	**Fusion of scores**	**Number of passwords**	**LIST Metric**	**BF Metric**	**AMM Metric**	**CFHMM Metric**	**Fusion of scores**
Lithnet	7,575,463	9.98	6.87	4.46	5.21	3.73	75,947,488	9.978	6.006	3.140	4.602	2.572
OpenPasswordFilter	11,320,668	9.99	7.09	4.06	4.98	3.48	650,669	9.900	3.817	2.609	4.207	1.698
Entra Password Protection	75,947,884	9.99	6.53	3.33	4.72	2.83	9,298,143	9.945	4.501	2.414	4.210	1.715

Table 5.5: MultiPSM strength score results

In Figure 5.7 the length impact on the MultiPSM scores can be seen. Here we again see that length has a significant impact on the scores. As with the zxcvbn results, we can see Lithnet having the highest score for all three password lengths, followed by OPF, and then finally Entra. Again Lithnet and OPF are very close in their results, as reflected from Table 5.5.

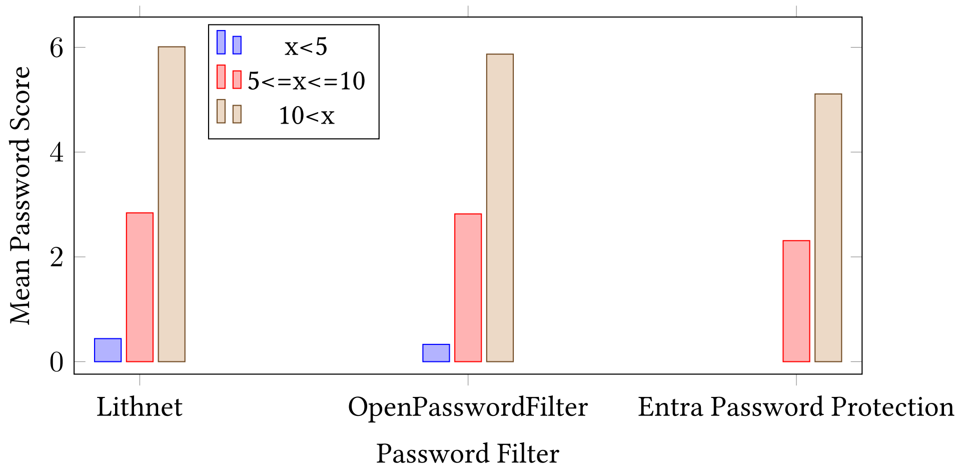


Figure 5.7: Column plot showcasing password length impact on MultiPSM score

In Figures 5.8, 5.9, and 5.10, the distribution for the MultiPSM [15] scores can be seen. The distributions show that Lithnet and OPF are quite similar, with OPF having more passwords scored between 0 and 1. Furthermore, the rest of Lithnets scores are rated slightly higher than those of OPF. Lithnets scores starts at around 4, whereas OPFs scores begin at around 3. Finally, we see that Entras distribution is comparable to the two others but it drops off more rapidly, likely leading to the overall lower score.

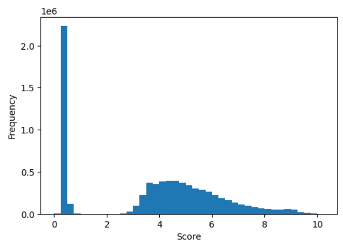


Figure 5.8: Histogram of MultiPSM scores on the accepted passwords from Lithnet

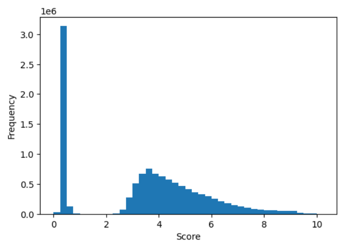


Figure 5.9: Histogram of MultiPSM scores on the accepted passwords from OPF

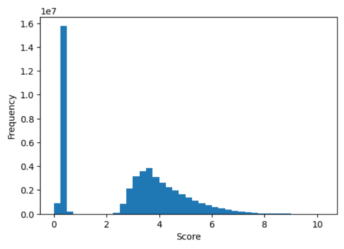


Figure 5.10: Histogram of MultiPSM scores on the accepted passwords from Entra

5.2.3 Monte Carlo

The results from the Monte Carlo PSM [9] can be seen in Table 5.6, where 6 distinct scores are calculated: 2-, 3-, 4-, 5-gram, Backoff, and PCFG. These scores represent the estimated number of guesses needed to guess the password. The results show that Lithnet performs the best in all scores, with OPF following quite close after, and Entra having the worst scores, and scoring significantly lower in 2-gram and 3-gram.

	Accepted Passwords							Rejected Passwords						
Password filter	**Number of passwords**	**Mean 2-gram**	**Mean 3-gram**	**Mean 4-gram**	**Mean 5-gram**	**Mean Backoff**	**Mean PCFG**	**Number of passwords**	**Mean 2-gram**	**Mean 3-gram**	**Mean 4-gram**	**Mean 5-gram**	**Mean Backoff**	**Mean PCFG**
Lithnet	12,037,408	2.36+67	1.66+54	8.18e+35	2.70e+23	3.86e+19	6.81e+21	76,308,890	5.08e+65	6.82e+56	1.01e+37	1.32e+26	3.92e+16	1.05e+24
OpenPasswordFilter	15,720,334	1.85e+67	1.41e+54	7.80e+35	2.68e+23	3.23e+19	6.44e+21	72,625,555	5.01e+65	6.70e+56	9.89e+36	1.30e+26	3.87e+16	1.04e+24
Entra Password Protection	78,561,899	4.55e+66	6.24e+53	5.04e+35	2.16e+23	1.40e+19	5.02e+21	9,784,082	1.32e+66	7.58e+56	8.47e+36	1.06e+26	2.74e+16	7.93e+23

Table 5.6: Monte Carlo PSM strength score results

In Figure 5.11 the impact on length can be seen for three of the Monte Carlo PSM scorings: 5-Gram, Backoff, and PCFG. Some of these PSM scores are the first to not show any clear impact due to password length. We see that for the 5-Gram scorings, passwords with a length of less than 5 scores best in both Lithnet and OPF, whereas for all three filters passwords with a length above 10 characters scores better than passwords with a length between 5 and 10 characters.

For the Backoff scoring, we have the same overall pattern of passwords with a length above 10 characters scoring better than passwords with a length between 5 and 10 characters. Intrestingly, these two scores are for Backoff significantly worse than scores for passwords with a length of less than 5 characters. We think the reason for passwords with less than 5 characters having such a big impact on 5-gram and Backoff is due to the overwhelming number of Chinese passwords in this category.

Finally, for the password length impact on PCFG scorings we see that passwords with a length less than 5 characters are scoring the worst. The two other length categories are scoring almost equally.

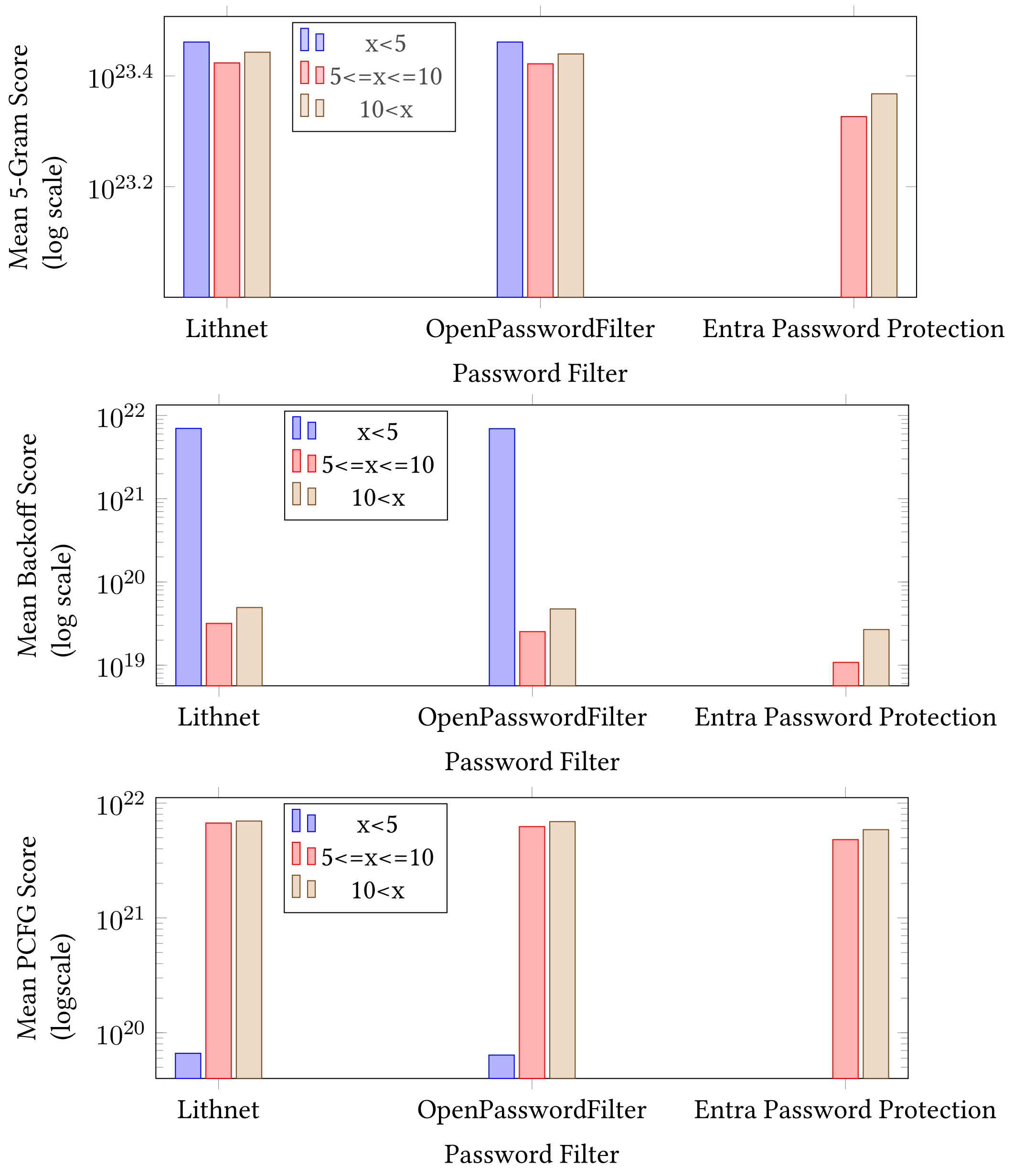


Figure 5.11: Column plot showcasing password length impact on 5-Gram, Backoff, and PCFG Monte Carlo PSM scores (log scale)

Figures 5.12, 5.13, and 5.14 shows the collective average distribution of all the Monte Carlo PSM scores. The distributions show that Lithnet and Entra is close in their distributions, with Lithnets slope towards higher scores being slightly less steep. We see that all three filters have their highest distribution of scores around 10^{14}, then sloping down towards higher scores. Intrestingly, both Lithnet and Entra have both have a spike at around a score of 10^{65}

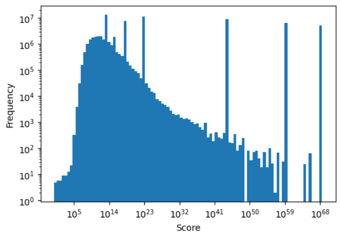


Figure 5.12: Histogram of MonteCarlo scores on the accepted passwords from Lithnet

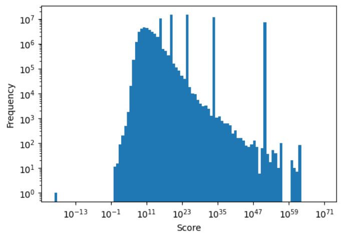


Figure 5.13: Histogram of MonteCarlo scores on the accepted passwords from OPF

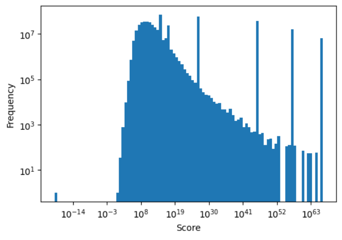


Figure 5.14: Histogram of MonteCarlo scores on the accepted passwords from Entra

5.2.4 fuzzyPSM

Table 5.7 shows the results of the fuzzyPSM. For this PSM lower scores is preferable. The results show that Entra has the best average score. We also see that OPF has the second best average score followed closely by Lithnet. This is the first PSM to score Entra the best and Lithnet the worst. However, it still scores OPF and Lithnet closely. For the rejected passwords, we can see that Lithnet actually scores the best, followed by OPF, and then Entra.

	Accepted Passwords		Rejected Passwords	
Password Filter	**Number of passwords**	**Mean Score**	**Number of passwords**	**Mean Score**
Lithnet	12,037,398	1.75e-05	76,308,850	1.15e-06
OpenPasswordFilter	15,720,334	1.31e-05	72,625,555	1.25e-06
Entra Password Protection	78,561,899	3.04e-06	9,784,072	6.03e-06

Table 5.7: fuzzyPSM strength score results

In Figure 5.15 password length impact on the fuzzyPSM scores can be seen. Again, for these scores a lower rating is better, and we see that passwords with lower character length also scores worse with this PSM. Intrestingly, we can see that for Entra, the score difference between passwords with lengths between 5 and 10 characters, and those with lengths above 10 characters is not very significant. Whereas, the difference is more impactful for Lithnet and OPF. The results also shows the likely reason for Entra to be scoring the best for this PSM is due to Entra having no passwords with a length of less than 5 characters, and these are clearly the most impactful on the overall scoring.

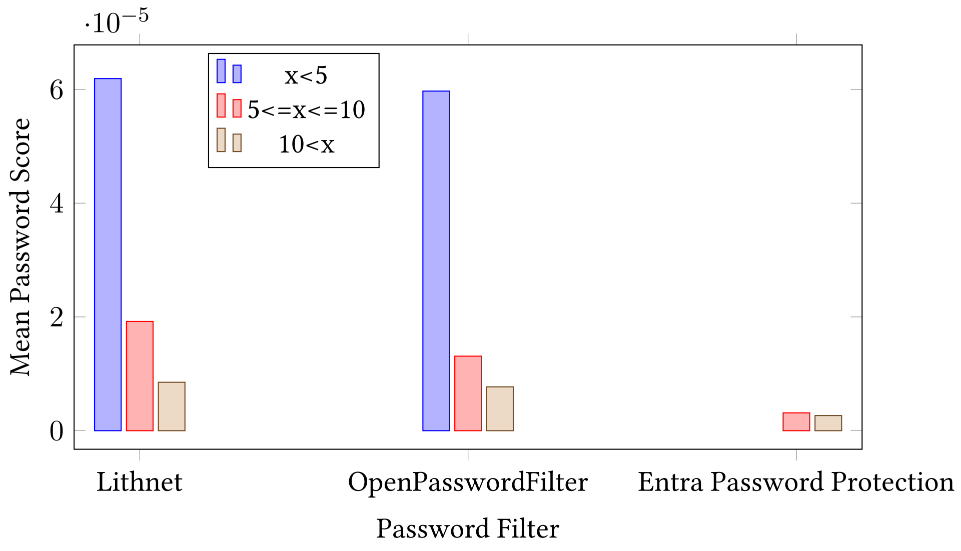


Figure 5.15: Column plot showcasing password length impact on fuzzyPSM score

The fuzzyPSM score distributions can be seen in Figures 5.16, 5.17, and 5.18. The distributions shows that a large number of very low scores are present for all three filters. However, Entra's distribution shows that a number of bins are quite a bit higher on the y-axis than the two other filters. These bins are also on the lower end of the scores, which is likely why Entra scores the best with fuzzyPSM.

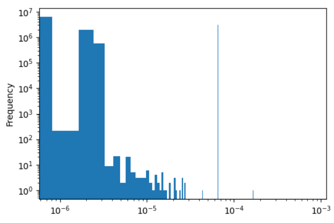


Figure 5.16: Histogram of fuzzyPSM scores on the accepted passwords from Lithnet.

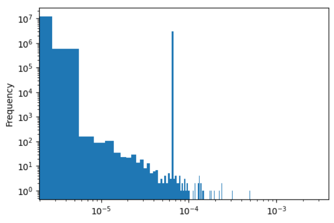


Figure 5.17: Histogram of fuzzyPSM scores on the accepted passwords from OPF.

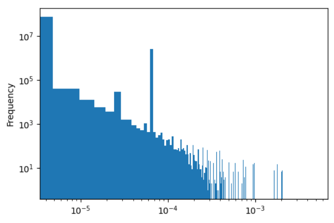


Figure 5.18: Histogram of fuzzyPSM scores on the accepted passwords from Entra.

5.3 Guessing Attacks

5.3.1 Online Attack on Accepted Passwords

This section contains the results from the online guessing attacks. The method is for online guessing attacks is described in Section 4.6.2.2. The attack was conducted on the passwords accepted by the filters.

	Lithnet			OPF			Entra		
Threshold	Seclist	Nordpass	Wikipedia	Seclist	Nordpass	Wikipedia	Seclist	Nordpass	Wikipedia
50	0	0	0	0	0	0	0	0	0
100	0	0	0	0	0	0	1	0	1
200	0	0	0	0	0	0	11	7	11
10,000	0	N/A	0	0	N/A	0	4,748	N/A	4748
54,720	46	N/A	N/A	81	N/A	N/A	32,839	N/A	N/A

Table 5.8: Number of guessed passwords with different thresholds for accepted passwords.

Looking at the results from Table 5.8 we see that no filter has any successful attacks at a threshold of 50, and that only Entra has any successful attacks at thresholds 100, 200, 10,000. This shows that the passwords accepted by Entra is by far the most susceptible to an online guessing attacks, and that Lithnet and OPF shows great resistance to online guessing attacks only producing successful guesses at a very high threshold of 54,720.

5.3.2 Offline Guessing Attacks on Accepted Passwords

This section presents the results of the offline guessing attacks against each filter, following the method presented in section 4.6.2.1. The attacks were conducted using two Nvidia A40 GPUs on Aalborg University's CLAAUDIA [57].

Table 5.9 presents the results of all the offline guessing attacks against the three filters. The table shows that even though Lithnet and OPF has significantly less total accepted passwords, the number of unguessed passwords are very high, especially when compared to the results of the Entra filter.

Filter	Total accepted passwords	Brute Force	Dictionary	Dictionary + Rules	PassGAN + Rules	Unguessed
Lithnet	12,037,408	1,651,767	30,780	1,078,945	1,021,039	8,254,877
OPF	15,720,396	2,062,157	25,293	3,127,743	1,609,823	8,895,380
Entra	78,561,900	17,716,342	8,449,032	24,735,292	3,095,713	24,565,521

Table 5.9: Amount of passwords cracked for each attack on all accepted passwords for each filter

Table 5.10 shows the results in percentages, and here we see that Entras passwords are the most vulnerable to offline guessing attacks, with a high overall percentage guessed (68.73%). Especially the brute force attack (22.55%) and dictionary + rules (31.48%) indicate that Entra fails significantly in blocking weak passwords. Entra also has the highest percentage of guessed passwords from the dictionary attack (10.75%), this is a simple attack, and this result highlights the failure to effectively block easily guessed passwords.

Lithnet, shows that it is resistant to most offline guessing attacks, indicating that it succeeds in blocking weak and easily guessable passwords.

OPF shows the same resistance as Lithnet in the simpler guessing attacks i.e. brute force and dictionary. However, OPF does seem to be more vulnerable to more advanced guessing attacks strategies with dictionary + rules (19.89%) being especially effective.

Filter	Brute Force	Dictionary	Dictionary + Rules	PassGAN + Rules	Total
Lithnet	13.72%	0.25%	8.96%	8.48%	31.42%
OPF	13.12%	0.16%	19.89%	10.24%	43.42%
Entra	22.55%	10.75%	31.48%	3.94%	68.73%

Table 5.10: Percentage of passwords guessed for each attack on all accepted passwords for each filter.

In Figure 5.19 we see the percentage of passwords guessed for each length category for accepted passwords. Entra again shows high percentage of guessed passwords for passwords between 5 and 10 characters long and even passwords above 10 characters long. This illustrates that weak passwords are allowed through the Entra filter. When compared to Lithnet and OPF, it shows the impact an effective filter has on password security. The low percentage of guessed passwords with a length below 5 characters for Lithnet and OPF is likely caused by the large amount of Chinese character passwords that dominate the category.

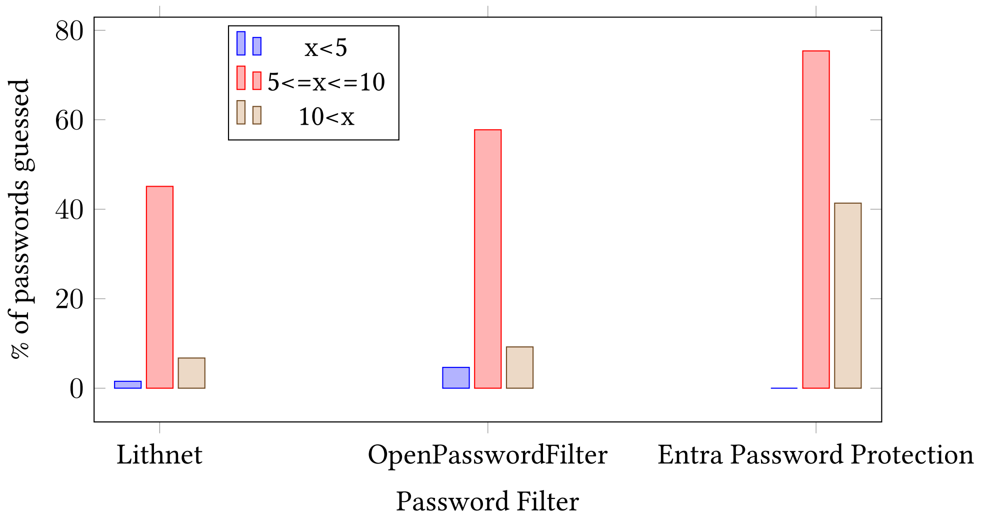


Figure 5.19: Column plot showcasing the percentage of password guessed for each length category, for accepted passwords.

5.3.3 Offline Guessing Attacks on Rejected Passwords

This section presents the results of the offline guessing attacks on the rejected passwords for each filter, following the method presented in Section 4.6.2.1. The attacks were conducted using two Nvidia A40 GPUs on Aalborg University's CLAAUDIA [57].

Table 5.11 shows the results from the rejected offline guessing attacks for all the filters. The results show that Lithnet and OPF has almost the same amount of passwords not guessed, and that Entra has the least amount of passwords not guessed. However for these results, it is easier to get an overview with percentages, which is done in Table 5.12.

Filter	Total rejected passwords	Brute Force	Dictionary	Dictionary + Rules	PassGAN + Rules	Unguessed
Lithnet	76,308,850	18,763,411	9,627,030	26,919,671	2,136,566	18,862,172
OPF	72,625,555	18,353,297	9,632,614	24,871,083	1,547,781	18,220,780
Entra	9,784,072	2,698,750	1,209,159	3,263,348	61,892	2,550,923

Table 5.11: Amount of passwords guessed for each attack on all rejected passwords for each filter

Table 5.12 indicate that rejected passwords for all filters are highly vulnerable to offline guessing attacks, with Lithnet showing the highest total percentage of guessed passwords (75.28%), followed by OPF (74.91%) and Entra (73.92%). This higlights that the filters indeed are effective in preventing the use of weak passwords. A key observation is the significant difference in the percentages of guessed passwords between accepted passwords seen in Table 5.10, and rejected passwords seen in Table 5.12. Lithnet and OPF exhibit a 30-40% difference, whereas Entra shows only a 5.19% difference. Suggesting that Entra fails to effectively block weak passwords, and that the rejected passwords are not substantially easier to guess than the accepted passwords.

Filter	Brute Force	Dictionary	Dictionary + Rules	PassGAN + Rules	Total
Lithnet	24.59%	12.62%	35.28%	2.80%	75.28%
OPF	25.27%	13.26%	34.25%	2.13%	74.91%
Entra	27.58%	12.36%	33.35%	0.63%	73.92%

Table 5.12: Percentage of passwords guessed for each attack on all rejected passwords for each filter

In Figure 5.20 we see the percentage of rejected passwords guessed for each length category. The high guess percentages across all length categories for Lithnet and OPF indicate that most weak passwords were effectively filtered out. Entra has high guess percentages but when compared to Figure 5.19, only a small difference between the rejected and accepted passwords is to be found, indicating failure to effectively filter weak passwords.

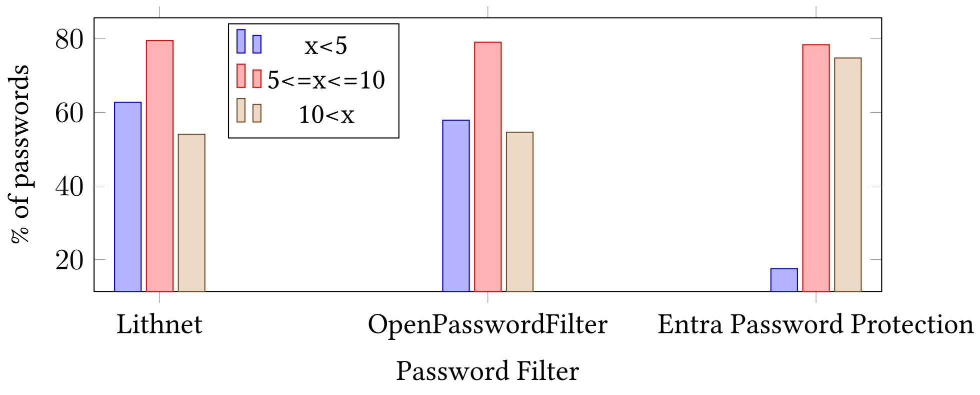


Figure 5.20: Column plot showing the percentage of rejected password guessed for each length category.

5.4 Combined Results

In this section we use the normalization method described in Section 4.7 to normalize the results from the different PSMs and guessing attacks. We then use these normalized values to calculate a final score for all three Active Directory password filters.

5.4.1 Normalized Password Strength Meter Scores

5.4.1.1 zxcvbn PSM

We use Lithnets, OPFs, and Entras average zxcvbn score from 5.4:

$$\text{Lithnet zxcvbn} = 2.667615708976309$$

$$\text{Normalized Score} = \frac{2.667615708976309 - 0}{4 - 0} = 0.666903927244$$

$$\text{Log Normalized Score} = \frac{log(2.667615708976309)}{log(4)} = 0.707775424962$$

$$\text{OpenPasswordFilter zxcvbn} = 2.5606279103230247$$

$$\text{Normalized Score} = \frac{2.5606279103230247 - 0}{4 - 0} = 0.640156977581$$

$$\text{Log Normalized Score} = \frac{log(2.5606279103230247)}{log(4)} = 0.678248813712$$

$$\text{Entra zxcvbn} = 2.2247675555806$$

$$\text{Normalized Score} = \frac{2.2247675555806 - 0}{4 - 0} = 0.556191888895$$

$$\text{Log Normalized Score} = \frac{log(2.2247675555806)}{log(4)} = 0.576827305359$$

In Table 5.13, all the normalized scores for zxcvbn can be seen. For both normalized and log normalized, Lithnet has the highest score followed by OPF and then Entra. It can be seen that the log normalization increases both Lithnet and OPF scores more than Entras score.

Filter	Normalized Score	Log Normalized Score
Lithnet	0.666903927244	0.707775424962
OpenPasswordFilter	0.640156977581	0.678248813712
Entra	0.556191888895	0.576827305359

Table 5.13: zxcvbn Normalized Scores

5.4.1.2 Monte Carlo PSM

Here we use the Lithnet, OPF, and Entra Monte Carlo PSM scores from Table 5.6. Since the Monte Carlo PSM returns 6 results, we have taken the average of all the scores, to calculate the normalized scores.

$$\text{Monte Carlo PSM Lithnet} = 3.94e + 66$$
$$\text{Normalized Score} = \frac{3.94e + 66}{5.42e + 67} = 0.072693726937269$$
$$\text{Log Normalized Score} = \frac{log(3.94e + 66)}{log(5.42e + 67)} = 0.983191557021510$$

$$\text{Monte Carlo PSM OpenPasswordFilter} = 3.09e + 66$$
$$\text{Normalized Score} = \frac{3.09e + 66}{5.42e + 67} = 0.057011070110701$$
$$\text{Log Normalized Score} = \frac{log(3.09e + 66)}{log(5.42e + 67)} = 0.981633436380290$$

$$\text{Monte Carlo PSM Entra} = 7.59e + 65$$
$$\text{Normalized Score} = \frac{7.59e + 65}{5.42e + 67} = 0.014003690036900$$
$$\text{Log Normalized Score} = \frac{log(7.59e + 65)}{log(5.42e + 67)} = 0.972631801899060$$

In Table 5.14 the normalized Monte Carlo PSM scores for all three filters can be seen. These scores shows a significant difference between the normalized score and log normalized score. Since for both of the normalized scores a higher score is better, it is clear that the log normalized score will have a much bigger impact than the normalized score. This is due to the huge numbers coming from the Monte Carlo PSM and how the logarithmic function works. While the log normalized score will have a higher impact, there is not a big difference between the three filters scores, so the impact will be equal for all filters.

Filter	**Normalized Score**	**Log Normalized Score**
Lithnet	0.072693726937269	0.983191557021510
OpenPasswordFilter	0.057011070110701	0.981633436380290
Entra	0.014003690036900	0.972631801899060

Table 5.14: Monte Carlo PSM Normalized Scores

5.4.1.3 MultiPSM

For the normalized MultiPSM scores, we use the Fusion Score results from Table 5.5, for all three filters.

$$\text{MultiPSM Lithnet} = 3.73$$

$$\text{Normalized Score} = \frac{3.73 - 0}{10 - 0} = 0.373$$

$$\text{Log Normalized Score} = \frac{log(3.73)}{log(10)} = 0.571708831809$$

$$\text{MultiPSM OpenPasswordFilter} = 3.48$$

$$\text{Normalized Score} = \frac{3.48 - 0}{10 - 0} = 0.348$$

$$\text{Log Normalized Score} = \frac{log(3.48)}{log(10)} = 0.541579243947$$

$$\text{MultiPSM OpenPasswordFilter} = 2.83$$

$$\text{Normalized Score} = \frac{2.83 - 0}{10 - 0} = 0.283$$

$$\text{Log Normalized Score} = \frac{log(2.83)}{log(10)} = 0.451786435524$$

The MultiPSM Fusion Scores were in a value range of $0 - 10$, the normalized score is simply just the Fusion Score, where the decimal has moved, giving us the score range of $0 - 1$. In Table 5.15 we can see the log normalized scores are a bit higher than the normalized scores, but the overall relationship between the three filters scores remain the same, with Lithnet having the highest score, followed by OPF and then Entra.

Filter	Normalized Score	Log Normalized Score
Lithnet	0.373	0.571708831809
OpenPasswordFilter	0.348	0.541579243947
Entra	0.283	0.451786435524

Table 5.15: MultiPSM Normalized Scores

5.4.1.4 fuzzyPSM

Since the fuzzyPSM score is in percentages, it is already in the range that we are normalizing to, therefore we only need to calculate the logarithmic normalized scores. However, fuzzyPSM scores a lower percentage as good score, and a higher percentage as bad score. This would result in a good fuzzyPSM score having a negative impact on the overall score, which is not desired. To avoid this we subtract the score from 1. This is possible because the score is already in the correct range of $0 - 1$.

Furthermore, to avoid dividing by 0, we multiply the scores by 100 in the logarithmic normalized score. This approach maintains the percentage score while preventing log(1) from being the maximum score dividend, thereby avoiding division by zero.

$$\text{fuzzyPSM Lithnet} = 0.0000175$$

$$\text{Normalized score} = 1 - 0.0000175 = 0.9999825$$

$$\text{Log Normalized Score} = \frac{log(100 - (0.00175 * 10))}{log(100)} = 0.99999619989$$

$$\text{fuzzyPSM OpenPasswordFilter} = 0.0000131$$

$$\text{Normalized score} = 1 - 0.0000131 = 0.9999869$$

$$\text{Log Normalized Score} = \frac{log((100 - (0.0000131 * 100)))}{log(100)} = 0.999997155353$$

$$\text{fuzzyPSM Entra} = 0.00000304$$

$$\text{Normalized score} = 1 - 0.00000304 = 0.99999696$$

$$\text{Log Normalized Score} = \frac{log(100 - (0.000304 * 100))}{log(100)} = 0.99999933987$$

In Table 5.16 the final normalized fuzzyPSM scores can be seen. Due to the original scores from fuzzyPSM, in both the normalized scores and log normalized scores we end up with near identical scores for all three filters. And as the table shows, these scores are all very high, being at least 0.99 in a range of $0 - 1$. This means that the impact of fuzzyPSMs scores will only be positive for all three filters.

Filter	Normalized Score	Log Normalized Score
Lithnet	0.9999825	0.99999619989
OpenPasswordFilter	0.9999869	0.99999933987
Entra	0.99999696	0.999997155353

Table 5.16: fuzzyPSM Normalized Scores

5.4.2 Normalized Guessing Attack Scores

5.4.2.1 Normalized Online Guessing Attack Scores

For the score calculations for online guessing attacks, we take the average successful guesses for all thresholds and calculate the combined average guess percentage as seen in Table 5.17. We then use the combined average percentage along with the normalization formulas described in Section 4.7 to calculate the score.

	Lithnet		OPF		Entra	
Threshold	**Average guess**	**Average guess %**	**Average guess**	**Average guess %**	**Average guess**	**Average guess %**
50	0	0%	0	0%	0	0
100	0	0%	0	0%	0.667	0.667%
200	0	0%	0	0%	9.667	4.833%
10000	0	0%	0	0%	4748	47.48%
54720	46	0.084%	81	0.148	32839	60.013%
Combined average %	0.016%		0.029%		22.599%	

Table 5.17: Table showing the average amount of successful guesses for each threshold for online guessing attacks. And the total average successful guess percentage across all three filters.

Looking at the scores in Table 5.18 we see the resilience of Lithnet and OPF has to online guessing attacks. Entra falls behind with a lower score reflecting its inability to effectively block weak passwords. It can also be seen that the log normalized score from Entra is significantly closer to the scores of Lithnet and OPF compared to the difference seen between Entras normalized score and the normalized scores of Lithnet and OPF.

Filters	Normalized Score	Log Score
Lithnet	0.9998318713	0.9999634883
OPF	0.9997039474	0.9999357035
Entra	0.7740144152	0.9443745245

Table 5.18: Normalized online guessing attack scores for each of the password filters.

5.4.2.2 Normalized Offline Guessing Attack Scores

For the score calculations we use the the formulas described in Section 4.7 with the percentages from the offline guessing attack results from Table 5.10. An example of the normalization calculations for the brute force attack are shown here, for the rest of the calculations refer to appendix A.2.

$$\text{Entra Brute Force} = 22.55\%$$

$$\text{Normalized Score} = \frac{(100 - 22.55) - 0}{100 - 0} = 0.7745$$

$$\text{Log Normalized Score} = \frac{log(100 - 22.55)}{log(100)} = 0.94450845$$

Looking at Table 5.19, Lithnet performs the best in the Dictionary+Rules and PassGAN attacks. Whereas, OPF performs the best in Brute Force and Dictionary attacks. Lithnets scores suggest a well-balanced filter with high resistance across various attack methods. OPF scores also suggest a well-balanced filter, that is on par with Lithnet, but is slightly worse at protecting against the more sophisticated attacks (dictionary + rules and PassGAN). Entra is by far the weakest filter, with the lowest scores in all the attacks but PassGAN, where it surprisingly performs the best.

	Brute force		Dictionary		Dictionary + Rules		PassGAN	
Filter	Normalized Score	Log Score	Normalized Score	Log Score	Normalized Score	Log Score	Normalized Score	Log Score
Lithnet	0.8627805089	0.9679501627	0.9974429711	0.9994440371	0.9103673316	0.9796083324	0.9151778356	0.9807527468
OPF	0.8688228337	0.9694656131	0.9983910711	0.9996503442	0.8010391723	0.9518268772	0.8975965364	0.976540584
Entra	0.7744919357	0.94450845	0.892453823	0.9752928771	0.6851490099	0.9178925172	0.9605952376	0.9912702146

Table 5.19: Normalized scores for offline guessing attacks

5.4.3 Final Scores

Taking the results of the normalized scores, we can calculate the final scores for the filters, that determine their strength.

5.4.3.1 Average Score

The average score is calculated using the all the scores from Section 5.4.1 and Section 5.4.2.

Looking at the final average scores seen in Table 5.20 Lithnet scores the best with OPF right behind. Entra scores the worst as expected considering the scores from Section 5.4.1 and 5.4.2.

Filter	Average Score	Log Scaled Score
Lithnet	0.7553534081	0.9098702956
OPF	0.7345231676	0.8998755507
Entra	0.6599885511	0.8638423646

Table 5.20: Table of final unweighted scores for Lithnet, OPF and Entra

5.4.3.2 Weighted Score

The weights for the weighted score calculations can be seen in Table 5.21 and Figure 5.21. The heavily weighted scores are the zxcvbn and MultiPSM scores as well as the dictionary and brute force attacks scores. The brute force and dictionary attacks is weighted higher due to the simplicity of the attacks, and a low score in either would indicate that a filter has failed in blocking weak passwords. The PassGAN attack is weighted lower due to the high barrier of entry in terms of carrying out the attack. The dictionary+rules attack is also weighted lower considering the complexity of the attack compared to both the brute force and dictionary attacks.
We have chosen to give higher weights to both the zxcvbn PSM and MultiPSM. This is due to how well the two PSMs performed, and in part due to how both Monte Carlo PSM and

fuzzyPSM performed. The Monte Carlo PSM is weighted lower, due its results not being very precise, the PSM is supposed to give a estimated guess number for the passwords. The results seen from the Monte Carlo PSM, are too high compared to the results we have seen with our own guessing attacks. fuzzyPSM is weighted lower for two main reasons, first the results for all three filters are too close, thereby not giving any real impact. The second reason is that we don't know what training data was actually used for the PSM, which makes the results a bit less transparent.

Base weight	zxcvbn	MultiPSM	Monte Carlo PSM	fuzzyPSM	Online	Brute Force	Dictionary	Dictionary+Rules	PassGAN
0.1111111111	0.1851851852	0.1851851852	0.03703703704	0.03703703704	0.1111111111	0.1666666667	0.1666666667	0.05555555556	0.05555555556

Table 5.21: Weights for the various scores for the Weighted Sum calculation

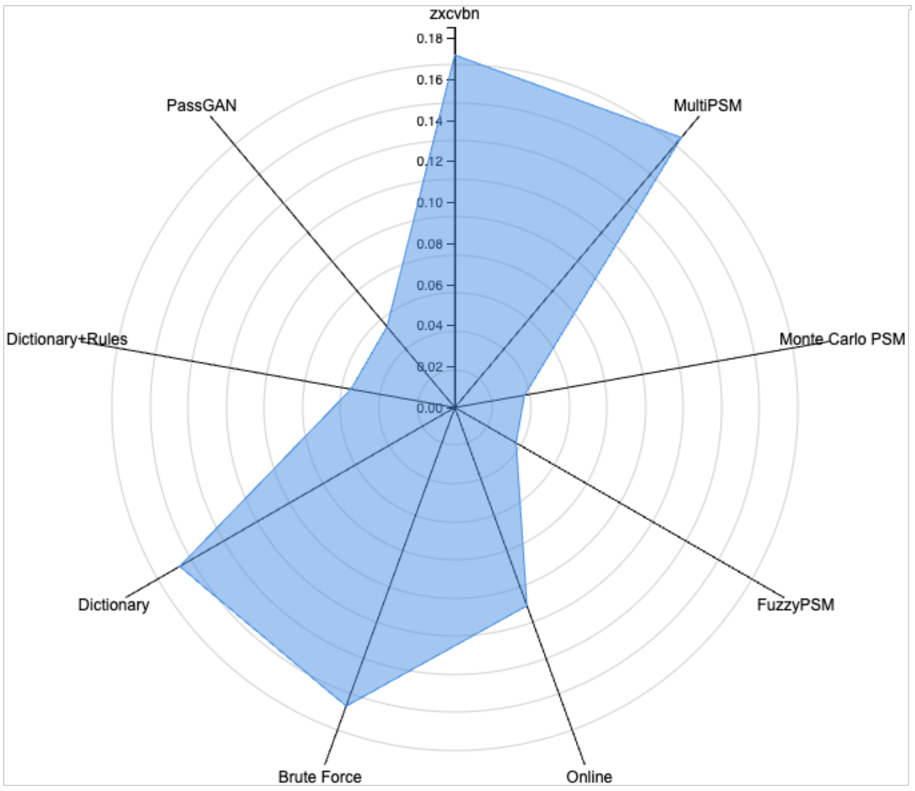


Figure 5.21: Radar plot of the weights used for the weighted score calculations

We use the weighted sum formula from Chapter 4, Section 4.7 to calculate the weighted score, using the weights from table 5.21:

Filter	Average Score	Log Scaled Score
Lithnet	0.7548524033	0.8582502557
OPF	0.7387892902	0.8457093768
Entra	0.6482176158	0.7945063446

Table 5.22: Table of final weighted scores for Lithnet, OPF and Entra

Using the weights we see a drop in the overall scores for all filters, Lithnet and OPF still scores the best with Entra scoring the worst and being affected by the weights the most.

Chapter 6

Discussion

6.1 Evaluation

Looking at the results in Table 5.1 from Section 5.1.1 it is clear that Lithnet is the most restrictive filter, accepting the least amount of passwords, only accepting 13.62% of the overall passwords passed through the filters. It is also evident that the passwords Lithnet does accept are strong with the lowest guess percentages from guessing attacks and highest scores from the PSMs. Lithnet also achieves the highest final score both weighted and unweighted of the three filters evaluated in this thesis.

OPF produces very similar results to Lithnet, it has a slightly higher percentage of accepted passwords at 17.7%. This difference compared Lithnet stems from OPF generally accepting more passwords from each list, especially from the PassGAN generated passwords and the Linkedin dataset. This is likely the cause of the slightly worse scores for most of the PSMs and guessing attacks, though we would argue that the difference between OPF and Lithnet is negligible. While this only takes our method of calculating the filters strength into account, it must be clarified that the performance difference between Lithnet and OPF is not negilably, as seen in Table 4.5.

Entra accepts by far the largest amount of passwords, accepting a total of 88.9% of the overall passwords passed through the filter. This makes Entra the only filter evaluated in this thesis that has a majority of accepted passwords. Entra also scores the worst in the final scores as seen in Section 5.4.3, and consistently scores the worst, in zxcvbn PSM, MultiPSM, Monte Carlo PSM, offline guessing attacks, and online guessing attack. This by our evaluation means that Entra is a worse Active Directory password filter compared to both Lithnet and OpenPasswordFilter.

Looking at the results from Chapter 5, we see that Entra did not accept any password with a character length less than five. Interestingly this is a feature of the filter that is not specifically mentioned in the documentation. It is also the only one of the three filters evaluated in this thesis to do this. Entra while scoring the lowest and performing the worst, does block significantly more of the Chinese dataset compared to both Lithnet and OpenPasswordFilter. Hinting at Entra being better suited for non-western languages, at least when using Lithnets and OpenPasswordFilters default setups.

A noteworthy observation regarding the results of the Lithnet password filter is that, despite utilizing HaveIBeenPwned [21], it failed to block a significant number of passwords from the leaked lists, such as the LinkedIn list. Additionally, it did not block 13.5% of the passwords from the Xato-Net list of commonly used passwords. Passwords that we would expect to exist within the HaveIBeenPwned database [21].

When selecting a dictionary for offline guessing attacks, for the dictionary and dictionary+rules attacks, our initial strategy was to use a password dictionary that contained passwords not present in any of our password lists from Table 4.1. This led us to initially dismiss the Crackstation password compilation dictionary, as we were confident it included many of the passwords we used to test the filters. However, after further discussion, we realized that excluding it would introduce bias and fail to accurately represent the filters effectiveness. If the filters are not effectively blocking insecure passwords found in commonly used dictionaries, this issue would not be revealed if we used a dictionary that didn't include any of the passwords used to test the filters.

When it comes to the online guessing attacks it is clear that Entra is significantly more vulnerable than the other two filters. This is interesting as it shows Entra's inability to block a large number of commonly used passwords, which arguably is the most important function of a blocklist and filter. It shows that even with a high lockout restriction an attacker could potentially get access within a relatively short amount of time. While we have not configured the custom blocklist for Entra, it is only limited to 1,000 different words.

There is one aspect of the filters that has not been considered for this thesis and that is the effect the filters have on usability. Much of the literature mentions that configuring blocklists is a challenge and wrong implementations has a significant effect on usability [26] [55]. It is clear from the results, that the various filters will have different impacts on usability due to the different number of accepted passwords. We think it would be interesting to research what exactly this impact is. To conduct this research, the user study from the works of Tan et al. [55] could be followed. The reason this aspect has been left out of the thesis is for scoping reasons.

To get all the results from MultiPSM on the rejected passwords in time for the deadline of the thesis, a smaller random selection of rejected passwords from OPF was used to calculate the score of the rejected passwords. This was done due to the speed of MultiPSM, even when running more than 35 instances of MultiPSM on two different desktop computers it would take up towards a week to calculate roughly 70 millon passwords scores. Calculating scores on a random selection of passwords potentially has an effect on the score of the rejected passwords from OPF. As we can see in Table 5.5 in Section 5.4.1, the rejected passwords from OPF score closer to Entra, where one would expect it to be closer to Lithnet when compared to the other PSM results.

Using the Monte Carlo PSM [8] created by Dell'Amico et al. [9] we followed the GitHub page instructions for training the PSM. Nevertheless, once we started using the PSM we saw that the results where quite high. The PSM is supposed to estimate the number of guesses needed to guess a given password. But the results returned by this PSM did just not match the number of guesses actually needed. While we think there could be multiple reasons behind this issue, we have two main theories. First, is that the training data used is just not sufficient enough, even though it is documented on the GitHub page. The second theory could be that the PSM might be outdated by today's standards, the paper was published in 2015, and password security has changed a lot since then.

6.2 Challenges and Obstacles

Our method described in Chapter 4 use a number of Password Strength Meters to help calculate the strength of the password filters. Some of these PSMs are built on models that are trained with password data. Therefore, it is likely that some of the PSMs are trained on the same data that we have used in our method, which potentially has an impact on our methods results. The impact is not necessarily negative. It can be argued that while there might be a password overlap between our method and the training data for the PSMs, this data was chosen because these passwords are commonly used in the real world. By excluding data in our method due to an overlap, we could introduce bias and be negatively impacting our method and results. An example of the password overlap in our method and PSMs, is fuzzyPSM by Wang et al. [61] and MultiPSM by Galbally et al. [16] who both use the RockYou dataset, which we also have used to test the filters.

As showcased in Chapter 4, we are using two different scripts for automatically changing passwords in our Active Directory environments. While we originally intended to only use a single script. We quickly realized that due to the number of passwords we wanted to run through the password filters, the original simple script would not suffice, with the speeds we were seeing. We therefore developed a script that utilized parallel programming, as to improve the speed. Unfortunately, we realized that the filters themselves had quite the impact on speed. From our tests, we saw that the Lithnet filter improved significantly from the parallel script, but the two other filters did not. Due to the time-limit of the thesis we therefore decided to use both scripts, and get our results faster. The parallel script does come with a downside of splitting the passwords lists. In our case on the Lithnet filter, this means that up to 30 of the passwords from each list potentially has been split randomly. This will have an impact on the results, however we valued the time gain over the minimal interference of up to a total of 360 passwords out of 88,346,298 passwords.

The time limit on the thesis and speed of the filters, also significantly impacted the amount of password we in total got to run through the filters. While we did manage to use 12 passwords lists with a total of almost 90 million password for each filter, we initially wanted to do much more extensive experiments. The works of Nisenoff et al. [41] showcased a bootstrapped collection of passwords, that was found by searching in more than 450 service breaches and 12 breach compilations. We initially wanted to use their methodology and also run a much larger number of passwords through the filters, we had more password lists, including password compilations with hundreds of millions of passwords. However, due to the speed and time factor, we had to significantly reduce our total number of passwords. We therefore, opted to pick some of the more commonly used lists from academic works such as the LinkedIn and RockYou dataset. We also specifically chose to work with deep learning generated passwords. Due the nature of generated passwords we can be almost certain that the list we generate will contain passwords not present in any online or public list. Furthermore, we chose to follow Wang et al. [62] and also include passwords of differing origins, including passwords that use characters that is non-ASCII. The inclusion of non-ASCII password lists, highlighted some interesting results. It is clear from both our PSM results and guessing attacks, that non-ASCII passwords have a huge effect. Interestingly, we noticed that the biggest impact here came from the Chinese passwords, which had a much bigger impact than for example Cyrillic passwords or other non-ASCII passwords we found in the RockYou dataset.

Our thesis proposes a method for calculating the strength of Active Directory password filters, and we showcase this method against 3 filters. We originally wanted to evaluate four filters the fourth being the Improsec password filter [22] but we eventually dropped it. This is due to the speed of the Improsec filter simply being too slow as well as taking up too many resources. Running the filter in one of our VMs we had to allocate 22GB RAM to the VM, for the filter to run. These two factors required us to drop the Improsec filter. While not impossible to run all the passwords through the filter, we calculated that with the speed of the filter it would take around 2000 hours. And since the filter was very resource intensive, we could not simply split the task out on multiple VMs.

Two of the three filters we have tested, are open source and the third is commercial filter developed by Microsoft. We also would have liked to include more commercial filters, simply due to the difference in nature between closed commercial projects and open source projects. As we can see with the Microsoft Entra filter, the blocklist is kept secret and while score calculation for the filters backend is known due to Microsoft's documentation [33], there is still no way for an end user to know exactly what will be blocked and how secure the filter is. There exists many more commercial filters, that would have proven interesting to test in our thesis but due to costs and time it was not possible to include them.

As seen in our literature review, a lot of research into password strength uses Ur et al. [58] Password Guessability Service. After conducting our literature review, we also wanted to incorporate this into our thesis. We had two main thoughts for how we could use the service for the thesis. First, using the service for the final filter score estimation and calculation besides our own password guessing attacks. Furthermore, it could potentially have been used as a accuracy ideal. Meaning that our own results from guessing attacks could have been compared to the results from the service. Unfortunately, we never got access to the Password Guessabiltiy Service. We requested the access twice, following the guide to request access from their own documentation [5].

When running the PSM from Galbally et al. [15] we found that the executable often crashed when running. The crash would happen if the password evaluated used non-ASCII characters, or when the model used for the PSM encountered a password it could not handle. To fix this issue we added exception handling to the code of the PSM so that it still continued to evaluate passwords skipping the ones that caused it to crash. This came into effect when running passwords that contained Cyrillic, æøå, or Chinese characters with the PSM completely unable to parse them. The impact of this can be seen in Chapter 5, specifically in Table 5.5 where a lower amount of passwords are tested. The majority of the passwords missing comes from the Russian and Chinese password lists.

Additionally, the speed of this password strength meter was found to be quite slow compared to the other password strength meters used in our thesis. Fortunately, we could somewhat solve the speed issue of the PSM by simply running multiple instances of the PSM, where we at times had 30 instances of the PSM running. While 30 instances definitely sped up the run time significantly, we would have liked to have run even more instances, but due to the program being quite resource intensive, we were only able to run 15 instances on a machine with 64GB of RAM, of which we had two.

When it became evident that we would need the computing power of CLAAUDIA [57] a HPC (High-Performance Computing) cloud service provided by Aalborg University for the password guessing attacks, we faced the challenge of installing either John the Ripper (JtR) or Hashcat on CLAAUDIA. Due to CLAAUDIA's limited functionality, we had to build a container with either JtR or Hashcat locally and then transfer it to CLAAUDIA. Initially, we preferred JtR, as Ji et al. [23] had found it to be more effective. However, we discovered that JtR was significantly more difficult to set up on CLAAUDIA and slower in terms of password cracking speed. Consequently, we opted for Hashcat, which not only was easier to set up but also offered additional functionality that benefited our thesis.

Chapter 7

Conclusion

This masters thesis presents a novel method combining Password Strength Meters (PSMs) and password guessing attacks to evaluate password filters for Microsoft Active Directory. The method is used to evaluate three different password filters from Lithnet [28], OpenPasswordFilter [54] and Microsoft Entra Password Protection [33]. We found that Lithnet and OpenPasswordFilter successfully prevents weak passwords from being used, while Entra falls behind proving to be less effective in filtering out weak passwords. The method successfully leverages state of the art PSMs to measure the strength of passwords, and several password guessing attacks to determine the effectiveness of the password filters. Our method captures these results in a final score, that ranks Lithnet as the best filter, followed by OpenPasswordFilter, and finally Microsoft Entra Password Protection.

This masters thesis also presents an extensive literature search and review into contemporary research of password policies, password strength, and password guessing attacks. Where we find that measuring password strength is a complicated and difficult endeavor. Password Polices should not rely on password complexity, and blocklists are efficient but are hard to effectively implement. We find and utilize modern techniques for password guessing attacks such as PassGAN [20] which utilizes machine learning in order to conduct advanced offline guessing attacks.

Chapter 8

Future Work

The method we have proposed and tested in this thesis have shown that it can successfully evaluate the effectiveness of Active Directory password filters. However, as mentioned throughout the thesis, the method is very time consuming for various reasons. One big pitfall is the need to evaluate millions of passwords in the Active Directory to gather the list of accepted passwords. For future work we would like to explore the possibilities of using our method on a significant lower number of passwords, that have been randomly sampled from carefully selected password lists. This could potentially improve the speed of our method to such an extent that it could be used in professional enterprise environments as a part of security auditing. We would want to test this by running multiple experiments, to see exactly how few passwords are needed to get good reliable results. This approach would enhance the method discussed in this report, making it more feasible for further development such as introducing new PSMs, password filters, and reducing the hardware demands associated with processing millions of passwords.

This could potentially lead to the development of a program that incorporates the techniques described in this report, to make the test and evaluation of a password filter fast and easy.

Additionally the method could include a baseline evaluation of the passwords one would run through the filters, to see how well the passwords score without filtering at all, which could give great insights to how effective the filters are.

Furthermore, we have successfully shown that it is possible to reveal at least part of the Microsoft Entra Password Protection's hidden default global banned list, that is responsible for blocking weak passwords. We would like to further explore the possibilities of exposing more of this list.

Bibliography

[1] MITRE ATTCK®. *Use Alternate Authentication Material: Pass the Hash.* `https://attack.mitre.org/techniques/T1550/002/`. 2023.

[2] beta6. *PassGAN.* `https://github.com/beta6/PassGAN`. 2023.

[3] Marina Sanusi Bohuk et al. “Gossamer: Securely Measuring Password-based Logins”. In: *31st USENIX Security Symposium (USENIX Security 22).* 2022, pp. 1867–1884.

[4] Leon Bošnjak and Boštjan Brumen. “Rejecting the death of passwords: Advice for the future”. In: *Computer Science and Information Systems* 16.1 (2019), pp. 313–332.

[5] Passwords Research Team at Carnegie Mellon University. *The Carnegie Mellon University Password Research Group’s Password Guessability Service.* `https://pgs.ece.cmu.edu/`.

[6] Crowdstrike. *What is a pass-the-hash attack?* 2023. URL: `https://www.crowdstrike.com/cybersecurity-101/pass-the-hash/`.

[7] Center for Cybersikkehed. *Password-sikkerhed.* 2023. URL: `https://www.cfcs.dk/da/forebyggelse/vejledninger/passwords/`.

[8] Matteo Dell’Amico. *montecarlopwd.* `https://github.com/matteodellamico/montecarlopwd`. 2017.

[9] Matteo Dell’Amico and Maurizio Filippone. “Monte Carlo strength evaluation: Fast and reliable password checking”. In: *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security.* 2015, pp. 158–169.

[10] dizcza. *dizcza/docker-hashcat.* `https://hub.docker.com/r/dizcza/docker-hashcat`. 2023.

[11] Qiying Dong et al. “RLS-PSM: a robust and accurate password strength meter based on reuse, Leet and separation”. In: *IEEE Transactions on Information Forensics and Security* 16 (2021), pp. 4988–5002.

[12] Brannon Dorsey. *PassGAN.* `https://github.com/brannondorsey/PassGAN`. 2017.

[13] Joshua Eckroth, Lannie Hough, and Hala ElAarag. “OneRuleToFindThem: Efficient Automated Generation of Password Cracking Rules”. In: *Journal of Computing Sciences in Colleges* 39.3 (2023), pp. 226–248.

[14] Enzoic. *Enzoic for Active Directory.* 2024. URL: `https://www.enzoic.com/active-directory-password-monitoring/`.

[15] Javier Galbally, Iwen Coisel, and Ignacio Sanchez. “A new multimodal approach for password strength estimation—Part I: Theory and algorithms”. In: *IEEE Transactions on Information Forensics and Security* 12.12 (2016), pp. 2829–2844.

[16] Javier Galbally, Iwen Coisel, and Ignacio Sanchez. “A new multimodal approach for password strength estimation—Part II: Experimental evaluation”. In: *IEEE Transactions on Information Forensics and Security* 12.12 (2017), pp. 2845–2860.

[17] Eva Gerlitz, Maximilian Häring, and Matthew Smith. “Please do not use!? _ or your license plate number: analyzing password policies in german companies”. In: *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*. 2021, pp. 17–36.

[18] Maximilian Golla and Markus Dürmuth. “On the accuracy of password strength meters”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 1567–1582.

[19] Hana Habib et al. “Password creation in the presence of blacklists”. In: *Proc. USEC* (2017), p. 50.

[20] Briland Hitaj et al. “Passgan: A deep learning approach for password guessing”. In: *Applied Cryptography and Network Security: 17th International Conference, ACNS 2019, Bogota, Colombia, June 5–7, 2019, Proceedings 17*. Springer. 2019, pp. 217–237.

[21] Troy Hunt. *haveibeenpwned*. 2024. URL: `https://haveibeenpwned.com/`.

[22] Improsec. *Improsec Password Filter*. 2021. URL: `https://github.com/improsec/ImprosecPasswordFilter`.

[23] Shouling Ji et al. “Zero-sum password cracking game: a large-scale empirical study on the crackability, correlation, and security of passwords”. In: *IEEE transactions on dependable and secure computing* 14.5 (2015), pp. 550–564.

[24] Saul Johnson et al. “Skeptic: Automatic, justified and privacy-preserving password composition policy selection”. In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 2020, pp. 101–115.

[25] JRC-PaStMe. *JRC-PaStMe - License*. `https://github.com/ec-jrc/jrcpastme`. 2019.

[26] Kevin Lee, Sten Sjöberg, and Arvind Narayanan. “Password policies of most top websites fail to follow best practices”. In: *Eighteenth Symposium on Usable Privacy and Security (SOUPS 2022)*. 2022, pp. 561–580.

[27] Kyungchan Lim et al. “Evaluating Password Composition Policy and Password Meters of Popular Websites”. In: *2023 IEEE Security and Privacy Workshops (SPW)*. IEEE. 2023, pp. 12–20.

[28] Lithnet. *Lithnet Password Protection*. 2023. URL: `https://docs.lithnet.io/password-protection`.

[29] LastPass by LogMeIn. *THE PASSWORD EXPOSÉ 8 truths about the threats–and opportunities–of employee passwords*. `https://www.lastpass.com/-/media/cd40d79ac0324dfa857b7942f0e0e080.pdf`. 2017.

[30] Microsoft. *Active Directory Domain Services Overview.* https://learn.microsoft.com/en-us/windows-server/identity/ad-ds/get-started/virtual-dc/active-directory-domain-services-overview. 2022.

[31] Microsoft. *ActiveDirectory.* URL: https://learn.microsoft.com/en-us/powershell/module/activedirectory/?view=windowsserver2022-ps.

[32] Microsoft. *Domain Controller Roles.* https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc786438(v=ws.10). 2014.

[33] Microsoft. *Eliminate bad passwords using Microsoft Entra Password Protection.* https://learn.microsoft.com/en-us/entra/identity/authentication/concept-password-ban-bad. 2021.

[34] Microsoft. *NTLM Overview.* https://learn.microsoft.com/en-us/windows-server/security/kerberos/ntlm-overview. 2023.

[35] Microsoft. *Password Filters.* https://learn.microsoft.com/en-us/windows/win32/secmgmt/password-filters. 2021.

[36] Microsoft. *Password Filters.* https://learn.microsoft.com/da-dk/windows/win32/secmgmt/password-filters?redirectedfrom=MSDN. 2021.

[37] Microsoft. *Password Policy.* https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc783512(v=ws.10). 2009.

[38] Microsoft. *Passwords must meet complexity requirements.* https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc786468(v=ws.10). 2012.

[39] Microsoft. *Passwords technical overview.* https://learn.microsoft.com/en-us/windows-server/security/kerberos/passwords-technical-overview. 2021.

[40] Robert Morris and Ken Thompson. "Password security: A case history". In: *Communications of the ACM* 22.11 (1979), pp. 594–597.

[41] Alexandra Nisenoff et al. "A {Two-Decade} Retrospective Analysis of a University's Vulnerability to Attacks Exploiting Reused Passwords". In: *32nd USENIX Security Symposium (USENIX Security 23).* 2023, pp. 5127–5144.

[42] NIST. *Digital Identity Guidelines.* 2023. URL: https://pages.nist.gov/800-63-3/sp800-63b.html.

[43] NIST. *Electronic Authentication Guideline.* 2004. URL: https://web.archive.org/web/20040712152833/http://csrc.nist.gov/publications/nistpubs/800-63/SP800-63v6_3_3.pdf.

[44] David Pereira, Joao F Ferreira, and Alexandra Mendes. "Evaluating the accuracy of password strength meters using off-the-shelf guessing attacks". In: *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW).* IEEE. 2020, pp. 237–242.

[45] Joseph Ryan Ries. *PassFiltEx.* 2023. URL: https://github.com/ryanries/PassFiltEx.

[46] safepass.me. *safepass.me*. 2024. URL: https://safepass.me/safepass-me/.

[47] safepass.me. *Specops Password Policy*. 2024. URL: https://specopssoft.com/product/specops-password-policy/.

[48] ScottyDoo. *ChangePassword ADSI method not working (constraint violation)*. 2023. URL: https://forums.powershell.org/t/changepassword-adsi-method-not-working-constraint-violation/21535.

[49] Sectona. *Passwords technical overview*. https://sectona.com/pam-101/authentication/active-directory-based-authentication/.

[50] Sean M Segreti et al. "Diversify to survive: Making passwords stronger with adaptive policies". In: *Thirteenth symposium on usable privacy and security (SOUPS 2017)*. 2017, pp. 1–12.

[51] Tobias Seitz et al. "Do differences in password policies prevent password reuse?" In: *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*. 2017, pp. 2056–2063.

[52] Claude Elwood Shannon. "A mathematical theory of communication". In: *The Bell system technical journal* 27.3 (1948), pp. 379–423.

[53] Beth Sheresh and Doug Sheresh. *Understanding directory services*. Sams Publishing, 2002.

[54] Josh Stone. *OpenPasswordFilter*. https://github.com/jephthai/OpenPasswordFilter. 2018.

[55] Joshua Tan et al. "Practical recommendations for stronger, more usable passwords combining minimum-strength, minimum-length, and blocklist requirements". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2020, pp. 1407–1426.

[56] Hugo Buff Typhlosaurus. *How can I split a text file using PowerShell?* 2012. URL: https://stackoverflow.com/questions/1001776/how-can-i-split-a-text-file-using-powershell.

[57] Aalborg University. *CLAAUDIA*. 2023. URL: https://www.researcher.aau.dk/contact/claaudia.

[58] Blase Ur et al. "Measuring {Real-World} Accuracies and Biases in Modeling Password Guessability". In: *24th USENIX Security Symposium (USENIX Security 15)*. 2015, pp. 463–481. URL: https://pgs.ece.cmu.edu/.

[59] Maxsud Usmonov. "Identification and Authentication". In: *Scienceweb academic papers collection* (2021).

[60] Ding Wang. *fuzzyPSM*. https://github.com/NKUSec/fuzzyPSM. 2019.

[61] Ding Wang et al. "fuzzyPSM: A new password strength meter using fuzzy probabilistic context-free grammars". In: *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2016, pp. 595–606.

[62] Ding Wang et al. “No single silver bullet: Measuring the accuracy of password strength meters”. In: *Proc. USENIX SEC 2023*. 2023, pp. 1–28.

[63] Daniel Lowe Wheeler. “zxcvbn:{Low-Budget} Password Strength Estimation”. In: *25th USENIX Security Symposium (USENIX Security 16)*. 2016, pp. 157–173.

[64] Claes Wohlin. “Guidelines for snowballing in systematic literature studies and a replication in software engineering”. In: *Proceedings of the 18th international conference on evaluation and assessment in software engineering*. 2014, pp. 1–10.

[65] *zxcvbn*. URL: `https://github.com/dropbox/zxcvbn?tab=readme-ov-file`.

[66] *zxcvbn*. 2021. URL: `https://github.com/dwolfhub/zxcvbn-python`.

Appendix A

Appendixes

A.1 Advanced Password Changing Script

```
$stopWatch = [System.Diagnostics.Stopwatch]::StartNew()

$from = "C:\Users\Administrator\Desktop\Passwords\linkedinclean.txt"
$rootName = "C:\Users\Administrator\Desktop\Passwords\Split\split"
$ext = "txt"

$limit = 30


$upperBound = (Get-Item $from).Length / $limit

#Calculate uppbound = filsize / 300


$fromFile = [io.file]::OpenRead($from)
$buff = new-object byte[] $upperBound
$count = $idx = 0
try {
    do {
        $count = $fromFile.Read($buff, 0, $buff.Length)
        if ($count -gt 0) {
            $to = "{0}.{1}.{2}" -f ($rootName, $idx, $ext)
            $toFile = [io.file]::OpenWrite($to)
            try {
                #"Writing $count to $to"
                $tofile.Write($buff, 0, $count)
            } finally {
                $tofile.Close()
            }
        }
        $idx ++
    } while ($count -gt 0)
}
finally {
    $fromFile.Close()
}


#
# Test the passwords:
#

$splitList = Get-ChildItem -Path
↪  "C:\Users\Administrator\Desktop\Passwords\Split" -Attributes !Directory

$splitList | ForEach-Object -Parallel {
    $list = $_

    # $passwords = Get-Content $list


    $number = ($list | Select-Object -ExpandProperty Name)
    $number = $number.Split('.')[-2]
    # $user = 'CN=test' + $number + ',CN=Users,DC=passwordfilter,DC=local'

    $acceptedFile = ($list | Select-Object -ExpandProperty Name) +
    ↪  "_accepted.txt"
    $rejectedFile = ($list | Select-Object -ExpandProperty Name) +
    ↪  "_rejected.txt"
    New-Item -Path "C:\Users\Administrator\Desktop\Passwords\Split\accepted\"
    ↪  -Name $acceptedFile -ItemType "file"
    New-Item -Path "C:\Users\Administrator\Desktop\Passwords\Split\rejected\"
    ↪  -Name $rejectedFile -ItemType "file"
    $acceptedFile = "C:\Users\Administrator\Desktop\Passwords\Split\accepted\"
    ↪  + $acceptedFile
    $rejectedFile = "C:\Users\Administrator\Desktop\Passwords\Split\rejected\"
    ↪  + $rejectedFile


    $User = "test" + $number
    $DomainDN = $(([adsisearcher]"").SearchRoot.path)
    $Filter =
    ↪   "(&(objectCategory=person)(objectClass=user)(samaccountname=$User))"
    $Searcher = New-Object System.DirectoryServices.DirectorySearcher
    $Searcher.Filter = $Filter
    $Searcher.SearchScope = "Subtree"
    $Searcher.SearchRoot = New-Object
    ↪   System.DirectoryServices.DirectoryEntry('LDAP://CN=test' + $number +
    ↪   ',CN=Users,DC=passwordfilter,DC=local')


    $reader = New-Object -TypeName System.IO.StreamReader -ArgumentList $list

    while ($password = $reader.ReadLine()){
    #foreach ($password in $passwords) {

        try {

            $objUser = $Searcher.FindOne().GetDirectoryEntry()
            $objUser.PsBase.Invoke("SetPassword", $password)
            $objUser.CommitChanges()

            # Set-ADAccountPassword -Identity $user -Reset -NewPassword
            ↪   (ConvertTo-SecureString -AsPlainText $password -Force)
            # Write-Output($password + " accepted by " + $user)

            # The password change succeeded, so log the password in the allowed
            ↪   list
            Out-File -FilePath $acceptedFile -Append -Encoding utf8
            ↪   -InputObject $password
        }
        catch {
            # The password change failed, so log the password in the denied
            ↪   list
            # Write-Output($password + " rejected by " + $user+ " " +
            ↪   $_.Exception.Message)

            ($password + " " + $_.Exception.Message) | Out-File -FilePath
            ↪   $rejectedFile -Append -Encoding utf8
        }
    }
```

```
} -ThrottleLimit $limit



#
# Combine the split files:
#

$rejectedList = Get-ChildItem -Path "C:\Users\Administrator\Desktop\Passwords\Split\rejected"
New-Item -Path "C:\Users\Administrator\Desktop\Passwords\Split\rejected\" -Name "combined.txt" -ItemType "file"

$upperBound = $upperBound*2

foreach($list in $rejectedList) {
    $file = "C:\Users\Administrator\Desktop\Passwords\Split\rejected\" + ($list | Select-Object -ExpandProperty Name)


    $fromFile = [io.file]::OpenRead($file)
    $toFile = [io.file]::Open("C:\Users\Administrator\Desktop\Passwords\Split\rejected\combined.txt", [io.FileMode]::Append, [io.FileAccess]::Write)
    $buff = new-object byte[] $upperBound

    try {
        $count = $fromFile.Read($buff, 0, $buff.Length)

        $tofile.Write($buff, 0, $count)
    } finally {
         $tofile.Close()
    }
    $fromFile.Close()
}


$acceptedList = Get-ChildItem -Path "C:\Users\Administrator\Desktop\Passwords\Split\accepted"
New-Item -Path "C:\Users\Administrator\Desktop\Passwords\Split\accepted\" -Name "combined.txt" -ItemType "file"

```

```
foreach($list in $acceptedList) {
    $file = "C:\Users\Administrator\Desktop\Passwords\Split\accepted\" + ($list
    ↪   | Select-Object -ExpandProperty Name)


    $fromFile = [io.file]::OpenRead($file)
    $toFile =
    ↪   [io.file]::Open("C:\Users\Administrator\Desktop\Passwords\Split\accepted\combined.txt",
    ↪   [io.FileMode]::Append, [io.FileAccess]::Write)
    $buff = new-object byte[] $upperBound

    try {
        $count = $fromFile.Read($buff, 0, $buff.Length)

        $tofile.Write($buff, 0, $count)
    } finally {
         $tofile.Close()
    }
    $fromFile.Close()
}

Write-Output("Deleting files: " + $stopWatch.Elapsed)

Get-ChildItem "C:\Users\Administrator\Desktop\Passwords\Split" -Include *split*
↪   -Recurse -Force | Remove-Item -Recurse -Force




$stopWatch.stop()
Write-Output("Time: " + $stopWatch.Elapsed)
```

A.2 Password Guessing Attacks Scripts

A.2.1 Python Script For Offline Password Guessing Using Hashcat

```python
import subprocess

def brute(file_path):
    hash_cmd = f'hashcat -d 1,2 -m 1000 --attack-mode 3 --increment --increment-max=11 {file_path} "?d?d?d?d?d?d?d?d?d?d?d" -o hashcat_attack1.txt --outfile-format=2 --potfile-path=potfile.pot'
    subprocess.run([hash_cmd], shell=True)
    hash_cmd = f'hashcat -d 1,2 -m 1000 --attack-mode 3 --increment --increment-max=8 {file_path} "?l?l?l?l?l?l?l?l" -o hashcat_attack1.txt --outfile-format=2 --potfile-path=potfile.pot'
    subprocess.run([hash_cmd], shell=True)
    hash_cmd = f'hashcat -d 1,2 -m 1000 --attack-mode 3 --increment --increment-max=8 {file_path} "?u?u?u?u?u?u?u?u" -o hashcat_attack1.txt --outfile-format=2 --potfile-path=potfile.pot'
    subprocess.run([hash_cmd], shell=True)
    hash_cmd = f'hashcat -d 1,2 -m 1000 --attack-mode 3 --increment --increment-max=8 {file_path} "?s?s?s?s?s?s?s?s" -o hashcat_attack1.txt --outfile-format=2 --potfile-path=potfile.pot'
    subprocess.run([hash_cmd], shell=True)
    hash_cmd = f'hashcat -d 1,2 -m 1000 --attack-mode 3 --increment --increment-max=6 {file_path} "?a?a?a?a?a?a" -o hashcat_attack1.txt --outfile-format=2 --potfile-path=potfile.pot'
    subprocess.run([hash_cmd], shell=True)

def dictionary(file_path, wordlist):
    hash_cmd = f'hashcat -d 1,2 -m 1000 {file_path} {wordlist} -o hashcat_attack2.txt --outfile-format=2 --potfile-path=potfile.pot'
    subprocess.run([hash_cmd], shell=True)

def rule(file_path, wordlist, ruleset):
    hash_cmd = f'hashcat -d 1,2 -m 1000 {file_path} {wordlist} -r {ruleset} -o hashcat_attack3.txt --outfile-format=2 --potfile-path=potfile.pot'
    subprocess.run([hash_cmd], shell=True)

def PassGAN(file_path, wordlist, ruleset):
    hash_cmd = f'hashcat -d 1,2 -m 1000 {file_path} {wordlist} -r {ruleset} -o hashcat_attack4.txt --outfile-format=2 --potfile-path=potfile.pot'
    subprocess.run([hash_cmd], shell=True)

```

```python
def main():
    file_path = '' #input all accepted passwords for a given filter
    wordlist = 'realhuman_phill.txt'
    ruleset = 'OneRuleToRuleThemAll.rule'
    GAN = 'gen_passwords100mil.txt'
    try:
        brute(file_path)
        dictionary(file_path, wordlist)
        rule(file_path, wordlist, ruleset)
        PassGAN(file_path, GAN, ruleset)
    except:
        print("An error occurred.")

if __name__ == "__main__":
    main()

```

A.2.2 Python Script For Online Password Guessing

```python
def count_matching_entries(file1, file2, num_entries):
    with open(file1, 'r') as f1:
        entries1 = set(f1.read().splitlines()[:num_entries])

    with open(file2, 'r') as f2:
        entries2 = set(f2.read().splitlines())

    matching_entries = entries1.intersection(entries2)
    return len(matching_entries)

file1_path = '' #input list of common passwords
file2_path = '' #input all accepted passwords for a given filter

guesses = [50, 100, 200, 10000, 54720]
for i in range(5):
    matching_count = count_matching_entries(file1_path, file2_path, guesses[i])
    print(f"Passwords guessed with {guesses[i]}:", matching_count)

```

A.3 Score Calculations For Password Guessing Attacks

A.3.1 Normalized Offline Brute Force Scores

$$\text{Entra Brute Force} = 22.55\%$$

$$\text{Normalized Score} = \frac{(100 - 22.55) - 0}{100 - 0} = 0.7745$$

$$\text{Log Normalized Score} = \frac{log(100 - 22.55)}{log(100)} = 0.94450845$$

$$\text{Lithnet Brute Force} = 13.72\%$$

$$\text{Normalized Score} = \frac{(100 - 13.72) - 0}{100 - 0} = 0.8628$$

$$\text{Log Normalized Score} = \frac{\log(100 - 13.72)}{\log(100)} = 0.9679501627$$

$$\text{OpenPassWordFilter Brute Force} = 13.12\%$$

$$\text{Normalized Score} = \frac{(100 - 13.12) - 0}{100 - 0} = 0.8688$$

$$\text{Log Normalized Score} = \frac{\log(100 - 13.12)}{\log(100)} = 0.9694656131$$

A.3.1.1 Normalized Offline Dictionary Scores

$$\text{Entra Dictionary} = 10.75\%$$

$$\text{Normalized Score} = \frac{(100 - 10.75) - 0}{100 - 0} = 0.8925$$

$$\text{Log Normalized Score} = \frac{\log(100 - 10.75)}{\log(100)} = 0.9752928771$$

$$\text{Lithnet Dictionary} = 0.25\%$$

$$\text{Normalized Score} = \frac{(100 - 0.25) - 0}{100 - 0} = 0.9975$$

$$\text{Log Normalized Score} = \frac{\log(100 - 0.25)}{\log(100)} = 0.9994440371$$

$$\text{OpenPasswordFilter Dictionary} = 13.12\%$$

$$\text{Normalized Score} = \frac{(100 - 13.12) - 0}{100 - 0} = 0.8688$$

$$\text{Log Normalized Score} = \frac{\log(100 - 13.12)}{\log(100)} = 0.9996503442$$

A.3.1.2 Normalized Offline Dictionary+Rules Scores

$$\text{Entra Dictionary} = 31.48\%$$

$$\text{Normalized Score} = \frac{(100 - 31.48) - 0}{100 - 0} = 0.6852$$

$$\text{Log Normalized Score} = \frac{\log(100 - 31.48)}{\log(100)} = 0.9178925172$$

$$\text{Lithnet Dictionary} = 8.96\%$$

$$\text{Normalized Score} = \frac{(100 - 8.96) - 0}{100 - 0} = 0.9104$$

$$\text{Log Normalized Score} = \frac{\log(100 - 8.96)}{\log(100)} = 0.9796083324$$

$$\text{OpenPasswordFilter Dictionary} = 19.89\%$$

$$\text{Normalized Score} = \frac{(100 - 19.89) - 0}{100 - 0} = 0.8011$$

$$\text{Log Normalized Score} = \frac{\log(100 - 19.89)}{\log(100)} = 0.9518268772$$

A.3.2 Normalized Offline PassGAN+Rules Scores

$$\text{Entra Dictionary} = 3.94\%$$

$$\text{Normalized Score} = \frac{(100 - 3.94) - 0}{100 - 0} = 0.9606$$

$$\text{Log Normalized Score} = \frac{\log(100 - 3.94)}{\log(100)} = 0.9912702146$$

$$\text{Lithnet Dictionary} = 8.48\%$$

$$\text{Normalized Score} = \frac{(100 - 8.48) - 0}{100 - 0} = 0.9152$$

$$\text{Log Normalized Score} = \frac{\log(100 - 8.48)}{\log(100)} = 0.9807527468$$

$$\text{OpenPasswordFilter Dictionary} = 10.24\%$$

$$\text{Normalized Score} = \frac{(100 - 10.24) - 0}{100 - 0} = 0.8976$$

$$\text{Log Normalized Score} = \frac{\log(100 - 10.24)}{\log(100)} = 0.976540584$$