



TR

USKUDAR UNIVERSITY

INSTITUTE OF SCIENCE

DEPARTMENT OF CYBER SECURITY PROGRAM

MASTER THESIS

**ENHANCING PASSWORD SECURITY AND MEMORABILITY
FOR ARABIC USERS USING ARTIFICIAL INTELLIGENCE**

ROAA ADEL SAIED HAMAD

Thesis advisor

ASSISTANT PROF. HAMZA ABUNIMA

ISTANBUL-2024

T.R.
USKUDAR UNIVERSITY
INSTITUTE OF SCIENCE

DEPARTMENT OF CYBER SECURITY
MASTER'S DEGREE PROGRAM IN CYBER SECURITY

MASTER THESIS

**ENHANCING PASSWORD SECURITY AND MEMORABILITY
FOR ARABIC USERS USING ARTIFICIAL INTELLIGENCE**

ROAA ADEL SAIED HAMAD

Thesis advisor
ASSISTANT PROF. HAMZA ABUNIMA

ISTANBUL-2024

ENHANCING PASSWORD SECURITY AND MEMORABILITY FOR ARABIC USERS USING ARTIFICIAL INTELLIGENCE

ABSTRACT

This thesis investigates the integration of Arabic characters into password creation through the development of a novel Password Generator and Password Requirements Extractor. The aim is to enhance password strength and memorability while addressing the challenges of multilingual password requirements. The Password Generator modifies the user's chosen base word to include Arabic characters, symbols, and Latin characters, generating secure and easy-to-remember passwords, and generating an English password in case Arabic letter is not accepted. Additionally, the Password Requirements Extractor employs AI to analyze and extract password policies from codes, providing users with clear guidelines and preventing password rejection due to unsupported characters.

The study compares the strength of English and Arabic passwords through entropy calculations, brute-force attack simulations, and online password strength meters. Results show that Arabic passwords significantly increase entropy, providing stronger resistance to brute-force attacks. While English passwords were cracked within six seconds, Arabic passwords took double that time or more. Online strength checkers estimated the cracking time for Arabic passwords to be 849,000 years, compared to just two years for English passwords. The Password Requirements Extractor effectively identifies password policies, addressing the lack of non-English rejection datasets by generating a custom AI-powered dataset, thus improving the tool's ability to manage multilingual passwords. The findings demonstrate that incorporating Arabic characters enhances password complexity, security, and user experience without compromising memorability. The user-friendly interface, browser extension, and AI-driven policy extraction make the Password Generator and Extractor a comprehensive solution for strengthening password security in multilingual environments.

Keywords: Password Generation, Password Strength, Password Memorability, Multilingual Authentication, AI-driven Password Policy Extractor..

THANK

Prof. Dr Hamza Abunima, your guidance, and support have truly been a beacon of light in my journey. Your belief in my ideas, your positive energy, and the confidence you've instilled in me—especially when you described my writing as demure— will continue to shape and strengthen my confidence. Thanks T.C. Uskudar University, Istanbul, for providing a comfortable study environment that has greatly supported my academic journey.

I would also like to thank the cyber security faculty members at Üsküdar University for their help, teachings, and moral support that they have given me during my studies.

My beloved family, words can hardly express the depth of my gratitude for all that you've done for me. My parents, your unwavering encouragement and belief in me have been the foundation of my dreams. Your financial, emotional, and moral support has lifted me in ways I can't fully articulate. You've taught me that it's never too late to pursue my goals, a lesson that will stay with me always. Thank you for being my rock and my inspiration.

DECLARATION

I have obtained all the information and documents in this study within the framework of academic rules. I have presented all visual, audio, and written communication and results following scientific and ethical practices and have not falsified any data I used. I have referred to the sources I used following scientific norms, and my thesis is original except for the cases cited. It is produced and written according to the Üsküdar University Institute of Sciences Thesis Writing Guide.



Roaa Adel Saied Hamad

Signature

CONTENTS

ABSTRACT.....	i
THANK.....	i
DECLARATION	ii
CONTENTS	iii
INDEX OF TABLES	vi
INDEX OF FIGURES	vii
INDEX OF SYMBOLS AND ABBREVIATIONS	x
1. INTRODUCTION	1
1.1 Problem Statement	2
1.2 Research Objectives	3
1.3 Scope	4
1.4 Thesis Contribution	5
1.5 Dissertation Outlines	6
2. GENERAL INFORMATION.....	7
2.1 Introduction	7
2.2 Memorable Passwords	9
2.3 Password Policies.....	10
2.4 Leaked Passwords	11
2.5 Password Categories	12
2.6 Cracking Hashes.....	14
2.7 Password Generators	15
2.7.1 Random Password Generators.....	16
2.7.1.1 ALPHANUM	16

2.7.1.2	Programmed Random Password Generators.....	16
2.7.2	Pronounceable Password Generators	17
2.7.3	Passphrase Password Generators.....	18
2.7.4	Pattern-Based Password Generators.....	19
2.7.4.1	Password Generation Based On User Inputs	19
2.7.5	Hybrid Password Generators	20
2.7.5.1	VowPass.....	20
2.7.5.2	PassMan	21
2.8	Mnemonic password strategy.....	22
2.8.1	Alphapwd	22
2.8.2	AvoidPwd.....	22
2.9	Identification of Gaps in the Literature	23
2.10	Summary	25
3.	EQUIPMENT AND METHOD.....	27
3.1	Introduction.....	27
3.2	Service Providers and Users.....	27
3.3	Key features of the developed tool:	28
3.4	Password Requirements Extractor	30
3.4.1	Webpage Content Extraction.....	31
3.4.2	Search for Specific Sentences in JS Code	33
3.4.3	Dataset Preparation.....	35
3.4.4	Logistic Regression	37
3.4.5	Classifying Requirement Sentences	40
3.4.6	Similar Sentence Identification	42
3.5	Arabic Password Generator.....	43
3.5.1	Arabizi or Franco-arabic.....	43
3.5.2	Search the Password	44
3.5.3	Arabic Random Letters Replacement.....	46
3.5.4	Latin Random Letters Replacement	48
3.5.5	Iterative Modification Process.....	49

3.6	Web Application Integration.....	53
3.7	HTML Structure and Client-Side JavaScript Functionality.....	55
3.8	Summary	61
4.	FINDINGS AND DISCUSSION	62
4.1	Introduction	62
4.2	Password Requirements Extractor Examination	62
4.3	Arab Password Generator Examination	66
4.3.1	Uniqueness	67
4.3.2	Memorization	68
4.3.3	Password Entropy	68
4.3.4	Brute-Force Attack	69
4.3.5	Online password strength checkers	83
4.4	Summary	85
5.	CONCLUSION & RECOMMENDATION	89
5.1	Conclusion.....	89
5.2	Recommendations	92
6.	RESOURCES.....	93
7.	ANNEXES.....	99
7.1	Appendix A. Dataset and Codes	99
7.2	Appendix B. CV.....	128

INDEX OF TABLES

Table 2.1: Non-English passwords leaked.....	24
Table 3.1: Non-Latin acceptance comparison	28
Table 3.2: Examples of <code>classify_sentence</code> function results.....	41
Table 3.3: Arabizi writing style	44
Table 3.4: <code>replacement_dict</code>	47
Table 3.5: <code>replace_eng_dict</code>	49
Table 4.1: Arabic password acceptance.....	66
Table 4.2: Arab password samples	66
Table 4.3: Results of cracking hashes.....	77
Table 4.4: Results of cracking sha512 hashes	79
Table 4.5: Results of cracking MD5, SHA512 and SHA256 hashes	80
Table 4.6: Extractor summary results	86
Table 4.7: Results of cracking SHA512 hashes.....	87
Table 4.8: Evaluation of password strength.....	88

INDEX OF FIGURES

Figure 2.1: Sample of 8x8 password	8
Figure 3.1: Password requirements extractor flowchart	29
Figure 3.2: Password Generator Flowchart	30
Figure 3.3: Password requirements extractor	31
Figure 3.4: Snippet screenshot of the full extracted HTML code	32
Figure 3.5: Snippet screenshot of the extracted script of requirements.....	32
Figure 3.6: <code>extract_js_from_url</code> function.....	33
Figure 3.7: <code>search_specific_sentences</code> function.....	34
Figure 3.8: Example of found sentences (requirements) list	34
Figure 3.9: Sample of the dataset 'arbarb.txt'.....	35
Figure 3.10 : The code used to convert text file into json file	36
Figure 3.11: Running the code to convert .txt to .json.....	36
Figure 3.12 : Sample of the dataset "labeled_arbarbup.json"	37
Figure 3.13 : Loading and splitting the dataset.....	38
Figure 3.14: Code for vectorizing	39
Figure 3.15: The evaluation metrics of training	40
Figure 3.16: LogisticRegression model.....	40
Figure 3.17: <code>classify_sentence</code> function.....	42
Figure 3.18: <code>find_similar_sentence</code> function	42
Figure 3.19: Password Generator Steps.....	43
Figure 3.20: <code>get_search_results</code> function	45
Figure 3.21: Example Output of Running this Function	45
Figure 3.22: <code>random_replace</code> Function	46
Figure 3.23: Example of a Modified Word.....	47
Figure 3.24: <code>random_engreplace</code> function.....	48
Figure 3.25: Example of a Latin Modified Word	49
Figure 3.26 : Calculating password metrics function	51
Figure 3.27: Results of testing 50 passwords	52
Figure 3.28 : Closer examination of the results	52

Figure 3.29: <code>modify_word</code> Function	53
Figure 3.30: Output of <code>modify_word</code> Function.....	53
Figure 3.31: The Chrome Extension.....	55
Figure 3.32: Route <code>submit()</code>	55
Figure 3.33: <code>popup.html</code>	56
Figure 3.34: Snippet Code From <code>popup.js</code>	57
Figure 3.35: <code>displayFoundSentences</code> Function	57
Figure 3.36: <code>urlSubmitButton.addEventListener</code>	59
Figure 3.37: <code>wordSubmitButton.addEventListener</code>	60
Figure 4.1: Windows live password requirements.....	63
Figure 4.2: Windows live Arabic password rejection.....	63
Figure 4.3: Yahoo extracted requirements.....	64
Figure 4.4: Yahoo Arabic password rejection	64
Figure 4.5: GMX extracted requirements	65
Figure 4.6: GMX rejection.....	65
Figure 4.7: Requirements Extractor of Arabic password acceptance	66
Figure 4.8: Verified number of search results	67
Figure 4.9: Steps of pressing keyboard buttons to type the password.....	68
Figure 4.10: Generating <code>arwordlist.txt</code> using crunch	70
Figure 4.11: Sample of the <code>arwordlist.txt</code> password combinations.....	70
Figure 4.12: Generating hashes	71
Figure 4.13: Hashes identifier.....	71
Figure 4.14: Hashcat cracking the md5 hash of the Arabic password.....	72
Figure 4.15: Hashcat cracking results of the Arabic password.....	72
Figure 4.16: Hashcat cracking the sha256 hash of the Arabic password	73
Figure 4.17: Hashcat cracking results of the Arabic password.....	73
Figure 4.18: Generating <code>enwordlist.txt</code> using crunch	74
Figure 4.19: Sample of the <code>enwordlist.txt</code> password combinations.....	74
Figure 4.20: Hashcat cracking the md5 hash of the English password	75
Figure 4.21: Hashcat cracking results of the English password	75
Figure 4.22: The size of <code>enwordlist2.txt</code> English password combinations	76
Figure 4.23: Hashcat cracking the sha256 hash of the English password	76

Figure 4.24: Hashcat cracking results of the English password	76
Figure 4.25: Hashcat cracking results of the Arabic password.....	78
Figure 4.26: Hashcat cracking results of the English password	78
Figure 4.27: Hashcat cracking results of the English password	79
Figure 4.28: Cracking ".D 1 \$", "\$.o1 D"	83
Figure 4.29: Cracking "1oDo\$1", "1oDDoo"	81
Figure 4.30: Cracking "e1سٲL^", "^eسٲeٲL"	81
Figure 4.31: Cracking "eeL^L2", "L2e^1h"	82
Figure 4.32: Cracking "ش%سH4ؤ", "H4%ش4ؤ"	83
Figure 4.33: Cracking "shA^5@", "5A5s@^"	82
Figure 4.34 Number of guesses vs cracking time	83
Figure 4.35: The password meter	84
Figure 4.36: The Password Checker	85

INDEX OF SYMBOLS AND ABBREVIATIONS

LSTM	Long Short-Term Memory
RNN	Recurrent Neural Networks
GAN	Generative Adversarial Networks
ID	Identification
ML	Machine Learning
AI	Artificial Intelligence
NLP	Natural Language Processing
TF-IDF	Term Frequency-Inverse Document Frequency
JS	JavaScript
PIN	Personal identification number
IBM	International Business Machines (a multinational technology company)
SSO	Single Sign-On
IT	Information Technology
IoT	Internet of Things
BCI	Brain-computer Interfaces
EEG	Electroencephalogram
ASCII	American Standard Code for Information Interchange (a standard data-encoding format)
DPPG	Dynamic Password Policy Generator
PCFG	Problematic Context-Free Grammars

PGTCN	Temporal Convolutional Network-based Password Guessing
ITCN	Improved Temporal Convolutional Network
MDBSCAN	Modified Density-Based Spatial Clustering of Applications with Noise
MD5	Message Digest
SHA1	Secure Hash
URL	Uniform Resource Locator
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol

CHAPTER 1

INTRODUCTION

Password authentication remains the most widely used method for securing user accounts and sensitive information. Finding a replacement for passwords remains an unsolved challenge. Passwords offer the advantage of not requiring any additional devices or equipment; the user simply types a sequence of characters, which may include lowercase letters, uppercase letters, digits, or symbols. However, to resist cracking, passwords must adhere to security guidelines. Short passwords or those with a limited variety of character classes are quick to create and easy to remember, but they can be easily cracked. In contrast, strong passwords are longer and incorporate a wide range of character classes, making them more secure but also more confusing, difficult to remember, and time-consuming to create.

Despite the availability of various authentication methods, password authentication remains dominant, as noted by Zimmermann & Gerber (2020). In their study, 41 users interacted with 20 different authentication schemes, with participants rating passwords as the most preferred authentication method followed by fingerprints as the highest in terms of preference, usability, and intention to use but lowest in terms of expected problems and effort.

A considerable amount of research has focused on passwords, including studies on strength, memorization, brute force attacks, and cracking methods such as neural networks. Zhang et al. (2020) treated passwords as text strings and utilized neural network architectures such as Long Short-Term Memory (LSTM), Recurrent Neural Networks (RNN), and Generative Adversarial Networks (GAN) for password guessing through learning or transfer learning. However, deep learning methods heavily rely on training datasets, particularly large leaked password datasets.

Nevertheless, passwords continue to be widely used, as evidenced by Ezugwu et al. (2023), who assessed the experiences of staff and students in a university community in Southeastern Nigeria. Respondents reported positive experiences with password usage, indicating that there are still unexplored aspects and strategies for creating strong passwords.

1.1 Problem Statement

Sensitive information nowadays is digitized as personal data, financial transactions, or intellectual property. Securing information is crucial in preventing unauthorized access, maintaining trust with users, and protecting against financial losses and reputational damage. There are several methods to secure data, such as encryption, access control, and strong authentication. Implementing strong authentication methods that serve as a primary line of defense, such as face identification (ID), fingerprint, or passwords, is the process of verifying that someone is who they claim to be. It is necessary to prevent unwanted potential access to private data, or personal information, leading to identity theft, fraud, or unauthorized use of resources. Face ID and fingerprint are not supported by all service providers or devices. In some situations, such as a period of inactivity or unavailability, password recognition is needed as an alternative.

While passwords are essential, they must be strong and unique by following the best practices to avoid easily guessable combinations, common patterns and phrases, or using personal information. User mostly prefer certain approaches or patterns to follow in the password creation process, whether keyboard walks, including personal information in the password or habits, or leet words. Users have two primary concerns when it comes to passwords: strength and memorability. Strength means meeting complex criteria, and memorability means easy recall. To force the user to create a strong password, many service providers require passwords to meet specific complex requirements, such as a minimum length, the inclusion of lowercase letters, uppercase letters, numbers, and special characters, which is not easy for non-English speakers. Balancing these requirements while creating an easy-to-remember password is difficult. Memorable passwords need to be recalled easily without the need to write them down, which might make them easier for the attacker to guess. This vulnerability poses significant risks to personal and financial security, potentially resulting in bank fraud, identity theft, or data breaches. If a banking password is compromised, unauthorized access can lead to stolen funds and fraudulent transactions. Leaked passwords can also facilitate identity theft, where attackers impersonate victims to open new credit accounts or take out loans in their name. Additionally, compromised passwords for services like email or cloud storage can lead to data breaches, exposing sensitive information and causing reputational and legal issues. Forgetting a password can also lock users out of

essential services, such as banking, healthcare, or communication platforms, leading to delays and significant inconvenience. Non-English speakers face a challenging task, so using their names or the simple words they know is often the result. This is why password generators employing various strategies are widely utilized. These tools help users adhere to password policies, which are often not communicated by services or sites until a password is rejected.

Strategies for creating strong, memorable passwords are continuously evolving. These strategies include using phrases, formulas, hashes, and word combinations with symbols, leading to complicated passwords that are hard to use or recall without a password manager to store these passwords. It is getting more secure but harder by adding a leaked password checker that either checks if the password is leaked or mangles it to alert the user about all possible combinations. However, cracking and guessing methods have also evolved, with neural networks being used to extract meaningful words or substituted symbols from passwords to guess them.

Notably, the majority of the research has been focused on Latin-letter passwords. To create secure and hard-to-guess passwords, password generators have begun to make password creation a hard task for the user, such as using the hash of a chosen word or part of the hash to be used as a password. The user then needs to regenerate this hash to retrieve the password and use it to sign in.

It seems as though Latin-letter passwords are becoming less useful and using math formulas and hashes is seen as a solution. Why is diversity in languages and alphabet characters not offered as an option in password generators? Communities where the English language is not universally learned by every person are forced to use Latin letters, which are used in the English language as passwords. This poses challenges for individuals who have a limited vocabulary, which is reflected in their passwords.

1.2 Research Objectives

This research aims to study a new strategy in password creation by using right-to-left Arabic letters or its keyboard, leveraging the strengths of random passwords in an easily memorable way, and analyzing the results. It also includes extracting password requirements from the service providers to provide them clearly to the user.

The objectives of this thesis can be summarized as follows:

1. To extract and provide clear password requirements from service providers for the user using Machine Learning (ML).
2. To evaluate the popularity of the chosen Arabic password of the user.
3. To assist the user in creating memorable passwords that meet high-security requirements and align with their personal preferences. Additionally, offer modified versions of Arabic password and alternative English password, in case service providers do not support Arabic characters.

1.3 Scope

This research focuses on a novel strategy for generating strong, random-looking passwords based on user input, developed through two complementary tools: a Password Generator and a Password Requirements Extractor. With the increasing reliance on technology, the task of password creation has become tedious and burdensome. Modifying passwords to meet the new policies repeatedly often leads users to make mistakes, as they may mistakenly believe that their password is more secure and hard to guess. Additionally, passwords composed primarily of English letters have become overused, resulting in a higher incidence of guessing or cracking attacks, particularly those targeting breached or leaked password lists. Research indicates that English letters and words dominate password choices, highlighting the need for both service providers and users to adopt alternatives that expand the search space for attackers and enhance password security.

The Password Generator developed in this study integrates Arabic letters with other character classes, creating multilingual passwords that increase complexity while maintaining memorability. This approach encourages users to create passwords beyond traditional English letters, making password patterns more resistant to common attacks. The generator leverages user input and randomizes characters, shuffling between Arabic, Arabizi (Franco-Arabic), symbols, and Latin characters, providing an adaptable, randomized password with each generation.

The Password Requirements Extractor complements the generator by employing Artificial Intelligence (AI) and Natural Language Processing (NLP) to analyze and

extract password policies, ensuring passwords meet the necessary criteria with clear user guidelines. Logistic regression with Term Frequency-Inverse Document Frequency (TF-IDF), help identify essential password requirements stated in JavaScript (JS) and other source codes. This process assists in pinpointing the policies—even if ambiguously written—that may affect users' choices, especially for non-English password constraints. ML models trained help detect non-English password rejection rules with accuracy.

Upon investigation, it became evident that there is a lack of studies focused on the use of Arabic letters in passwords or even a leaked password datasets that include such passwords. Unfortunately, not all service providers accept Arabic letters as password; some allow them but enforce requirements for uppercase or lowercase letters, digits, and symbols. The extractor identifies these restrictions, thus preventing users from repeated trial-and-error attempts at meeting password rules.

Together, the Password Generator and Requirements Extractor offer a comprehensive solution. The generator's strategy of incorporating Arabic letters, Arabizi (Franco-Arabic), and character drawing serves Arabic-speaking users and those from countries using similar scripts, such as Persian and Urdu, accommodating various regional needs and expanding the overall security framework for password creation.

1.4 Thesis Contribution

Passwords are required to authenticate the user in every service; creating a password that is memorable and hard to guess is a boring task for the user. Most users keep using certain popular passwords, as evidenced in many leaked password datasets. Additionally, Latin letters are the dominant alphabet found in datasets, which must be avoided because of the wide range of password guessing and generating algorithms that are trained. A novel approach to password generation is needed to help the user create hard-to-guess passwords. To improve the creation of strong passwords, using the Arabic alphabet with Franco-Arabic or Arabizi can be beneficial. This right-to-left alphabet automatically changes the order of characters when mixed with numbers and symbols, while users enter these characters in the same order as the original words chosen to be used in the password. Note that some service providers do not support Arabic letters in

passwords. Therefore, as an alternative to Arabic letters, pressing the QWERTY keyboard to resemble a random string is used to generate a Latin-letter password.

Some service providers forbid using non-English alphabets or force the user to add lowercase and uppercase letters without mentioning these password requirements clearly to the user. The tool features an AI password requirements extractor that helps the user to gain a comprehensive understanding during password creation.

The proposed password generator in this thesis aims to enhance the aspect of creating strong passwords using right-to-left Arabic letters. Besides a password requirements extractor to help the user.

1.5 Dissertation Outlines

The structure of this dissertation is organized into five chapters, as follows:

- i. Chapter 1 provides an overview of the research background and introduces the general concept of passwords and the password requirements extractor.
- ii. Chapter 2 presents a literature review of prior studies on password generators and discusses the research review for this research's implementation.
- iii. Chapter 3 describes how the research was implemented in the industry and presents codes.
- iv. Chapter 4 presents the results of the tool, including pieces of evidence and screenshots.
- v. Chapter 5 presents the conclusion of the research, detailing how this research can benefit industries and outlining its limitations are based on its findings.

CHAPTER 2

GENERAL INFORMATION

2.1 Introduction

Since 1976, with the widening of teleprocessing applications, the demand for new cryptographic systems has increased. Diffie and Hellman (1976) introduced the concept of public-key cryptography to secure communications, highlighting the importance of Secrecy. Lamport et al. (1981) introduced a method of password authentication to access a computer system or service remotely, even in the presence of insecure communication, through a secure one-way encryption function that could be used 1000 times, then the user must choose a new value. Password usage expanded as Bergman et al. (1988) provided a password-protected power control device to power the computer system on and off. This need arose to access data in a personal computer via telephone, where the user was required to leave the computer switched on with the modem in auto-answer mode.

As single-factor passwords became insufficient, Weiss et al. (1990) discussed two-factor authentication. Several token products were developed, so appears to be two passwords; one is the classic password or Personal Identification Number (PIN), the other, the pseudo-random code displayed on the token device or card (i.e., the cardcode). The International Business Machines Corporation company (IBM) used the phrase 'static password' for the traditional memorized password, and 'one-time password' generated by the token. Smart cards or tokens were very helpful with passwords, as Chang et al. (1993) presented a password authentication scheme without the verification table, which was one of the vulnerabilities; the user's identity and the password were associated with his smart card. O'Gorman et al. (2003) compared passwords, tokens, and biometrics, and recommended that a password is a good choice if the user needs to memorize a single password with high entropy. For memorizing multiple passwords, Single Sign-On (SSO) is convenient, where the token generates or stores multiple passcodes in a secure manner. Nowadays, passwords can be used without the smart card due to hashing functions and the salt value added to passwords to ensure they are saved securely. Sælensminde, K., & Boonjing, V. (2010) proposed passwordless authentication for websites, which is convenient for users by removing the

requirement to remember a password. A token is generated and associated with a browser to authenticate the user. On the other hand, Safdar & Hassan. (2010) proposed two-dimensional $n \times n$ passwords that possess much more strength compared to linear passwords. These passwords are provided under certain alarming situations. Figure 2.1 shows a sample of 8x8 passwords.

A	B	C	D	1	2	E	F
*	-	\$	C	M	J	i	K
L	M	N	O	P	4	5	6
Q	R	S	T	U	V	w	X
y	z	1	2	3	7	4	2
q	w	e	r	t	T	R	E
W	Q	5	4	3	7	6	1
2	9	q	z	d	r	f	T

Figure 2.1: Sample of 8x8 password

With the increasing number of personal devices, the importance of passwords has increased. Shen et al. (2016) investigated how password characteristics have changed over time (1989-2016) and provided a comprehensive understanding of user practices in real-life passwords (over 6 million passwords) in terms of length, composition, and selection. The general findings include: the average password length is at least 12% longer to increase the strength, 75% of passwords have a length between 8 and 10 characters; a significant increase in using only numbers as passwords, and easy-to-reach symbols are always the first choice to be added to the password; a remarkable increase of about 40% in the use of meaningful combinations as passwords, and the use of the most common passwords or login names as passwords. Additionally, most users prefer alphanumeric characters in passwords. Furnell et al (2018-2022) conducted a series of studies every 3-4 years to examine the extent to which leading websites guide and support users in making appropriate passwords. With over ten years between the studies, found that the overall situation in 2018 remains largely similar to 2007. In 2022, passwords continue to occupy an interesting position in cybersecurity, and there are marginal improvements in some areas, such as enforcing a minimum password length,

but there are still numerous omissions in areas that could support the user in improving password practices.

A weak password represents the greatest vulnerability for the user, impacting not only their access but also potentially implicating them in cybercrimes they have not committed. Humans are well known as the weakest link in security. Alomari & Thorpe (2019) explored the differences in passwords between non-student and Information Technology (IT) professionals. They found that IT professionals engage in more secure behaviors, such as recording fewer financial passwords, reusing fewer passwords, and using more random passwords. Renaud et al. (2019) investigated whether the endowment effect explains resistance to change in password creation and concluded that people indeed feel endowed with their password creation routines. As technology becomes more prevalent, the need for online accounts and passwords for nearly everything has increased. A survey conducted at Wichita State University demonstrates that, while many people are aware of proper practices for safe password creation, they do not follow them. Even strict policies can reduce password creativity, resulting in weaker passwords (Davis, 2022).

Passwords serve as the primary method of authentication and access control in our modern life. The importance of passwords spans various aspects, including verifying the identity of the user, thwarting unauthorized access to personal and confidential data, and mitigating breaches and cyberattacks. Password breaches lead to financial loss for individuals and organizations, and customers may lose trust in a company. This is especially concerning in the context of Internet of Things (IoT) devices. If these devices are compromised, attackers can gain access to surveillance cameras, perpetrate data theft, and even cause physical harm in cases such as smart homes and industrial control systems.

2.2 Memorable Passwords

Passwords play a critical role in securing users' online services and protecting personal information. They serve as a barrier against unauthorized access, so they must be difficult to guess by creating complex password. Users who create simple or easy-to-guess passwords are vulnerable to attacks such as brute force or dictionary attacks. On the other hand, complex passwords pose a challenge for the human mind to remember over time. It is not easy to strike a balance between memorable and hard-to-guess

passwords. Brain-computer interfaces (BCI) have been used to classify memorable and non-memorable passwords based on the electroencephalogram (EEG) collected by the BCI. It was found that the memorable group consists of common passwords known as the “worst password list”; while randomly generated sequences of characters, symbols, and numbers are classified as the non-memorable group (Alomari, 2017).

Users usually create a password that has an emotional or personal connection, such as their language and country, to make it as memorable as possible. There is no other way to memorize passwords if you have no connection to them. Some users prefer using keyboard walks such as same row, zig-zag, and snake walks, while others create passwords with personal meanings such as names, phrases, locations, stories, or word combinations and mangling that are meaningful or related to the user which is usually easier to remember. On the contrary, it is difficult for the human mind to remember passwords that lack personal data or are randomly generated (Mori, 2019).

Linguistics and culture influence password creation habits. Gan et al. (2022) analyzed non-English speaking users and found that they may use predictable passwords such as 'abcd123', 'aabbcc1' or simply write their names using English letters. Using words from their language converted into American Standard Code for Information Interchange (ASCII) strings leads to unknown or meaningless words in segmentation. AlSabah et al. (2018), show that the cultural background of different users, e.g. Arabs, Filipinos, Indians, and Pakistanis, affects the creation of passwords. However, commonalities include incorporating their personal information and choosing memorable, easy-to-guess passwords.

2.3 Password Policies

Password policies are a set of rules that increase security by encouraging users to create strong passwords. The purpose of these policies is to set a standard for creating strong passwords and determine how often they should be changed. Each organization has its own password policy that specifies the requirements for creating passwords, including character length (minimum and maximum number of characters) and character categories (symbols, digits, uppercase, and lowercase letters).

Guo et al. (2019) proposed an advanced new policy called 'Optiwords' to arrange characters on the lines of the figure based on the user's choice such as “09*iKo%rdfGt”

or “t5\$eJnbvFu*9o”, besides including at least three of four character sets with a minimum length of 8. Some services strictly adhere to their password recommendations. Yang et al. (2018) proposed a Dynamic Password Policy Generator (DPPG) that forces the user to change the pattern when creating a password, e.g. include characters 'v', 'Z', avoid characters 'e', 'a', 's', or use the structure 4 'L' (lower-case) followed by 2 'U' (upper-case), then 1 'S' (symbol).

The user often tends to follow policies simplistically by combining lowercase letters with a few digits, followed by a special character, during the password creation process, regardless of their native language. Unfortunately, users often overlook whether the system accepts characters from the alphabet of their language. The common, thoughtless pattern is to use English letters without checking if the system supports characters specific to their native language. A common practice is to adhere to a password length of 6 to 8 characters, with the most common password '123456' of over 10 million users as collected from data breaches. This indicates that current policies are ineffective, and the minimum requirements influence user behaviors, Proposed a new method to store and use password patterns by adding a new phase to password policies, in addition to "Compliant with password policy," "Different from the previous three passwords," and "Different from related password leaks," a step proposed "Different from the previous password behavior". (Güven, 2022).

The user needs to be educated about secure password practices to mitigate the risk of easy attacks. Many online services lack clarity regarding their password policies or requirements. This ambiguity is intentional, as disclosing such policies could aid attackers in deciphering user password patterns. Understanding a service's password policy can significantly reduce the search space for attackers when choosing datasets. It is crucial for users to prioritize strong, unique passwords, and for service providers to implement robust security measures without divulging specifics that could compromise user accounts (Imamaliyev & Khudoykulov, 2021).

2.4 Leaked Passwords

Breaches can unintentionally contribute and become valuable resources for research, particularly in the realm of password analysis. Leaked passwords serve as critical datasets, enabling the examination of user behavior when creating passwords,

conducting security analysis to identify common patterns or trends, and assessing password strengths to set guidelines and recommendations for the user.

Gan et al. (2022) introduce a method to analyze the correlation between words in password and the user's language. Using the Russian dataset 'VK', which was leaked from Russia's largest social networking site, and BreachCompliance, which contains 3.2 billion leaked plaintext passwords, researchers were able to identify Japanese “jp” and Vietnamese “vn” users based on the domain names of their email addresses. AlSabah et al. (2018) analyzed the information of 403.870 customers that was breached in April 2016 from a Middle Eastern bank's database in Qatar. Han et al. (2021) use three different datasets: CSDN from a Chinese website, Rockyou which is the most popular dataset, and 000webhost from an English website, to transfer the grammar from short passwords to guessing long passwords.

In a similar vein, Song et al. (2019) used three datasets (178, 12306, and 7k7k) for testing, and another three datasets (CSDN, Renren, and Rockyou) for training, to compare the proposed password generator (alphapwd) with the leaked passwords sets. Yan & Ku (2023), improved a password guessing method using four Chinese datasets (CSDN, Duowan, Collection#1, and Dodonew), four Japanese datasets (Exploit.in, two Japanese parts of collection#1 and breachcompilation). Collection#1, Exploit.in, and BreachCompilation datasets were also used to collect Korean and other multilanguage passwords, besides the 000webhost and Rockyou datasets.

While leaked data is beneficial in studying password strengths, it also presents concerns, such as the potential misuse in upgrading password guessing algorithms. Murray & Malone (2018) show that a sample of passwords leaked from a particular dataset of an organization will compromise the remaining passwords in this dataset more effectively than samples from different sources. Notably, Arabic-letter passwords were absent in the examined leaked datasets.

2.5 Password Categories

By examining leaked passwords, researchers can evaluate the strengths or weaknesses of passwords within different categories, categorizing passwords based on their characteristics and security attributes. This assessment helps in establishing guidelines and recommendations for users to enhance their password security.

Simple, weak and common passwords are made of one type; numeric values (e.g., “123456”), lowercase letters (e.g., “password”), uppercase letters (e.g., “PASSWORD”), and special characters (e.g., “\$%/_^&*”). Medium-strength passwords are made of two types together: numeric and lowercase letters, numeric and uppercase letters, uppercase and special characters, numeric and special characters, and lowercase and special characters. Strong passwords combine all types and have lengths larger than 8 characters (Sarkar & Nandan, 2022).

MySpace, eHarmony, LinkedIn, and RockYou datasets were used by Shin & Woo (2022), to analyze password strength using tensor decomposition and provide recommendations to strengthen user passwords. Length is computed as the first dominant feature for strong passwords in the MySpace dataset. eHarmony shows that not only length is important but also uppercase letters are critical as the first dominant feature to make passwords stronger. Length is shown to be one of the second dominant features of LinkedIn and Rockyou. So, length appears consistently as the first two dominant features. Having three or more different classes (3class8) is the main contributor, meaning a password comprises three or more different classes among digits, special characters, uppercase letters, and lowercase letters. Han et al. (2021), found that long password-composition policies requiring more segments are more resistant to guessing attacks instead of more character classes.

Cui et al. (2020) proposed a strength evaluation of passwords based on sensitive personal information using segmentation algorithm and the bidirectional matching algorithm. Pereira et al. (2020) measured the accuracy of 13 password strength meters using password guessing resistance against off-the-shelf guessing attacks. The results show that a significant percentage of strong passwords were cracked. Hubbard et al. (2022), propose a novel password-scoring method that builds a custom wordlist by utilizing a person's social media accounts and then comparing the user's password to words found in the person's social media and leaked password databases. Ur et al. (2016) found that users often judge the strength of passwords correctly but generally have misconceptions about the attacker model and the scale of attacks. Pal et al. (2019), noted that attackers can weaponize historical knowledge of users' passwords to infiltrate their accounts. They proposed a personalized password strength meter that rates the specific user's password choice history and pattern. Thai & Tanaka (2024) introduced a new password strength meter that does not rely on heuristic results but is based on

statistical analysis of a large number of leaked passwords from English, Japanese, Russian, and Chinese-speaking users. It is not always easy to determine the nationality of a user from their leaked password. Therefore, passwords are categorized based on the country where the breached data is held or similar metadata associated with the users (Gan, 2022).

2.6 Cracking Hashes

The password hash is generated when a password is entered for comparison with the stored hash. To generate the same hash, the correct password or value is needed. The process of cracking involves attempting to identify that value.

John the Ripper is an offline password-cracking tool that works on leaked password files by taking information from the different fields in the file and applying mangling rules to try passwords. In single cracking mode, extensive mangling rules are applied due to the small input list, so it generates potential passwords. In wordlist mode, a wordlist is used as input, comparing the hash of each word against the password hash. Precomputed hashes used before cracking are represented by the Rainbow table, which maps between a hash and a password and stored in chains. It generates hashes from all possible password values. Some passwords are more probable and easier to crack; this is problematic with context-free grammars (PCFG) that generate guesses from highest to lowest probability, calculated based on the analysis of the structure of the password (alpha strings (L), digit strings (D) and special strings (S)) (Messier, 2021).

Coisel et al. (2017) reduced the search space during the password-guessing process through a two-stage approach. Firstly, a machine learning (ML) algorithm identifies and extracts syntactically meaningful patterns from datasets. Then, in the second stage, these previously segmented parts of passwords are recombined to generate potential passwords. Liu et al. (2018) employed a deep learning model called GENPass for password guessing, utilizing RNN and LSTM combined with PCFG to learn features from datasets, leveraging the fact that many users create passwords with meaningful strings. As Han et al. (2021) transferred knowledge from short passwords to facilitate guessing longer passwords. Jiang et al. (2022) shows passwords made of semantic information could be extracted from the password and then applied to a guessing algorithm based on PCFG. Yan & Ku (2022), Introduced an improved PCFG v4.1 password guessing method that increased the success rate from 2 % to 14 % for

different datasets by adding grammar rules related to Asian language input procedures and pronunciation. He et al. (2023) developed a password reuse model called PassTrans to evaluate its resilience against tweaking attacks, with the result showing that 67.51 % of accounts are vulnerable to breakage under 1000 guesses. Wu et al. (2022) generated a high-quality password dictionary and proposed a temporal convolutional network-based password guessing method (PGTCN) by introducing the Improved Temporal Convolutional Network (ITCN). PGTCN studies the structure and characteristics of passwords automatically and generates new passwords.

The neural network is used to figure out patterns and words in passwords to generate wordlists. Garg & Ahuja (2019) investigated how the PassGAN model would perform when trained on Russian words written in an English layout-based email database. PassGAN didn't perform very well; it wasn't able to understand the structure pattern to create wordlists. Only 6.19 % of passwords from other datasets of a similar nature were matched. Zhou et al. (2024) treated passwords as a sequence of chunks or segments that appear frequently in datasets, using the three datasets: CSDN, RockYou, and Yandex which are Chinese and English passwords. Instead of modeling the relations of chunks, the proposed method selects a seed password from a training set, breaks it into chunks, and then generates new passwords by choosing a chunk and replacing it with another according to their similarity.

2.7 Password Generators

Different password generators employ algorithms and strategies used to generate passwords, resulting in a wide range of user preferences. Some users prioritize usability and memorability over strict security measures, while others prioritize maximum security even at the expense of convenience. Additionally, certain generators strike a balance between usability and security, catering to users who seek both aspects in their passwords. While some users prefer to create their passwords manually, others are open to exploring new methods and strategies offered by password generators. These strategies encompass a spectrum, including random, pronounceable, passphrase, pattern-based, and hybrid password generators, each offering unique benefits and trade-offs.

2.7.1 Random Password Generators

Random passwords are generated using algorithms that select truly random sequences of characters from a defined set, typically including uppercase letters, lowercase letters, numbers, and special characters (e.g., 'jF75%x\$Hh'). This ensures that the generated passwords are as secure as possible. These generators may offer options for customizing the length and complexity of the passwords. Users often perceive random passwords as less usable due to their complexity, but they are widely used in high-security systems, such as military computers. They are also commonly employed for one-time authentication or as temporary passwords when users reset their passwords, ensuring immediate security. Although most users may not prefer random passwords for everyday use, they are often utilized in password managers, where users can store and access passwords without the need to type them each time (R. L. Brotman, 1985).

2.7.1.1 ALPHANUM

Generate a 6-character password containing a mix of uppercase letters, lowercase letters, and numbers (e.g., EvG6p9). Each character in the password has 26 possibilities of uppercase letters, 26 for lowercase letters, and 10 for numbers, totaling 62 possibilities per character. Therefore, the total password set, P_a , is calculated as $P_a = 62^6 = 2^{35.7}$.

2.7.1.2 Programmed Random Password Generators

Different programming languages offer various methods for creating random password generator software applications. Each language has several methods to accomplish this task of creating a combination of letters, numbers, and special characters. Kubariev et al. (2023) conducted research to determine the most suitable method among three languages.

For the object-oriented programming language C#, the class 'Random' is utilized to select characters randomly from a specified ASCII range. This allows for the inclusion of mixed-case letters, numbers, and special characters, with the flexibility to set additional rules for generating strong passwords (e.g., beginning with a symbol). In Java, 'SecureRandom' is employed to generate random integers, long integers within a

given range, doubles or floats, or byte streams. In Python, the module 'Secrets' is utilized, providing high-quality cryptographically secure random numbers.

The random password generators discussed typically generate passwords consisting of English letters; these passwords are completely made of random characters, which are hard to remember. ALPHANUM generates passwords made of uppercase and lowercase letters, and numbers, but it is better to add special characters to the password. However, the three generator applications generate passwords made of uppercase and lowercase letters, numbers, and special characters. According to Leonhard & Venkatakrishnan (2007), ALPHANUM has $26 + 26 + 10 = 62$ possibilities for the first character, and so on, depending on the length of the password. For a 6-characters length, it is $62^6 = 2^{35.7}$. Thus, Pa has 35.7 bits of entropy. Kubariev et al. (2023) used the “Password Strength Meter” to compare the strength of passwords in terms of centuries by applying K-means algorithm and EM-clustering algorithm, and found that the SecureRandom class in Java gives a reliable quality of random passwords.

The primary advantage of random passwords lies in their heightened security. Due to their random nature, attackers are faced with the daunting task of sifting through a vast number of potential passwords, significantly increasing the time and effort required to compromise an account. While random passwords can be challenging to memorize due to their lack of pattern or meaning, this necessitates the use of a password manager or keeping a written record in a secure location.

Moreover, the use of random passwords facilitates the practice of employing different passwords for various systems or services. In situations where accessing a password manager or written records is not feasible, having distinct passwords for different accounts can provide an additional layer of security, at the expense of usability. This places the user in a difficult situation when the password is needed.

2.7.2 Pronounceable Password Generators

Generate passwords phonetically pronounceable, aiming to make it easier for the user to remember while it looks random and relatively strong. Algorithms are employed to craft combinations of syllables or words that mimic natural speech patterns.

Produce passwords that look random but are phonetically pronounceable in English. The PRONOUNCE3 generator leverages the human mind's speech faculties to

aid memorization. It constructs passwords using a selection of 22 consonants (including "ch," "ph," and "st") and 5 vowels, with two rules restricted to ensure consistency with English spelling. First, no passwords may begin or end with two consonants. Secondly, passwords may not contain three consecutive consonants or three consecutive vowels.

Applying these restrictions yields 13 templates ($T_{v,c}$) when using 4 vowels (v) and 4 consonants (c). The generated passwords are in the English language, made of lowercase letters only. They have no meaning, but sound like something pronounceable to the user, which may be confusing between certain sounds such as 'sh' and "ch". According to Leonhard & Venkatakrishnan (2007), the set of templates for $T_{4,4}$ comprises 13 elements (e.g., vvcvccv). By randomly selecting one template, each vowel and consonant is equally chosen, ensuring randomness in password generation. The resulting password set $P_{v,c}$ has an entropy of $2^{30.8}$, with a maximum of $2^{31.5}$, equating to 30.8 or 31.5 bits of entropy, respectively.

A notable flaw in pronounceable password generators, as identified by Ganesan and Davies (1994), lies in selecting syllables based on frequency in English writing, leading to some passwords being more likely to be chosen. The PRONOUNCE3 scheme, however, avoids this flaw.

2.7.3 Passphrase Password Generators

Generate passwords by combining random phrases or words. These passwords are longer and easier to remember than random passwords, as they create mental connections.

DICEWARE generates passwords by combining random phrases or words, taking advantage of the mental connections users have with words. These passwords, known as "passphrases," are longer and easier to remember compared to random passwords. Reinhold et al. (2006) provide a generator with a list of 7,776 English words, each assigned to a unique five-digit number. By rolling the dice five times, the user obtains a number that guides to the first word of the password and so on, resulting in a password composed of words depending on the user's chosen length (e.g., "kraut gwen nagoya"). The goal is to use words that differ from those typically used by users to create passwords.

A set of English words is generated to be used as a password. Some words are obscure or too short in the list, and the proposed list is too small. According to Leonhard & Venkatakrishnan (2007), the possibility of a three-word password is 7776^3 , resulting in $2^{38.8}$ possibilities and 38.8 bits of entropy for the password set, Pd.

In their experiment, Leonhard & Venkatakrishnan (2007) found that while ALPHANUM may be helpful with password managers, DICEWARE is better suited if obscure words are removed from the list. PRONOUNCE3, with the addition of capital letters and special characters, helps to increase entropy. Participants preferred the PRONOUNCE3 scheme slightly more than ALPHANUM and DICEWARE. When the user chooses to have a password of three words from the wordlist in DICEWARE, the entropy reaches 38.8 bits, making it stronger than ALPHANUM and PRONOUNCE3.

2.7.4 Pattern-Based Password Generators

Pattern-based password generators create passwords using predefined patterns or rules, such as keyboard patterns like "qwerty". They may combine words from a dictionary, substitute certain letters with numbers or symbols (e.g., "p@s\$w0rd"), or incorporate personal information. These generators strive to strike a balance between usability, memorability, and security, often depending on user inputs or the shapes and patterns on the keyboard.

2.7.4.1 Password Generation Based On User Inputs

Glory et al. (2019), developed a password generator that combines user-provided texts and numbers to create up to ten passwords. The user is asked to provide five input texts and two numbers, which are used to generate a password like "81KathY%m@ngO|". Two words are randomly chosen, with one number mixed with symbols and capitalized letters. The system also incorporates known replacements of letters with visually similar special characters, such as "a" with "@", and "o" with "0". Additionally, if the user-inputted words are too short to meet the password policy, at least three random special characters are added.

The user must be cautious when choosing words and numbers, avoiding easily guessable information such as names, pet names, birthdates, and details shared on social media. Social engineering attacks could potentially crack these passwords by extracting

meaningful words, especially considering the known letter-to-symbol replacement method.

Mostly, the user will enter familiar words in different passwords. Targuess-II+, as described by Zhang et al. (2023), employs transformation rules based on structure and segments that delete, swap, and insert segments or characters, which enables faster online password guessing. Effective password-guessing techniques often utilize semantic information mining based on probabilistic models (Pal, 2019; Cui, 2020) or machine learning (Hubbard, 2022; Thai & Tanaka, 2024; Pereira, 2020). Li et al. (2023), introduced MDBSCAN (Modified Density-Based Spatial Clustering of Applications with Noise), a novel density-based clustering approach in machine learning that builds an automatic password mangling rule generator.

2.7.5 Hybrid Password Generators

These generators combine multiple strategies to generate passwords, aiming to strike a balance between security and memorability. They may generate a passphrase with mixed random words or characters.

2.7.5.1 VowPass

Alhamed & Bhatia (2021) introduce a method for generating secure and memorable passwords. This method involves combining a user-chosen number and a secret word, which are then hashed. After hashing, the first four digits of the hash are followed by a symbol assigned to a category of a service (e.g. \$ = finance), followed by the service name and the version of the password (e.g. v1,v2), with a symbol assigned in between. To enhance unpredictability, all vowels are removed from the secret word and service name, and the user is free to capitalize any letter. Alternatively, if the user prefers not to use hashing, a simpler mechanism is available. In this case, the secret word and service name, without vowels, are combined with symbols assigned and the version of the password (e.g., flwrs&CRs&v2).

The user needs to remember a number and a secret word to regenerate the hash of them to recover the first four digits, as the hashing method (Message Digest (MD5), Secure Hash (SHA1)) is chosen by the user. The resulting password will look like “6E3B&crs&V1”. The user must have the ability to regenerate the hash of this combination at any time and anywhere, in addition to remembering four random digits and letters. Furthermore, the user must remember the symbol assigned for each

category, with 40 symbols available. If the user wants an easy way to assign symbols related to services such as \$ finance, @ email, # Twitter or Snapchat, % online shopping, and ! Unimportant services, this might be vulnerable to attackers who are aware of this method and can guess the symbols assigned to each category. The other way is to assign unexpected symbols to services as \$ for games and @ for online shopping.

In addition to remembering the number, secret word, and assigned symbols for each account or service, the user must also remember the last four digits of the hash when it is hard to regenerate them, as well as the version (v2) of the password, which needs to be updated each time it changes. Whether the user is saving passwords on a computer or writing them on paper, there is a lot of information to remember, which may present usability challenges.

The alternative mechanism proposed, without using the hashing function and relying solely on the secret word without vowels, could potentially be cracked using a dictionary without vowels, further adding to the memory burden for the user. While this method may be strong from a security standpoint, it presents usability challenges.

2.7.5.2 PassMan

An approach for generating and managing passwords proposed by Billa et al. (2019) eliminates the need for passwords to be stored locally or in a cloud database. Instead, they are generated by hashing (MD5) the username and a phrase (quote or any alphanumeric value) chosen by the user, which is saved in local storage, and hashing (MD5) a hint about the system. The hint can be a word, number, or sentence representing the system. These two hashes are then hashed (MD5) again, trimmed, and the last 11 characters are combined with digits, lowercase letters, uppercase letters, and special characters to create a password with a length of 16 characters.

To generate different passwords, the hint for each service or account varies. In guest mode, the user needs to provide a username, phrase, and hint to generate a password without registering or signing up.

Using another device that doesn't have access to the system storing the two values and the hint, the user must remember their username, application password, phrase, and the hint for each service to be hashed. Then, they must hash these two hashes again to

generate the password. The user has to remember different hints for each service and must remember the username, application password, and phrase to log in. This approach may not be practical when the user needs to access the password urgently, as they would require access to the device where the values are stored.

2.8 Mnemonic password strategy

A strategy that involves creating passwords based on memorable associations, leveraging the human brain's ability to recall information more easily when it is presented in a meaningful context. For example, Optiwords helps users design separate line drawings on the keyboard as a password figure.

2.8.1 Alphapwd

A password generation strategy is based on mnemonic shape, achieved by combining the order of writing strokes of letters. This method simulates the writing process of a character on the keyboard, allowing users to create passwords where the stroke order of the letters in a word, or in some letters of the word, determines the password. For example, the letter “H” can be replaced by “UHBijnhj”. The user has the option to use uppercase or lowercase letters, digits, or symbols by pressing the “shift” key, resulting in a password that appears as a random string (Song, 2019).

2.8.2 AvoidPwd

A novel mnemonic password generation method is based on keyboard transformation. The user customizes the “route” to bypass an “obstacle” and selects the characters on the route as the final password. The “obstacle” is a word chosen by the user in any language, and the keys adjacent to the “obstacle” are typed with the “Shift” key pressed. (Lyu,2022)

The arrangement of keyboard keys may vary between companies, especially concerning the placement of symbols. Some companies aim to create smaller devices or keyboards that better suit the position of users’ hands, whether for gaming or typing faster. Therefore, the user must remember which keys were pressed with the “Shift” key and which were not.

A common method of creating passwords is to choose a pattern of nearby keys, which can be adapted to any language depending on the user’s preference. Tsokkis and Stavrou (2018) identify and combine a range of poor password construction strategies

and design a tool to practically demonstrate the strategies to the user. One of the participants chose a strategy in password generation which was a key combination after her name Jessica + a small triangle (aws) = Jessicaaws, which was found in the generated list of potential passwords.

However, if keyboard keys change from one device to another, the user may forget the route, as it heavily relies on the order of keys on the keyboard. Additionally, Android or smartphone devices have different ways of representing symbols than computer keyboards. The user needs to remember the word chosen for each service and the route shape, as well as which keys are chosen from the surrounding area of the obstacles. The user might choose letters next to each other in a straight line, which can easily be cracked.

2.9 Identification of Gaps in the Literature

Research on passwords has largely focused on the global north, where English is often the first or only language. Studies on password generators, password strength meters, and password cracking have primarily centered around English-letter passwords. Moreover, multilingual users typically base their passwords on the language of instructions and the dominant language of interaction with technology, which is often English. However, some leaked password datasets contain a small number of non-English letter passwords as shown in table 2.1. Additionally, some sites or services allow the use of non-English letters in passwords or just require the inclusion of lowercase or uppercase letters.

One notable research study by Doucek et al. (2020) explored password strength estimation for Czech and Slovak passwords by incorporating Czech and Slovak dictionaries into ‘zxcvbn’. However, upon examining the dataset, it was found to contain non-English words composed of English letters, lacking Czech or Slovak letters. This approach could be beneficial, given the similarity between the European alphabet and Czech or Slovak alphabet letters.

Another study by Yan & Ku (2023) enhanced PCFG v4.1 by incorporating pronunciation rules of Traditional Chinese, Simplified Chinese, Japanese, and Korean. However, these datasets primarily contain English letters but non-English words.

Table 2.1: Non-English passwords leaked

Language	Sample of Passwords	Number of passwords
French	Été12345 Février2016 Maîtriser1	19
German	Frühling2017 März123 Fußball1	10
Portuguese	Dragão12 Dragão2017	7
Russian	Зима1234 Весна123 Лето2017 Февраль17 Ялюблютебя2016	154
Spanish	Contraseña1 Otoño2017 Dragón2017	18

There is a significant gap in research regarding passwords using non-English letters, as most of the leaked password datasets contain English letters. The mother-tongue language influences password creation, especially for users with a limited English vocabulary. They may resort to using their name, numbers, keyboard patterns, or weak words they know. Voyiatzis et al. (2011) studied Greek websites and found that some users prefer passwords containing Greek characters. Combining substrings from different languages has the potential to increase the search space and make passwords more random and difficult to guess, as suggested by Wang et al. (2015). This approach was also promoted by Maoneke et al. (2020), who advocated for using a multilingual passphrase policy to enhance security by increasing the search space. Users who prefer visual memorizing often use Mnemonic password strategies, which rely on the order of keyboard keys, and can be applied to any language's letters.

Additionally, an uncovered aspect is the use of right-to-left languages in passwords, such as Arabic, Persian, Urdu, and Hebrew, which are spoken by millions of people. Arabic is the fifth most spoken language; 22 countries use Arabic as the official spoken language, and each country has an undefined number of dialects in which the vocabulary may vary. The basic Arabic alphabet exists in non-Arab countries such as

Persian (Farsi), Urdu, Pashto, Kurdish (Sorani), Sindhi, Balochi, Kashmiri (Perso-Arabic), Azerbaijani, and Malay (Jawi). There are some differences in these alphabets compared to Arabic alphabet, such as;

Persian: ا ب پ ت ث ج چ ح خ د ذ ر ز ژ س ش ص ض ط ظ ع غ ف ق ک گ ل م ن و ه ی

Urdu: ا ب پ ت ث ج چ ح خ د ڈ ذ ر ژ س ش ص ض ط ظ ع غ ف ق ک گ ل م ن و ہ ے ی

The number of these users is nearly 660 million people, so ensuring the security of this large population to protect their information and privacy is essential. This large number of people attracts businesses to build trust and confidence with them in online activities. While creating and using Latin passwords may not be easy for them, accepting and supporting their official alphabets by service providers would be much easier and more secure. This leads to the support and acceptance of all worldwide alphabets in passwords.

2.10 Summary

Random passwords are often too complex for users to remember, leading to reliance on password managers or an increased risk of forgotten passwords. Additionally, they can be difficult to input or type, especially if they include special characters or symbols. Furthermore, random passwords are typically not phonetically pronounceable, making it challenging for users to verbally communicate or remember them. While pronounceable passwords offer improved usability, they may have lower security compared to randomly generated passwords, as attackers could exploit pronunciation patterns. Limited character set used, reducing the entropy and overall strength of the passwords.

Passphrase passwords are generally longer than traditional passwords and can still be vulnerable if they are too short or if the words chosen are common. Passphrases composed of dictionary words are particularly susceptible to dictionary attacks, especially if they lack variation or complexity. Additionally, longer passphrases may be difficult to remember, leading users to resort to less secure practices like writing them down.

Pattern-based passwords may be more predictable and vulnerable to guessing attacks, especially if the pattern is widely known or easily guessed. This lack of randomness can result in lower entropy compared to truly random passwords, reducing

overall security. Generators that incorporate personal information may increase the risk of password compromise if the information is easily accessible or guessable. Hybrid passwords that combine multiple strategies may increase the complexity of generated passwords, potentially making them difficult to remember or type. Finding the right balance between security and usability can be challenging, as overly complex passwords may sacrifice memorability, while overly simple passwords may sacrifice security. Mnemonic passwords rely entirely on the arrangement of keyboard keys and chosen shapes without any meaningful words, resulting in random characters but easier-to-remember passwords.

Password authentication is one of the most commonly used methods to secure access. Various security threats such as brute force, dictionary, and guessing attacks; are becoming faster and easier by employing increasingly sophisticated techniques to compromise passwords and gain unauthorized access. These attacks leverage the power of computational resources and trained datasets to crack passwords, exploiting common patterns, weak choices, and predictable combinations.

CHAPTER 3

EQUIPMENT AND METHOD

3.1 Introduction

Logging into an account, device, or service typically involves using a password created by the user, which must be memorized or saved. Alternatively, users may utilize face ID or fingerprint recognition, if supported by the device or service, as an alternative to using a password.

Most services do not store users' face ID or fingerprint data without first asking the user to set a password or pattern. This password or pattern is then required for authentication in situations where face ID or fingerprint recognition is unavailable, after a period of inactivity, or when accessing the account from a different device. Replacing passwords can be costly, particularly in cases of lost tokens or account recovery efforts.

Passwords remain widely necessary for authentication, whether for emails, accounts, services, or access devices, especially in cases where face ID or fingerprint authentication is not available. Users require passwords that are memorable and easily retrievable from anywhere, and it's essential not to reuse the same password across multiple accounts. Despite these challenges, passwords continue to offer a good balance of deployment and usability in various authentication scenarios.

While numerous strategies exist for creating and generating passwords using Latin letters, there is a need for innovative approaches to enhance security. One potential avenue is to leverage right-to-left languages, which offer new possibilities for password creation.

3.2 Service Providers and Users

There are three types of service providers based on password acceptance, as known due to the lack of this kind of research, illustrated in Table 3.1. First, some service providers do not support any non-Latin characters, such as Gmail and Hotmail. Second, some service providers accept alphabet characters from any language to be used in the password, but require the inclusion of at least one lowercase or uppercase letter in addition to numbers and symbols. These four character sets are dominant in the

password requirements of many service providers, such as GitHub. Third, some service providers accept any kind of characters, such as Facebook. The tool developed in this thesis is particularly useful for password creation for the second and third types of service providers.

Table 3.1: Non-Latin acceptance comparison

Service Provider	Non-Latin password acceptance			Accepted Password
	Yes	No	Provisional	
Gmail		✓		Latin password
Hotmail		✓		Latin password
Yahoo		✓		Latin password
GitHub			✓	ر2بعاڤل
Twitter (X)			✓	ر2بعاڤل
Amazon	✓			ر2بعاڤل
Facebook	✓			ر2بعاڤل
Instagram	✓			ر2بعاڤل

Giving the user freedom to choose characters from his native alphabet in password creation will increase the strength of passwords. The user must think outside the box and not follow common trends in password creation. When a multilingual password is supported by service providers, using non-Latin letters in passwords can include any language preferred by the user, such as mixing three or more different language alphabets. This research develops an AI-based tool that generates passwords using both Latin characters and Arabic characters, in addition to numbers and symbols. The tool is targeted at users who speak or read Arabic. There is a need for the availability of an Arabic keyboard or something equivalent.

3.3 Key features of the developed tool:

To set a password or sign up for a service, it is important to first understand the password requirements or policies of the service providers. These requirements may vary, and even if they seem familiar as known from most service providers, it is crucial to avoid reusing passwords across multiple services, as this is considered a poor security practice. Most service providers do not mention these requirements clearly for security reasons. The user must try an Arabic-characters password to figure out if the service providers support non-Latin characters in passwords. The password requirements

extractor ancillary tool provides a clear view of the requirements to the user as illustrated by Figure 3.1.

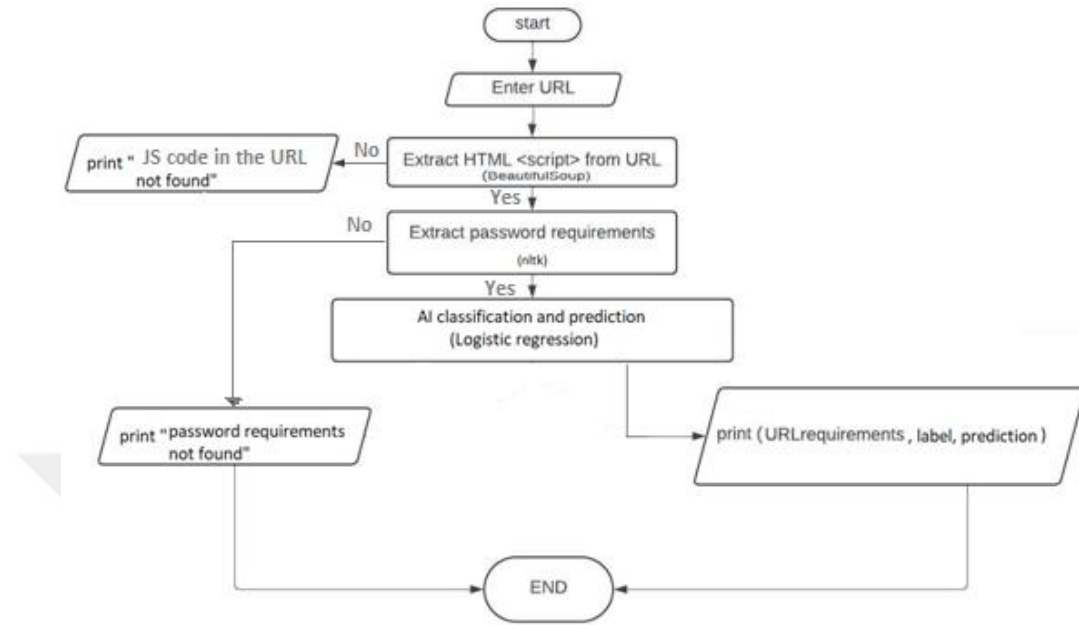


Figure 3.1: Password requirements extractor flowchart

The algorithm of the password requirements extractor is as follows:

- The user submits the Uniform Resource Locator (URL).
- The content of the webpage is extracted, by extracting HTML or JS from the URL. Then, Employing BeautifulSoup from the Python library to extract the <script>, if found, or print "JS not found".
- Search for the requirements by the AI module nltk pre-trained on English language in the extracted content if found, or print "password requirements not found".
- Classify the found requirements and predict rejection of using Arabic letters, by the logistic regression trained model.
- If the requirements found print the results, labels, and the prediction percentage of not accepting the Arabic letter.

By knowing the password requirements of the service providers and their acceptance of non-Latin characters, the user can decide whether to use an Arabic-character password or a Latin-character password. The user must enter the base word using Arabic characters; a quick search is done to avoid popular words due to the lack of

leaked Arabic-character passwords. Then, two passwords (an Arabic-character password and a Latin-character password) are generated based on the user's input. The generator is working as shown in figure 3.2.

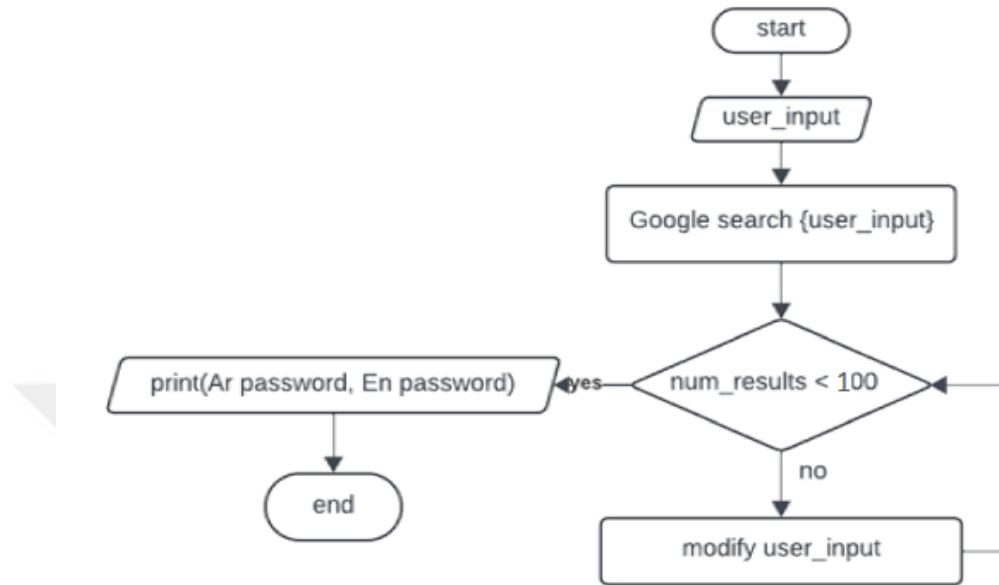


Figure 3.2: Password Generator Flowchart

The algorithm of the password generator is as follows:

- The user chooses and types the base word of the password.
- Search the popularity of the word iteratively for each generated password on the internet.
- If the number of searching results is more than the threshold the password is modified. When it is less than the threshold both passwords are generated.
- Generate the Arabic password from the base word.
- Generate the Latin password from the base word.
- Print both passwords.

3.4 Password Requirements Extractor

This study focused on password containing Arabic alphabets mixed with numbers and symbols, which is not popular. The user is not required to try the Arabic-letter password to know if the service provider accepts it. A password requirements extractor

was developed using the Flask framework to provide an interface for the user to submit the URL of the service provider, to extract the script of the service provider and search for the password requirements, then identify whether the Arabic-letter password is accepted by using the logistic regression trained model. The extractor provides a clear vision for the user about the password requirements. Extracting the HyperText Markup Language (HTML) or JS by submitting the URL can help the user by several steps; extracting JS code, searching for specific sentences (requirements) in JS code, then classifying and predicting the acceptance as illustrated in Figure 3.3.

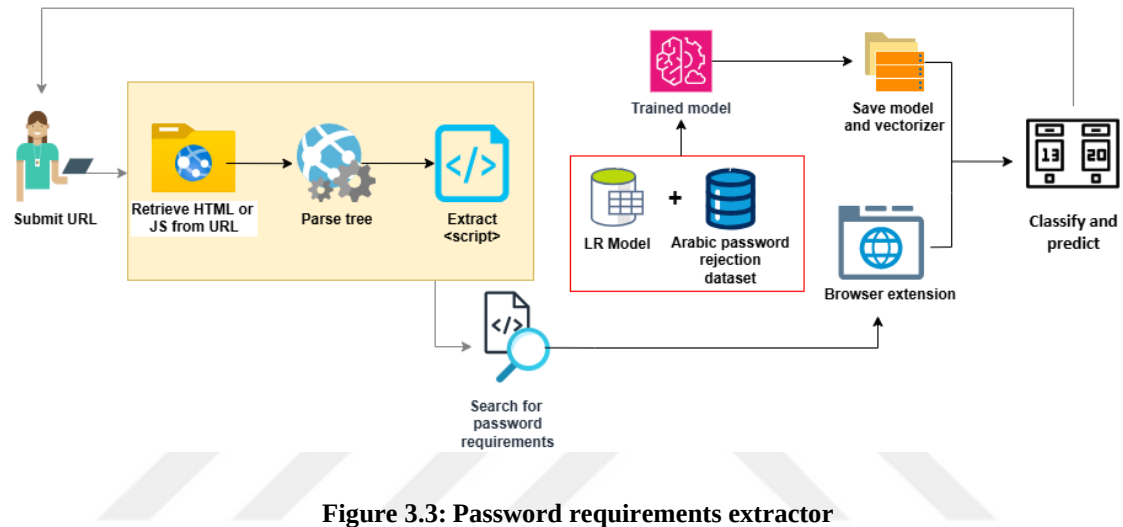


Figure 3.3: Password requirements extractor

3.4.1 Webpage Content Extraction

The tool extracts the script from the sign-up page to assess the password requirements defined by the service provider. This extraction process is performed on JS codes. It begins by employing the function `extract_js_from_url(url)` to get the website's HTML and JS code into a tree of Python objects, such as tags, navigable strings, or comments. Which makes it easier to interact with the website's content and make it more accessible; to extract the desired password requirements by parsing structured data. Utilizing the Python 'requests' library, which fetches the content of the webpage, the command `requests.get(url)` retrieves and stores the HTML or JS data that the server sends back, depending whether the website is static or dynamic. Figure 3.4 shows a part of the full extracted HTML script. Figure 3.5 shows employing the BeautifulSoup library, which parses the document from the parsed HTML or JS content, then extracts the codes embedded within `<script>` tags to obtain the password requirements using the method `find_all()`.


```

# Function to extract JavaScript code from a URL
def extract_js_from_url(url):
    try:
        # Fetch the content of the webpage
        response = requests.get(url)
        response.raise_for_status()

        # Extract JavaScript code using BeautifulSoup
        soup = BeautifulSoup(response.content, 'html.parser')
    or .text
        js_code = [script.text for script in
soup.find_all('script')]

        return js_code

    except requests.exceptions.HTTPError as http_err:
        print(f"HTTP error occurred: {http_err}")
    except Exception as err:
        print(f"An error occurred: {err}")

```

Figure 3.6: `extract_js_from_url` function

3.4.2 Search for Specific Sentences in JS Code

To make the password requirements extractor work faster, the extracted JS code provides the requirements in a common sentence that the user must follow during password creation. By employing the function `search_specific_sentences` (`js_code`) aiming to identify certain sentences within the JS code that contain specific keywords `["common", "lower", "lowercase", "lower-case", "upper", "uppercase", "upper-case", "english", "only use", "use only", "at least", "at most", "lower_case", "upper_case", "letters", "symbols", "special characters", "special-characters", "special_characters", "mixed case", "mixed-case", "mixed_case", "must contain", "should contain", "cannot contain", "should include", "must include", "cannot include", "password", "only contain"]`. These might be words commonly associated with requirements, such as "use the common alphabet or symbols", "your password must have a mix of lower and upper letters", "only use English letters", "your password must have at least two lowercase letters", or "your password must have at most one uppercase letter". The list of sentences found in JS code is then stored in `found_sentences = []`. Figure 3.7, shows the function `search_specific_sentences` that iterates to search through each line in the script.

```

# Function to search for specific words in JavaScript code and return
full sentences
def search_specific_sentences(js_code):
    specific_words = ["common", "lower", "lowercase", "lower-case",
"upper", "uppercase", "upper-case", "english", "only use", "use only",
"at least", "at most", .....]
    found_sentences = []
    for code in js_code:
        sentences = sent_tokenize(code) # Tokenize code into sentences
        for sentence in sentences:
            for word in specific_words:
                if word in sentence:
                    found_sentences.append(sentence)
    for code in js_code:
        sentences = code.split(';') # Split code into sentences
        for sentence in sentences:
            for word in specific_words:
                if word in sentence:
                    found_sentences.append(sentence.strip())
    return found_sentences

```

Figure 3.7: search_specific_sentences function

The 'search_specific_sentences' function iterates through each code snippet 'for code in js_code'. Each JS code is tokenized into sentences by sent_tokenize from the nltk.tokenize module, which includes a pre-trained Punkt tokenizer for the English language. Nested loops 'for sentence in sentences', and 'for word in specific_words' are used for each sentence in the tokenized code snippet to check if the requirements keywords appear, and append the sentence to the found_sentences list. An additional approach uses code.split(';') to split the code into sentences based on semicolons (;) which are used at the end of each line of code in JS. Then, similar to the previous iteration, the function iterates over each sentence to append it to the found_sentences list. Figure 3.8 shows the output of the function search_specific_sentences(js_code) after extracting the password requirements.

```

Enter the URL: https://login.yahoo.com/account/create?lang=en-US&src=ym&done=https%3A%2F%2Fmail.yahoo.com%2F%3Fguce\_referrer%3DaHR0

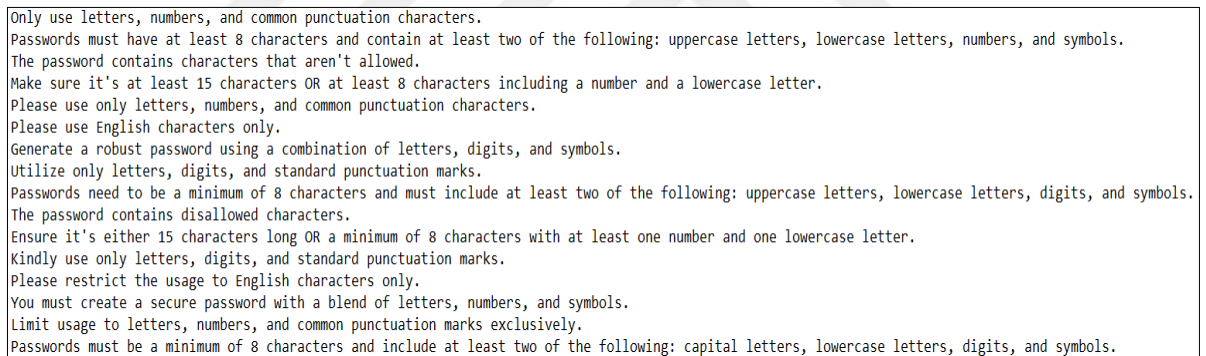
Found the following sentences in the JavaScript code:
", "SOME_SPECIAL_CHARACTERS_NOT_ALLOWED": "You can only use letters, numbers, periods ('.'), and underscores ('_') in your username.
", "SOME_SPECIAL_CHARACTERS_NOT_ALLOWED": "You can only use letters, numbers, periods ('.'), and underscores ('_') in your username.
", "PASSWORD_TOO_SHORT": "Must contain at least 8 characters.
", "INVALID_CHARACTERS_IN_PASSWORD": "Can only contain letters, numbers and , !
", "NEED_AT_LEAST_ONE_ALPHA": "Please use at least one letter in your username.

```

Figure 3.8: Example of found sentences (requirements) list

3.4.3 Dataset Preparation

The Arabic-character password rejection dataset was not found on the internet, nor were any similar datasets containing sentences of password constraints or requirements. Therefore, an AI language model was used to generate a list of password requirement sentences by feeding it real requirements taken from websites' sign-up page. The dataset "arbarb.txt" contains sentences ending with (.) describing how service providers would reject non-English passwords. The main purpose of the password requirements extractor is to help the user to know if an English letter must be added. Common requirements, such as " use English characters", or "include at least two of the following: uppercase letters, lowercase letters" often indicate that the password must be composed of English characters only or must contain uppercase or lowercase letters, which are not features of the Arabic alphabet. The dataset focuses on the type of characters required, in addition to other requirements such as numbers and symbols, to assist the user. Figure 3.9 shows a sample of the dataset used.



Only use letters, numbers, and common punctuation characters.
Passwords must have at least 8 characters and contain at least two of the following: uppercase letters, lowercase letters, numbers, and symbols.
The password contains characters that aren't allowed.
Make sure it's at least 15 characters OR at least 8 characters including a number and a lowercase letter.
Please use only letters, numbers, and common punctuation characters.
Please use English characters only.
Generate a robust password using a combination of letters, digits, and symbols.
Utilize only letters, digits, and standard punctuation marks.
Passwords need to be a minimum of 8 characters and must include at least two of the following: uppercase letters, lowercase letters, digits, and symbols.
The password contains disallowed characters.
Ensure it's either 15 characters long OR a minimum of 8 characters with at least one number and one lowercase letter.
Kindly use only letters, digits, and standard punctuation marks.
Please restrict the usage to English characters only.
You must create a secure password with a blend of letters, numbers, and symbols.
Limit usage to letters, numbers, and common punctuation marks exclusively.
Passwords must be a minimum of 8 characters and include at least two of the following: capital letters, lowercase letters, digits, and symbols.

Figure 3.9: Sample of the dataset 'arbarb.txt'

After getting a text file "arbarb.txt" that has requirement sentences, the code in figure 3.10 was used to convert the text file into a .json file "labeled_arbarbup.json" that is shown in appendix A.1, with a label for each requirement in the dataset as shown in figure 3.11.

```

import json
def txt_to_json(input_file, output_file):
    try:
        # Read the .txt file
        with open(input_file, 'r', encoding='utf-8') as file:
            sentences = file.read().split('.')

        # Remove any empty strings or trailing whitespace
        sentences = [sentence.strip() for sentence in sentences if
sentence.strip()]

        # Label each sentence as 0 or 1 (default: all 0)
        labeled_sentences = [{"sentence": sentence, "label": 0} for
sentence in sentences]

        # Save as a JSON file
        with open(output_file, 'w', encoding='utf-8') as json_file:
            json.dump(labeled_sentences, json_file,
ensure_ascii=False, indent=4)

        print(f"Converted {len(sentences)} sentences to JSON and saved
in '{output_file}'.")
    except Exception as e:
        print(f"An error occurred: {e}")

# Usage
input_txt = "arbarb.txt" # Replace with your input file name
output_json = "labeled_arbarbup.json" # Replace with your desired
output file name
txt_to_json(input_txt, output_json)

```

Figure 3.10 : The code used to convert text file into json file

```

import json

def txt_to_json(input_file, output_file):
    try:
        # Read the .txt file
        with open(input_file, 'r', encoding='utf-8') as file:
            sentences = file.read().split('.')

        # Remove any empty strings or trailing whitespace
        sentences = [sentence.strip() for sentence in sentences if
sentence.strip()]

        # Label each sentence as 0 or 1 (default: all 0)
        labeled_sentences = [{"sentence": sentence, "label": 0} for sentence in sentence

        # Save as a JSON file
        with open(output_file, 'w', encoding='utf-8') as json_file:
            json.dump(labeled_sentences, json_file, ensure_ascii=False, indent=4)

        print(f"Converted {len(sentences)} sentences to JSON and saved in '{output_file}'

    except Exception as e:
        print(f"An error occurred: {e}")

# Usage
input_txt = "arbarb2.txt" # Replace with your input file name
output_json = "labeled_arbarbup.json" # Replace with your desired output file name
txt_to_json(input_txt, output_json)

```

```

1 [
2   {
3     "sentence": "Only use letters, numbers, and common punctuation
4     "label": 0
5   },
6   {
7     "sentence": "Passwords must have at least 8 characters and cont
8     "label": 0
9   },
10  {
11    "sentence": "The password contains characters that aren't allow
12    "label": 0
13  },
14  {
15    "sentence": "Make sure it's at least 15 characters OR at least
16    "label": 0
17  },
18  {
19    "sentence": "Please use only letters, numbers, and common punct
20    "label": 0
21  },
22  {
23    "sentence": "Please use English characters only",
24    "label": 0
25  },
26  {
27    "sentence": "Generate a robust password using a combination of
28    "label": 0

```

Figure 3.11: Running the code to convert .txt to .json

The common sentence formulations that are taken from real-life scenarios when testing Arabic password; are "The password contains characters that aren't allowed.",

this directly informs the user that non-English characters are not allowed, "contain at least two of the following: uppercase letters, lowercase letters", by mentioning "upper and lower" it means that English letters are mandatory, or "Only use letters, numbers, and common punctuation characters.", in most cases, Arabic letters are treated as special characters, which indicates that they are not supported. All these types of requirements are labeled "1" (password requirements). Other requirements related to policies for the username, birthdate, email, or hints for creating strong passwords like "DOMAIN_NOT_SUPPORTED: This domain is not supported.", "Commonly known weak passwords are not accepted (password123, etc.).", or "You can only use letters, numbers, periods ('.'), and underscores ('_') in your username." labeled "0" (General policy). Labeling the "labeled_arbarbup.json" was done manually, as the figure 3.12 illustrates a sample of the dataset and the shape of it, by reading the JSON file into a Pandas DataFrame.

```
[7] import pandas as pd
import seaborn as sns

# Load the JSON file into a Pandas DataFrame
df = pd.read_json('/content/labeled_arbarbup.json')

[8] # Display the first few rows of the dataset
print(df.head())
```

	text	label
0	Only use letters, numbers, and common punctuat...	1
1	Passwords must have at least 8 characters and ...	1
2	INVALID_IDENTIFIER:Error: Invalid identifier.	0
3	The quick brown fox jumps over the lazy dog.	0
4	I enjoy reading books during my free time.	0

```
[9] # Display the shape of the dataset
print(df.shape)
```

(303, 2)

Figure 3.12 : Sample of the dataset "labeled_arbarbup.json"

3.4.4 Logistic Regression

A supervised machine learning algorithm used for classification tasks and performs well with small datasets. In this case the algorithm predicts the probability of

an input (the requirement sentence) belonging to a specific class. Each sentence can belong to one of the two possible classes "0" which represent "General policy" or "1" which represent "Password requirement". Training the logistic regression model begins by importing the necessary libraries from **sklearn**. The following steps and libraries are: Importing **TfidfVectorizer** that convert a raw text into numerical feature matrix using TF-IDF method. Importing **LogisticRegression** to implement regularized logistic regression using the 'liblinear' library, this is a statistical method for binary classification. Importing **accuracy_score** to evaluate the performance of the classification model by computing the subset accuracy. If the predicted label set for a sample strictly matches the true set of labels, the subset accuracy is 1.0; otherwise it is 0.0. Importing **classification_report** that provides detailed classification metrics. Importing **train_test_split** which will split the dataset into random training and validation subsets.

First, the dataset `labeled_arbarbup.json` is opened and loaded into the variable `labeled_data` using `json.load(file)` after reading its content. The values of 'text' and 'label' are extracted from each `entry` and stored in the lists `data` and `labels`. ML models such as Logistic Regression, require input features 'data' and target outputs 'labels'. By separating each raw text and their corresponding labels, the preprocessing step for training and testing the model becomes simpler. Then, splitting the dataset into training data with the size of 80% to build the model and validation data with the size of 20% to evaluate performance. Figure 3.13 shows the code for loading and splitting the dataset.

```
# Load the labeled dataset
with open('/content/labeled_arbarbup.json', 'r') as file:
    labeled_data = json.load(file)

data = [entry['text'] for entry in labeled_data]
labels = [entry['label'] for entry in labeled_data]

# Split the dataset
train_texts, val_texts, train_labels, val_labels =
train_test_split(data,
                  labels,
                  test_size=0.2,
                  random_state=42)
```

Figure 3.13 : Loading and splitting the dataset

ML models require numerical input, so `TfidfVectorizer` used to convert text into a numerical matrix, rows represent documents and columns represent words in the vocabulary. The TF-IDF score represents the importance of a word to a document in the context of the entire dataset. Each cell has the TF-IDF score of a word. TF measures how often each word appears in the document and IDF reduces the importance of commonly used words across all documents (e.g. "is", "the") and emphasizes unique words. The `vectorizer.fit_transform` applied on `train_texts`, where `fit` analyzes the training data to build the vocabulary and calculate the IDF values for all words, and `transform` converts it into a numerical TF-IDF matrix. The `vectorizer.transform` uses what is learned from training data to transform the `val_texts` into a TF-IDF matrix, the validation data should be processed using the same vocabulary and IDF values as the training data to ensure consistency. Figure 3.14 shows the code for vectorizing.

```
# Vectorize the text data using TF-IDF
vectorizer = TfidfVectorizer(max_features=5000)
X_train = vectorizer.fit_transform(train_texts)
X_val = vectorizer.transform(val_texts)
```

Figure 3.14: Code for vectorizing

Preparing to train the model by creating an instance of the `LogisticRegression` model, and setting `max_iter=1000` which is 1000 maximum iterations allowed for the optimization algorithm to converge, so the process has more time to find the solution. Then `fit()` is used to train the model using the TF-IDF transformed training data `X_train` and corresponding labels `train_labels` the model learns the relationship between the input features 'text' and the output labels '0' or '1'. Logistic regression learns weights which determine how strongly each word contributes to predicting the label. The `predict()` method uses the trained model to predict labels by computing the probability of each class (0 or 1) using sigmoid function and assign the label with the highest probability as the prediction for each validation data `X_val`, then save it to the list `val_preds`. Evaluating the model by `accuracy_score` that calculate proportion of the correct predicted labels using the formula :

$$Accuracy = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

After training and predictions the evaluation metrics are printed as shown in figure 3.15

Validation Accuracy: 0.95				
Classification Report:				
	precision	recall	f1-score	support
0	0.96	0.93	0.95	29
1	0.94	0.97	0.95	32
accuracy			0.95	61
macro avg	0.95	0.95	0.95	61
weighted avg	0.95	0.95	0.95	61

Figure 3.15: The evaluation metrics of training

Saving the trained model and vectorizers to be loaded and used later without retraining it by `joblib.dump`. The trained `log_reg` model with its learned parameters (weights) is saved in the file 'logistic_regression_model.pkl'. The TF-IDF `vectorizer` that was fitted on the training is saved in the file 'tfidf_vectorizer.pkl' to ensure that any new text is vectorized in the same way. Figure 3.16 shows the code of training, evaluating, and saving the model.

```
# Train a Logistic Regression model
log_reg = LogisticRegression(max_iter=1000)
log_reg.fit(X_train, train_labels)

# Evaluate the model
val_preds = log_reg.predict(X_val)
val_accuracy = accuracy_score(val_labels, val_preds)

print(f"Validation Accuracy: {val_accuracy:.2f}")
print("Classification Report:")
print(classification_report(val_labels, val_preds))

# Save the model and vectorizer
import joblib
joblib.dump(log_reg, "logistic_regression_model.pkl")
joblib.dump(vectorizer, "tfidf_vectorizer.pkl")
```

Figure 3.16: LogisticRegression model

3.4.5 Classifying Requirement Sentences

Classifying extracted requirement sentences using the trained logistic regression model, as table 3.2 shows results of the function `classify_sentence(sentence, model, vectorizer,`

threshold=0.5) that takes the requirement sentence extracted from the script, and processed it with the model and vectorizer. The `vectorizer.transform()` ensures that the input `[sentence]` is treated as a single document and convert it into a TF-IDF. The probability is calculated for each class '0' and '1' based on the model's sigmoid function output using `model.predict_proba`. Both probabilities that the sentence belongs to class 0 and 1 are calculated. The sentence is labeled '1' if `probabilities[1]` exceed the 50% threshold, otherwise it is labeled '0'. Then convert labels to human readable, so the text "Password Requirement" is printed if `predicted_label == 1`, otherwise the text "General Policy" is printed. Figure 3.17 shows the `classify_sentence` function.

Table 3.2: Examples of `classify_sentence` function results

Website Sentence	Classified as	Probabilities	Predicted Label
<code>", "SOME_SPECIAL_CHARACTERS_NOT_ALLOWED": "You can only use letters, numbers, periods ('.'), and underscores ('_') in your username.</code>	General Policy	33.59%	0
<code>"},"password": "Passwords must have at least 8 characters and contain at least two of the following\u003a uppercase letters, lowercase letters, numbers, and symbols</code>	Password Requirement	85.94%	1
<code>lNa=function(){return "Only use letters, numbers, and common punctuation characters"</code>	Password Requirement	91.48%	1
<code>, PASSWORD_LENGTH_INVALID: Your password isn't strong enough, try making it longer.</code>	General Policy	20.78%	0
<code>, NEED_AT_LEAST_ONE_ALPHA: Please use at least one letter in your username.</code>	General Policy	34.38%	0

```
def classify_sentence(sentence, model, vectorizer, threshold=0.5):
    vectorized_text = vectorizer.transform([sentence])
    probabilities = model.predict_proba(vectorized_text)[0]
    predicted_label = 1 if probabilities[1] > threshold else 0

    # Convert the label to a human-readable format
    label_text = "Password Requirement" if predicted_label == 1
else "General Policy"
    return label_text, probabilities
```

Figure 3.17: `classify_sentence` function

3.4.6 Similar Sentence Identification

The function `find_similar_sentence` iterates over each requirement sentence in the `found_sentences` list of the website. It analyzes and classifies the list by calling the `classify_sentence` function using the pre-trained logistic regression model and the TF-IDF vectorizer (`log_reg`, `vectorizer`) to return the label and probabilities. The score of `confidence_label_1` is calculated using `probabilities[1]` where the sentence belongs to class '1' multiplied by 100 to convert it into percentage, and `round(..., 2)` which rounds the confidence percentage to two decimal places. Then append to `similar_sentences` list to add a tuple (`(website_sentence, label, confidence_label_1)`) for the current sentence. Figure 3.18, shows the code of the `find_similar_sentence` function.

```
def find_similar_sentence(found_sentences):
    similar_sentences = []

    for website_sentence in found_sentences:
        label, probabilities =
        classify_sentence(website_sentence, log_reg, vectorizer)
        confidence_label_1 = round(probabilities[1] * 100, 2)
        similar_sentences.append((website_sentence, label,
        confidence_label_1))

    return similar_sentences
```

Figure 3.18: `find_similar_sentence` function

After providing the extracted requirements to the user, creating a secure password should become easier by following the instructions. Additionally, the user will know whether the service provider accepts Arabic-letter passwords. The user is prompted to choose a base word, which will be modified to generate passwords. The password generator then produces both Arabic and Latin versions of the password. The user can select the password that best suits their preferences and meets the service provider's requirements, enhancing inclusivity by accommodating users who prefer Arabic passwords.

3.5 Arabic Password Generator

The tool generates a strong password with novel characteristics to protect the user against cracking or guessing attacks by mixing the Arabic language with Arabizi (or Franco-Arabic), numbers, and special characters. The user inputs preferred Arabic words, and the popularity of these words is checked. Based on the search results the base word is modified to generate the new Arabic password. This password contains Arabic characters, symbols, numbers, or Latin characters. Finally, two passwords are available for the user: an Arabic password and a Latin password. Figure 3.19 illustrates the steps of the password generator.

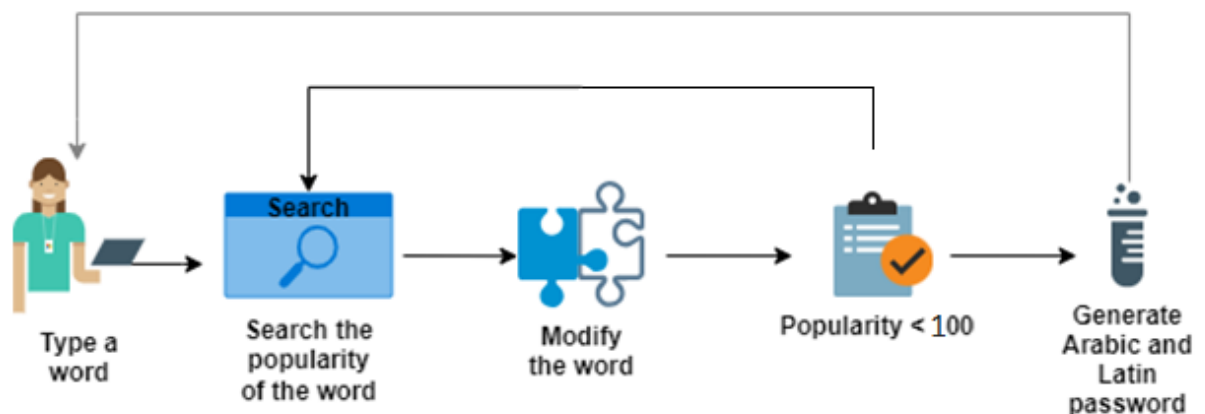


Figure 3.19: Password Generator Steps

3.5.1 Arabizi or Franco-arabic

It is an unofficial writing style that has emerged within the Arab community as a byproduct of the globalization of English-based technology. This led many people to

rely on the Latin alphabet for communication. In Arabizi, Arabic words are written phonetically using the Latin alphabet, with numbers replacing certain Arabic letters that have no direct English equivalent. These numbers resemble the shape of the corresponding Arabic letters. This writing style persists today, particularly because some Arabs find it faster to type using English keyboards.

Table 3.3 below shows the concept of Arabizi typing, though it is not standardized. Incorporating Arabizi into password creation can make the password more familiar and memorable for the user, while also enhancing its strength. The use of numbers in this context is meaningful, as opposed to using dates, which can be vulnerable to social engineering attacks. Unlike random numbers, which are easily forgotten, Arabizi-based passwords offer a balance between security and memorability.

Table 3.3: Arabizi writing style

Arabic letter	Equivalent number	Example
أ - ع - ئ	2	2na
ح	7	7md
خ	5	5alas
ع	3	3in
غ	3`	3`adab
ص	9	9br
ض	`9	`9ef
ق	8	8lb
ط	6	6abib
ذ	4	4hab

3.5.2 Search the Password

Verifying if the word is commonly used on the internet, the tool checks its popularity via Google search due to the non-existence of leaked Arabic password datasets. This function `get_search_results (user_input)` is employed to retrieve the number of search results for the user's chosen word from Google. Figure 3.20, shows the code for this function.

The function constructs a Google search query `url = f"https://www.google.com.tr/search?q={user_input}"` by inserting the user's password into the URL. To mimic a standard web browser's user-agent string and avoid being blocked as a bot or non-browser client, a header is set as `"User-`

```
Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/91.0.4472.124 Safari/537.36";
```

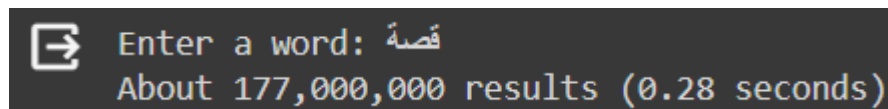
```
def get_search_results(user_input):
    url = f"https://www.google.com.tr/search?q={user_input}"
    headers = {
        "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.124 Safari/537.36"
    }
    # getting the number of search results from google
    response = requests.get(url, headers=headers)
    soup = BeautifulSoup(response.text, 'html.parser')

    result_stats = soup.find("div", {"id": "result-stats"})

    if result_stats:
        return result_stats.text
    else:
        return "No result stats found"
```

Figure 3.20: `get_search_results` function

Getting the number of results, HTTP GET `requests.get` is sent to the constructed `url`, including the `headers` to make the request appear as if it is coming from a standard web browser. The response HTML content `response.text` is parsed `'html.parser'` using the Python library `BeautifulSoup` for scraping web pages. To extract the number of search results, `soup.find` is used to search for the `<div>` element with the `id` attribute set to `"result-stats"`. If the `result_stats` element is found, the number of search results is returned by the command `return result_stats.text`, but if there is no result found it `return "No result stats found"`. Figure 3.21 shows an example of the output by running the function `get_search_results`, the number of search results is extracted and returned to measure the commonality of the password.



```
Enter a word: قصة
About 177,000,000 results (0.28 seconds)
```

Figure 3.21: Example Output of Running this Function

3.5.3 Arabic Random Letters Replacement

The user's chosen word is modified randomly to generate a strong password containing Arabic characters, numbers, special characters, lowercase characters, and uppercase characters. The function `random_replace` randomly replaces some letters from the base word that the user entered with a predefined dictionary '`replacement_dict`', where keys represent letters that may be replaced, and values are lists of replacement options for each letter. Figure 3.22 shows the code for the **`random_replace`** function.

```
def random_replace(word, replacement_dict):
    modified_word = ""
    for letter in word:
        if letter in replacement_dict:
            replacement_options = replacement_dict[letter]
            random_value =
random.choice(replacement_options)
            modified_word += random_value
        else:
            modified_word += letter
    return modified_word
# choose random letters and replace it
```

Figure 3.22: `random_replace` Function

The password is the modified version of the original word the user typed, stored in `modified_word = ""` where certain letters have been replaced. The function iterates over each letter `in` word, and `if` letter exists `in` `replacement_dict`, the function retrieves the list of `replacement_options` associated with that letter `replacement_dict[letter]`. Then, a random replacement option is selected `random_value = random.choice(replacement_options)`. Each chosen replacement `random_value` is concatenated to the `modified_word`. If the letter has no replacement options, the original letter retains `modified_word += letter`. After processing all the letters in the original word, the modified word is returned `return modified_word`. Figure 3.23, shows the output of a word that was searched and modified by running the **`random_replace`** function. The new word is stronger, contains more character sets, and looks random, though it is memorable and easy to type. It is more familiar to the user, achieving two goals: memorability and strength.

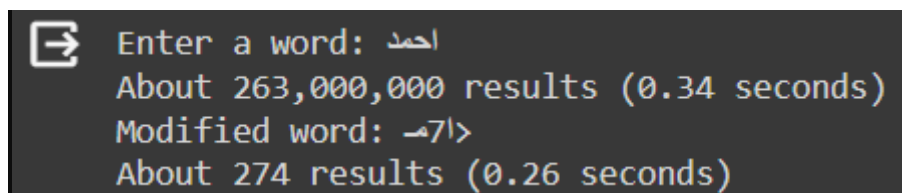


Figure 3.23: Example of a Modified Word

The replacement dictionary used here is a mix of Arabizi and additional symbols or Latin characters that represent the approximate shape of Arabic characters, which is not a known replacement or writing style. The approximate similarity between both is chosen and decided as shown in Table 3.4.

Table 3.4: replacement_dict

Keys	Values	Keys	Values
ا	'1' '1'	ط	'6'
أ	'2' 'i'	ظ	'`6'
إ	'!' '!''	ع	'3'
آ	'1`' '1`'	غ	'`3'
ب	'].'' ').''	ف	';' 'b.'
ت	'[:]'' '[:)''	ق	'8'
ث	'[:.]'' '[:.)'	ك	'L^'
ج	'>.'	ل	'L'
ح	'7'	م	'o'
خ	'`7' '5'	ن	'.]'' '.)'
د	'>'	ه	'@'
ذ	'4' '``>'	و	'&' 'g'
ر	')''	ؤ	'^g' '^,' '^و'
ز	'`)''	ي	'[:]' '):'
س	'?'	ى	']' ')''
ش	'\$'	ئ	'2'
ص	'9'	ء	'^'
ض	'`9'	ة	'[:@'

3.5.4 Latin Random Letters Replacement

This function is useful when the service provider does not accept the Arabic characters in the password. It generates a strong, unique password that looks random but memorable to the Arabic users with limited English vocabulary. The user presses the Arabic letters in the same order as the Arabic password, but with the Latin-character input mode enabled. The user's chosen word is modified randomly to return a strong password containing numbers, special characters, lowercase characters, and uppercase characters. The function `random_engreplace` randomly replaces some letters from the base word that the user input with a predefined dictionary `'replace_eng_dict'`, where keys represent letters that may be replaced, and values are lists of replacement options for each letter. Figure 3.24 shows the code of the `random_engreplace` function.

```
def random_engreplace(word, replace_eng_dict):
    modified_word = ""
    for letter in word:
        if letter in replace_eng_dict:
            replacement_options = replace_eng_dict[letter]
            random_value = random.choice(replacement_options)
            modified_word += random_value
        else:
            modified_word += letter
    return modified_word
```

Figure 3.24: `random_engreplace` function

The password is the modified version of the original word the user typed, stored in `modified_word = ""`, where certain letters have been replaced. The function iterates over each `for letter in word`, and `if letter exists in replacement_dict` the function retrieves the list of `replacement_options` associated with that letter `replace_eng_dict[letter]`, then a random replacement option is selected `random_value = random.choice(replacement_options)`. Each chosen replacement `'random_value'` is concatenated to the `modified_word`. Otherwise, means the letter has no replacement options, the original letter is retained `modified_word += letter`. After processing all the letters in the original word, the modified word is returned `return modified_word`. Figure 3.25, shows the Latin password output by running the `random_engreplace` function.

```
word = "رؤباعدل"
modified_word = random_engreplace(word, replace_eng_dict)
print(modified_word)
```

v2n3h]g

Figure 3.25: Example of a Latin Modified Word

The replacement dictionary used here is a mix of Arabizi and the Latin characters that are placed on the keyboard at the same key as the corresponding Arabic character. The user needs to know the position of each Arabic character on the keyboard to find the appropriate replacement, as shown in Table 3.5.

Table 3.5: replace_eng_dict

Keys	Values	Keys	Values
ا	'h'	ط	'6' ' '
أ	'H'	ظ	'`6' ' / '
إ	'Y'	ع	'3' 'u'
آ	'N'	غ	'`3' 'y'
ب	'f'	ف	't'
ت	'j'	ق	'8' 'r'
ث	'e'	ك	';'
ج	'['	ل	'g'
ح	'7' 'p'	م	'l'
خ	'`7' '5' 'o'	ن	'k'
د	']'	ه	'i'
ذ	'4' '``'	و	' ,'
ر	'v'	ؤ	'2' 'c'
ز	'.'	ي	'd'
س	's'	ى	'n'
ش	'a'	ئ	'z'
ص	'9' 'w'	ء	'x'
ض	'`9' 'q'	ة	'm'

3.5.5 Iterative Modification Process

This function iteratively modifies the base word until a certain condition is met. It works with the previous functions that continuously retrieve search results from

Google of the modified word and check its popularity. Figure 3.20, shows the function `modify_word` that takes the parameter `user_input`, which is the user's base word that will be modified. The function `get_search_results` is called to get the number of search results for `('"' + user_input + '"')`. The double quotes are used to search for the exact phrase without Google modification, and then the results of the search are returned. The function compares the number of search results `'num_results'` from Google with the predefined threshold of 100; `int(results.split()[1].replace('.', ''))` extracts the number of the search results from the string and converts it to an integer. If `num_results < 100`, which indicates a less common word – as was found that the top 200 used passwords were used by 4,000,000 to 16,000 users – therefore, a threshold of 5,000,000 to 10,000 search results is not ideal.

Shin & Woo (2022) categorized password strength groups: weak ($\text{guesses} < 10^6$), medium ($10^6 \leq \text{guesses} < 10^{14}$), and strong ($\text{guesses} \geq 10^{14}$) with the goal being to achieve strong passwords. To determine the optimal threshold, a number of generated passwords are tested based on strength, search results, and the cracking time, as demonstrated by the code in figure 3.26. By providing a password list with the frequency of each password, the program calculate metrics such as character set size, length, search space (number of guesses), entropy, worst time (in seconds and years), and the average time (in seconds and years). The results are saved as `password_metrics.csv` file.

```

def calculate_metrics(passwords, frequencies, guesses_per_second):
    """
    Calculate search space, entropy, cracking time, and number of guesses for a list of passwords,
    along with their frequencies.
    Args:
    passwords (list): List of passwords to evaluate.
    frequencies (list): Frequency of each password's usage.
    guesses_per_second (int): Number of guesses the attacker can make per second.
    Returns:
    pd.DataFrame: A DataFrame containing password metrics and frequencies.
    """
    charset_sizes = {          # Character set sizes
        'lowercase': 26,
        'uppercase': 26,
        'digits': 10,
        'special': 32,
        'arabic': 35 }
    data = []
    for password, frequency in zip(passwords, frequencies):
        # Determine character set size based on password composition
        char_set_size = 0
        if any(c.islower() for c in password):
            char_set_size += charset_sizes['lowercase']
        if any(c.isupper() for c in password):
            char_set_size += charset_sizes['uppercase']
        if any(c.isdigit() for c in password):
            char_set_size += charset_sizes['digits']
        if any(not c.isalnum() for c in password):
            char_set_size += charset_sizes['special']
        if any('\u0600' <= c <= '\u06FF' for c in password): # Check for Arabic characters
            char_set_size += charset_sizes['arabic']
        search_space = char_set_size ** len(password) # Calculate search space
        number_of_guesses = search_space # Number of guesses required to crack the password (equals the
        search space)
        entropy = math.log2(search_space) # Calculate entropy
        # Calculate cracking time (worst case and average case)
        worst_case_time = search_space / guesses_per_second
        average_time = worst_case_time / 2
        # Convert times to human-readable format (seconds to years)
        worst_case_years = worst_case_time / (60 * 60 * 24 * 365)
        average_years = average_time / (60 * 60 * 24 * 365)
        # Store results for the password
        data.append({
            'Password': password,
            'Frequency': frequency,
            'Character Set Size': char_set_size,
            'Password Length': len(password),
            'Search Space': search_space,
            'Number of Guesses': number_of_guesses,
            'Entropy (bits)': entropy,
            'Worst-Case Time (seconds)': worst_case_time,
            'Average Time (seconds)': average_time,
            'Worst-Case Time (years)': worst_case_years,
            'Average Time (years)': average_years })
        # Create a DataFrame and save it as a CSV
    df = pd.DataFrame(data)
    df.to_csv('password_metrics.csv', index=False)
    return df
# Example usage
passwords = ["^g", "١٤L", "L^L:م:O", "L^Lo: o&)", "(م)Loك", "1:[قصه ح]:"] # Arabic password
frequencies = [0, 11, 16, 12, 25] # Frequency of each password usage
guesses_per_second = 15_843_900 # 15,843,900 guesses per second
metrics_df = calculate_metrics(passwords, frequencies, guesses_per_second)
print(metrics_df) # Display the DataFrame

```

Figure 3.26 : Calculating password metrics function

By testing 50 passwords, the results shown in figure 3.27 demonstrate that nearly all of the generated passwords achieve between 0 and 100 results in a Google search, indicating that they are secure and uncommon. Upon closer examination of the graph in figure 3.28, it is obvious that a longer cracking time is required. Stronger passwords are achieved when the threshold is set to less than 100 search results, meaning the password is ready to use when Number of results is less than 100.

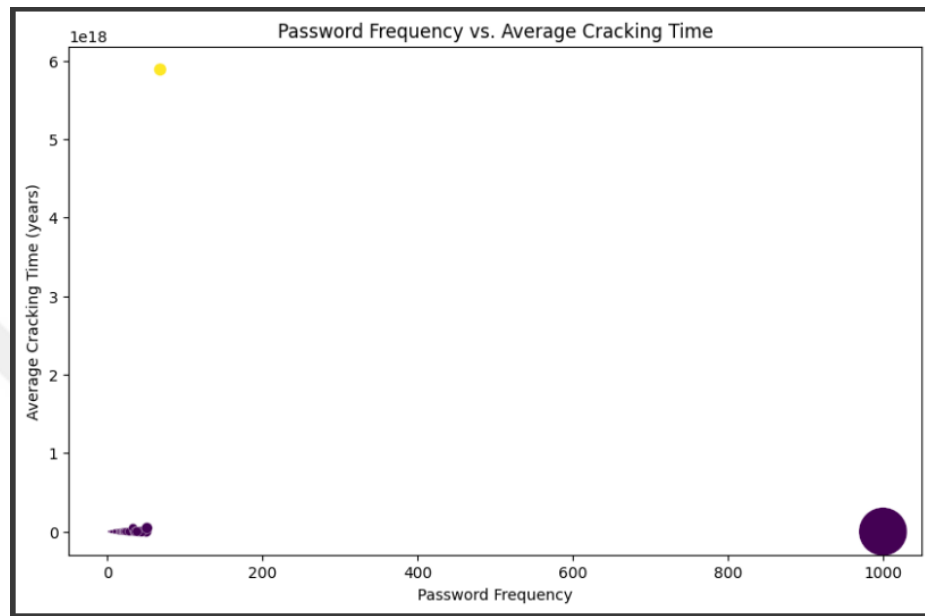


Figure 3.27: Results of testing 50 passwords

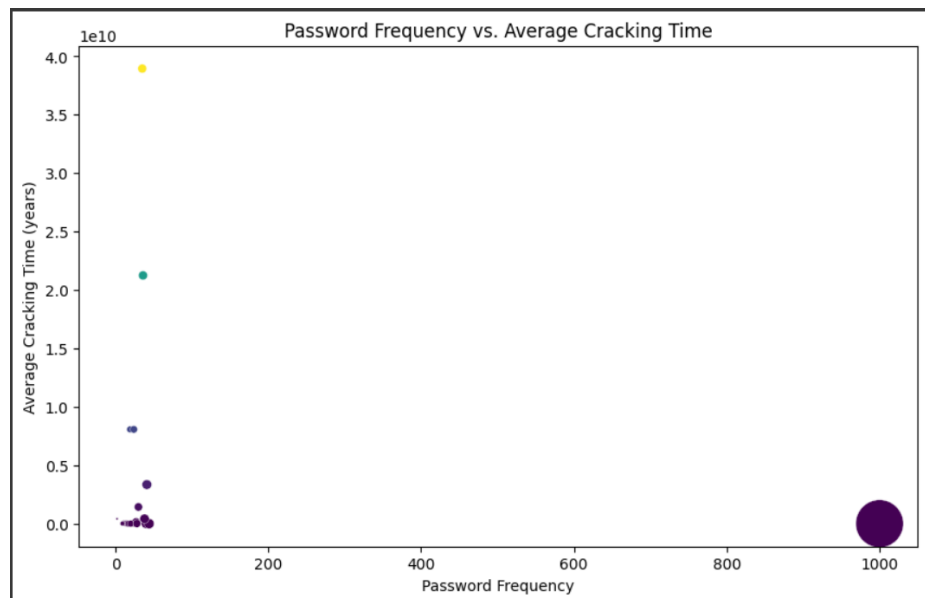


Figure 3.28 : Closer examination of the results

Otherwise, when it is 100 or more, the word will be modified by the `random_replace` function, which generates an Arabic password, and

`print(f"Modified word (Arabic): {modified_word_ar}", as`
shown in figure 3.29.

For the Latin password, the function `modify_word_with_eng` is used to replace the `user_input` with the English `replace_eng_dict` dictionary, and `f"Modified word (English): {modified_word_en}"` is print as shown in figure 3.30. When the `modified_word_ar` is the same as the base `user_input` word, it `print("No further modifications possible. Exiting.")`.

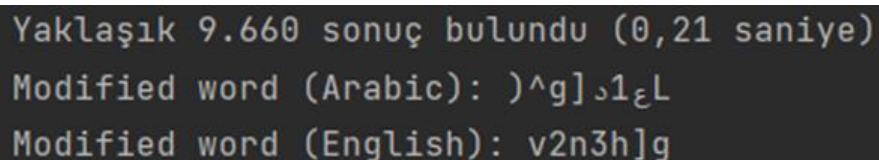
```
def modify_word(user_input):
    while True:
        results = get_search_results('' + user_input + '')
        print(results)

        num_results = int(results.split()[1].replace('.', ''))

        if num_results < 100:
            print(f"Use. Number of results is less than 100.")
            return user_input
        else:
            modified_word_ar = random_replace(user_input, replacement_dict)
            #modified_word
            modified_word_en = modify_word_with_eng(user_input,
            replace_eng_dict) #new
            print(f"Modified word (Arabic): {modified_word_ar}")
            print(f"Modified word (English): {modified_word_en}")
            # modified_word
            if modified_word_ar == user_input:
                print("No further modifications possible. Exiting.")
                return user_input

            user_input = modified_word_ar # modified_word
```

Figure 3.29: `modify_word` Function



```
Yaklaşık 9.660 sonuç bulundu (0,21 saniye)
Modified word (Arabic): )^g]ۛ1εL
Modified word (English): v2n3h]g
```

Figure 3.30: Output of `modify_word` Function

3.6 Web Application Integration

Appendix A.2 shows the functionality for generating passwords, it is integrated into a web application using the Flask framework. Figure 3.31 shows the interface of the tool where the user can submit and use both the extractor and generator as shown in

appendix A.3. The Flask web application defines a route Python function `/submit` an endpoint to handle incoming `['POST']` requests to receive JSON data containing either a URL in the extractor or a word input in the generator, `request.get_json()` is used to extract JSON data from POST request and store it in the `data` variable. There are two cases for which data will be checked:

Case 1, if `'url'` in `data` the URL provided in the JSON data, the requirements will be extracted using the function `extract_js_from_url` to retrieve the JavaScript code, then processed to find sentences containing keywords related to password requirements using the function `search_specific_sentences(js_code)`, when requirements are found they are analyzed and classified using the pre-trained logistic regression model and the TF-IDF vectorizer. By employing the function `find_similar_sentence(found_sentences)`, depending on the outcome it return `jsonify({'result': similar_sentences})` if `similar_sentences` were found, `print(f"\nFound the following sentences in the JavaScript code:")` with the message indicating "Not similar to the dataset." When there are no similar sentences. The `'message': "No specified words found in the JavaScript code."` is printed when there are no password requirements found in the JS code, and the `'message': "No JavaScript code found in the provided URL."` is printed when the JS code was not found in the provided URL.

Case 2, elif `'input'` in `data` contains a word submitted by the user indicating a word modification request to generate passwords, the `user_input` is modified using the function `modify_word` to generate a unique Arabic password which is `modified_word_ar` a modified version, and the function `modify_word_with_eng` to generate an English letters password which is `modified_word_en` a modified version. Both passwords are returned as a JSON response, and a `'message': "Invalid request."` is returned when the JSON data does not contain either a `'url'` or `'input'`. Figure 3. 32, shows the full code for both cases; if a URL is submitted or an input word is submitted by the user. Appendix A.4 shows `'Results.html'` that display the results.

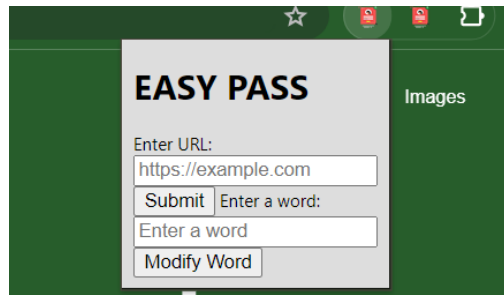


Figure 3.31: The Chrome Extension

```
@app.route('/submit', methods=['POST'])
def submit():
    data = request.get_json()
    if 'url' in data:
        # Case 1: Processing URL
        user_url = data.get('url')
        js_code = extract_js_from_url(user_url)
        found_sentences = []
        if js_code:
            found_sentences = search_specific_sentences(js_code)
            if found_sentences:
                similar_sentences = find_similar_sentence(found_sentences)
                if similar_sentences:
                    return jsonify({'result': similar_sentences})
                else:
                    print(f"\nFound the following sentences in the JavaScript code:")
                    for sentence in found_sentences:
                        print(sentence)
                    return jsonify({'found_sentences': found_sentences, 'message': "Not
similar to the dataset."})
            else:
                return jsonify({'message': "No specified words found in the JavaScript
code."})
        else:
            return jsonify({'message': "No JavaScript code found in the provided URL."})
    elif 'input' in data:
        # Case 2: Modifying Word
        user_input = data.get('input')
        modified_word_ar = modify_word(user_input) # modified_word
        modified_word_en = modify_word_with_eng(user_input, replace_eng_dict) #new
        return jsonify({'result_arabic': modified_word_ar, 'result_english':
modified_word_en})
    else:
        return jsonify({'message': "Invalid request."})
```

Figure 3.32: Route submit()

3.7 HTML Structure and Client-Side JavaScript Functionality

This section provides an overview of how the HTML file, JavaScript functionality, and Flask server interact to deliver the functionality of the Chrome extension. The HTML file `popup.html` as shown in figure 3.33, serves as the front-end interface for the Chrome extension called "ARAB PASSWORD" includes two input fields and buttons, "urlInput" to Enter URL to extract its JS code with

placeholder="https://example.com" to show the user an example of the URL, followed by a `urlSubmitButton` that the user can click to submit the URL, and "userInput" to Enter a word which is the base word of the password before modification with `wordSubmitButton` that the user can click to submit the base word. Both `resultsContainer` which displays the `resultsList` of the extracted JS codes, and `resultDisplay` which displays the generated passwords, are initially hidden until the output is displayed. `<script src="popup.js"></script>` includes an external JavaScript file named "popup.js" to provide functionality to the HTML elements that is shown in appendix A.5.

```
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Popup</title>
  <style>
    body { background-color: #dddddd; /* Set the background color to gray */}
  </style>
</head>
<body> <h1>EASY PASS</h1> <!-- Input field and button for URL -->
  <label for="urlInput">Enter URL:</label>
  <input type="text" id="urlInput" placeholder="https://example.com">
  <button id="urlSubmitButton">Submit</button>
  <!-- Display area for results -->
  <div id="resultsContainer" style="margin-top: 20px; display: none;">
    <h2>Results:</h2>
    <ul id="resultsList"></ul>
  </div> <!-- Input field and button for modifying word -->
  <label for="userInput">Enter a word:</label>
  <input type="text" id="userInput" placeholder="Enter a word">
  <button id="wordSubmitButton">Modify Word</button>
  <!-- Display area for modified word -->
  <div id="resultDisplay" style="margin-top: 20px; display: none;"></div>
  <!-- Message container for displaying errors or messages -->
  <div id="messageContainer" style="color: red; margin-top: 20px; display: none;"></div>
  <script src="popup.js"></script>
```

Figure 3.33: `popup.html`

Appendix A.6 shows the JavaScript file `popup.js` that is responsible for adding functionality to the HTML elements by handling the user interactions to make requests to the Flask server. `DOMContentLoaded` event listener ensures that this JS code executes only when the HTML document has been completely loaded. Figure 3.34 shows the declared variable that stores references to HTML elements as buttons, input fields, and result containers. The `messageContainer` and `resultsList` are

checked to ensure they exist if they are not, so they are dynamically created and `appendChild` appended.

```
document.addEventListener('DOMContentLoaded', function () {
    var urlSubmitButton = document.getElementById('urlSubmitButton');
    var wordSubmitButton = document.getElementById('wordSubmitButton');
    var urlInput = document.getElementById('urlInput');
    var userInput = document.getElementById('userInput');
    var resultsContainer = document.getElementById('resultsContainer');
    var resultDisplay = document.getElementById('resultDisplay');
    // Ensure messageContainer is defined before using it
    var messageContainer = document.getElementById('messageContainer');
    if (!messageContainer) {
        // If messageContainer is not found, create it dynamically
        messageContainer = document.createElement('div');
        messageContainer.id = 'messageContainer';
        document.body.appendChild(messageContainer);
    }
    // Ensure resultsList is defined before using it
    var resultsList = document.getElementById('resultsList');
    if (!resultsList) {
        // If resultsList is not found, create it dynamically
        resultsList = document.createElement('ul');
        resultsList.id = 'resultsList';
        document.body.appendChild(resultsList);
    }
});
```

Figure 3.34: Snippet Code From `popup.js`

Figure 3.35 shows the function `displayFoundSentences` that takes an array of found sentences as input and displays them as an `alertMessage`. The attached `addEventListener` handles user interactions with the submit buttons (`urlSubmitButton` and `wordSubmitButton`), by clicking buttons these functions are executed.

```
// Function to display found sentences as an alert
function displayFoundSentences(foundSentences) {
    // Create a string containing all the found sentences
    var alertMessage = "Found Sentences in JS code:\n";
    foundSentences.forEach(function(sentence) {
        alertMessage += "- " + sentence + "\n";
    });
    // Display the alert message
    alert(alertMessage);
}
```

Figure 3.35: `displayFoundSentences` Function

When the URL is submitted it triggers a fetch request to the Flask server with the URL entered and upon receiving a response from the server it processes the data and dynamically updates the user interface, the function `urlSubmitButton` defined inside the event listener to be executed by clicking the button, the entered value `urlInput.value` retrieved and trimmed to remove any whitespaces, then the request to the Flask server (`http://localhost:5000/submit`). Depending on the structure of the response data `data.result` exists means similar sentences were found and the function iterates `data.result.forEach(result =>` over each `result` that represents a pair of sentences with their similarity score which is converted to `percentage = result[2]` formatting it to two decimal places `percentage.toFixed(2) + "%"`, the line `percentageStr + " you cannot use Arabic letters"` displayed to inform the user that the percentage is high and similar to the dataset so, the service providers don't accept Arabic letters. The list item `resultsList` is then displayed, but if there is an error or no data, appropriate messages are displayed. Full code is shown in Figure 3.36.

```

urlSubmitButton.addEventListener('click', function () {
    console.log('Button clicked');
    var url = urlInput.value.trim();
    if (url) {
        console.log('Sending request to Flask server...');
        // Triggering a fetch request to the Flask server for URL
        fetch('http://localhost:5000/submit', {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
            },
            body: JSON.stringify({ url: url })
        })
        .then(response => response.json())
        .then(data => {
            console.log('Data received from Flask server:', data); //
            // Handle the response data here for URL
            // Example: Displaying results in the popup UI
            if (data.result) {
                console.log('Similar Sentences:');
                data.result.forEach(result => {
                    console.log('Website Sentence:', result[0]);
                    console.log('Classified as:', result[1]);
                    console.log('Prediction:', result[2]);
                    console.log('---');
                    // Display results logic here down add by me till elseif
                    // Calculate and print percentage
                    var percentage = result[2];
                    var percentageStr = percentage.toFixed(2) + "%"; // Format to two decimal places
                    console.log(percentageStr + " you cannot use Arabic
letters");

                    // Display results in the popup UI
                    var listItem = document.createElement('li');
                    listItem.innerHTML = `<strong>Website Sentence:</strong>
${result[0]}<br> <strong>Classified as:</strong> ${result[1]}<br>
<strong>Prediction:</strong> ${result[2]}<br> (${percentageStr} you cannot use Arabic
letters)`;

                    resultsList.appendChild(listItem);});
                // Show the results container and hide the message container
                resultsContainer.style.display = 'block';
                messageContainer.style.display = 'none';
            } else if (data.message) {
                console.log('Error:', data.message); //added by me
                // Display error message in the message container
                messageContainer.textContent = data.message;
                messageContainer.style.display = 'block';
                resultsContainer.style.display = 'none';
            }
            // Display found sentences if available
            if (data.found_sentences) {
                console.log('Found Sentences:', data.found_sentences);
                displayFoundSentences(data.found_sentences);
            }
        })
        .catch(error => {
            console.error('Error:', error);
        });
    } else {
        console.log('Please enter a URL.');
```

Figure 3.36: urlSubmitButton.addEventListener

When the word is submitted it triggers a fetch request to the Flask server with the word entered and upon receiving a response from the server it processes the data and dynamically updates the user interface, the function `wordSubmitButton` defined inside the event listener to be executed by clicking the button, the entered value `userInput.value.trim()` retrieved and trimmed to remove any whitespaces, if the two passwords `data.result_arabic` and `data.result_english` is returned means the input word was modified, the passwords displayed in the `resultDisplay` area. Code is shown in Figure 3.37.

```
wordSubmitButton.addEventListener('click', function () {
    var input = userInput.value.trim();

    if (input) {
        // Triggering a fetch request to the Flask server for word
        fetch('http://localhost:5000/submit', {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
            },
            body: JSON.stringify({ input: input }),
        })
        .then(response => response.json())
        .then(data => {
            // Handle the response data here for word
            // ...

            // Example: Displaying modified word in the popup UI
            if (data.result_arabic && data.result_english) {
                // Display modified word logic here
                resultDisplay.innerHTML = `<strong>Modified Word
(Arabic):</strong> ${data.result_arabic}<br> <strong>Modified Word
(English):</strong> ${data.result_english}`;
                resultDisplay.style.display = 'block';
                resultsContainer.style.display = 'none';
                messageContainer.style.display = 'none';
            } else if (data.message) {
                // Display error message in the message container
                messageContainer.textContent = data.message;
                messageContainer.style.display = 'block';
            }
        })
        .catch(error => {
            console.error('Error:', error);
        });
    } else {
        console.log('Please enter a word.');
        // Display a message in the message container
        messageContainer.textContent = 'Please enter a word.';
        messageContainer.style.display = 'block';
    }
});
```

Figure 3.37: `wordSubmitButton.addEventListener`

3.8 Summary

A novel tool integrating password requirement extraction and password generation to enhance the security and usability of multilingual passwords particularly for Arabic-speaking users. The tool addresses two key challenges in password management: meeting varied password policies across web services and generating strong, memorable passwords. The Password Requirements Extractor component leverages Artificial Intelligence (AI) and Natural Language Processing (NLP) to analyze HTML or JavaScript code from web pages. Extracted content is parsed into a tree structure, allowing the tool to identify `<script>` tags and search for specific keywords related to password policies. The extracted requirements are classified using the trained logistic regression model which help identify restrictions on non-English characters, such as Arabic letters, ensuring that users are aware of constraints before attempting password creation.

The Password Generator component allows users to input a base word, which is then modified to incorporate Arabic and Latin characters, Arabizi (Franco-Arabic) substitutions, numbers, and symbols, resulting in strong, unique passwords. The generator checks the popularity of each modification, ensuring that commonly used words (appearing more than 100 times in Google search) are avoided to strengthen security. The tool produces two versions of each password—a password with Arabic characters and one with Latin characters—for use with service providers that do not support Arabic input. This tool, implemented as a browser extension with an intuitive interface, addresses the gap in multilingual password security by supporting Arabic-speaking users and expanding the character space to deter attackers.

CHAPTER 4

FINDINGS AND DISCUSSION

4.1 Introduction

Passwords are the dominant line of defense to secure data, and because of their importance, most service providers recommend policies such as the length of the password. Some service providers do not mention the password requirements. The password requirements extractor saves time and helps the user understand how to create a strong password by extracting the requirements from the JavaScript code of the service provider. This study provides a new approach to password creation, enhancing both strength and memorability. The user chooses the base word of their password, whether a phrase or a pattern, to be used in generating the Arabic password. Allowing the user to use their native language, with its vast vocabulary and dialects, helps the user become more familiar with and easily memorize the password. The shape similarity between the Arabic characters and the chosen replacement Latin characters, numbers, or symbols enhances memorability. Arabic writing is right-to-left which helps in character shuffling, and it adds a new character set to be used in password creation, namely Arabic characters. Since there are no leaked Arabic password datasets, dictionary attacks for guessing these passwords is not easy. The only barrier to using Arabic characters in passwords is whether service providers accept Arabic passwords, so in such cases, another password without Arabic letters is generated for the user.

4.2 Password Requirements Extractor Examination

The password Requirements Extractor is a useful tool for guiding the user through the password creation process, avoiding the repetitive cycle of password rejection and retry, whether creating passwords with Arabic or Latin characters. The output of the extractor is based on how the requirements are defined in the service provider's JavaScript code.

When extracting the password requirements from Microsoft's Windows Live Email service, these were classified as 'password requirements' with prediction of 93.81% that non-English characters are not permitted, and both lowercase and uppercase letters are mandatory (Figure 4.1).

The signup link of Microsoft Windows live email	Enter URL: https://signup.live.com/signu Submit Results: Website Sentence: ";,"password": "Passwords must have at least 8 characters and contain at least two of the following\u003a uppercase letters, lowercase letters, numbers, and symbols. Classified as: Password Requirement Prediction: 93.81 (93.81% you cannot use Arabic letters) Enter a word: Enter a word Modify Word
https://signup.live.com/signup?lcid=1033&wa=wsignin1.0&rpsnv=16&ct=1696846431&rver=7.0.6738.0&wp=MBI_SSL&wreply=htps%3a%2f%2foutlook.live.com%2fowa%2f%3fnp%3d1%26signup%3d1%26cobrandid%3dab0455a0-8d03-46b9-b18b-df2f57b9e44c%26RpsCsrfState%3d67956a78-8d1f-5023-5337-7738710d1b68&id=292841&CBCXT=out&lw=1&fl=dob%2cflname%2cwld&cobrandid=ab0455a0-8d03-46b9-b18b-df2f57b9e44c&lic=1&uaid=29bb868549de475897eeaffaf589abde	
The extracted JS password requirement	
Website Sentence: ";,"password": "Passwords must have at least 8 characters and contain at least two of the following\u003a uppercase letters, lowercase letters, numbers, and symbols.	
LR pre-trained model classification	
Classified as: Password Requirement	
LR pre-trained model prediction	
Prediction: 93.81 (93.81% you cannot use Arabic letters)	

Figure 4.1: Windows live password requirements

The result was confirmed in Figure 4.2, by rejection of the password "دلاؤى3" which contains {4: ى ,ؤ ,ل ,د} Arabic characters, {1:) } symbol, {1: 3} number, and {1: l} lowercase letter.

Microsoft

← cysiipassii@hotmail.com

Create a password

Enter the password you would like to use with your account.

Passwords must have at least 8 characters and contain at least two of the following: uppercase letters, lowercase letters, numbers, and symbols.

دلاؤى3|

☒ Show password

Figure 4.2: Windows live Arabic password rejection

Figure 4.3 shows Yahoo Email service extracted requirements. Yahoo does not specify anything regarding the use of English characters, but does mention the allowed characters and predict the rejection of Arabic letters as 51.35% , so if the Arabic

characters treated by yahoo as symbols, Arabic password will be rejected. Other extracted requirements show the General Policies of the service provider. However, Figure 4.4 shows that password ")3للاؤى3 " was rejected because the Arabic letters were treated as symbols.

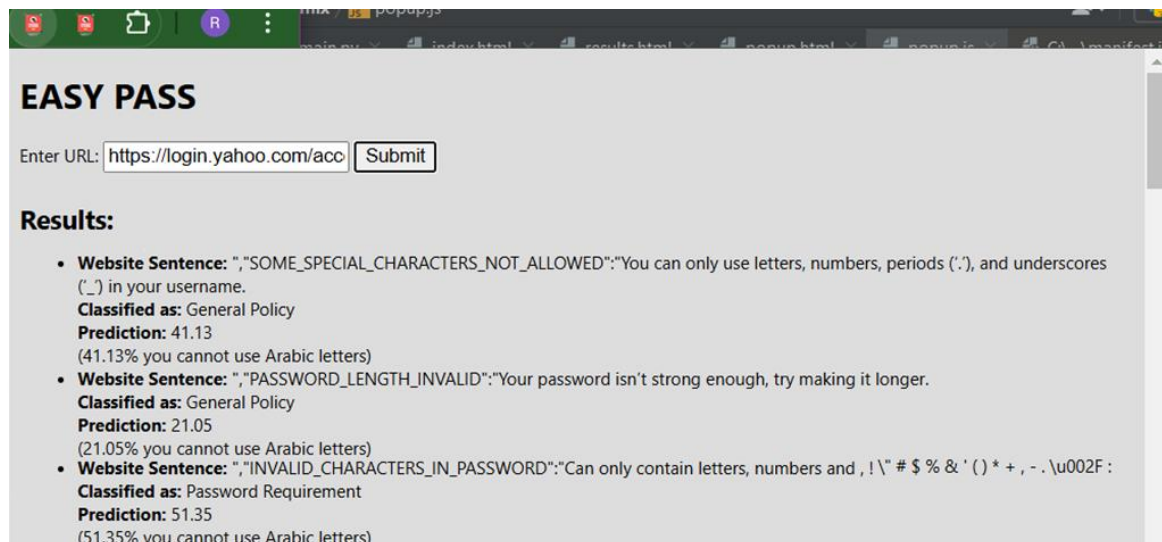


Figure 4.3: Yahoo extracted requirements

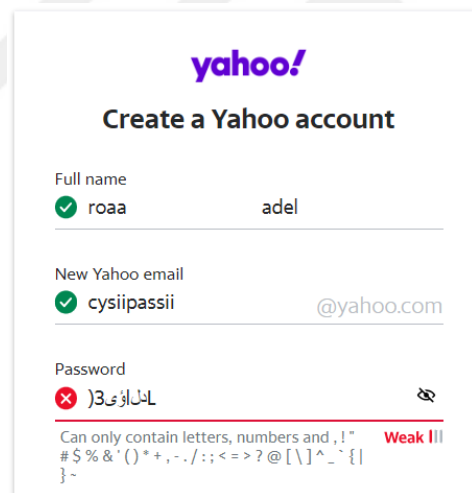


Figure 4.4: Yahoo Arabic password rejection

GMX mailbox service extracted requirements are shown in figure 4.5. GMX does not specify anything regarding the use of English characters, but does mention that the password must consist of upper and lower case letters, numbers, length, and special characters. The prediction that Arabic letters will be rejected is 59.12%. Other extracted requirements show the General Policies of the service provider. Figure 4.6 shows that password ")3للاؤى3 " was rejected.

- **Website Sentence:** It is safer to use an unique password that nobody else knows.
Classified as: General Policy
Prediction: 22.87
(22.87% you cannot use Arabic letters)
- **Website Sentence:** ", "SENDING_REGISTRATION_DATA": "Registration data are being transferred...", "INSECURE_PASSWORD": "Password strength: Low - Please choose another password."
Classified as: General Policy
Prediction: 34.54
(34.54% you cannot use Arabic letters)
- **Website Sentence:** ", "SECURE_PASSWORD_HINTS": "Tips for a secure password", "SECURE_PASSWORD_DESCRIPTION": "Use a unique password for each and every service.
A secure password consists of
 - At least eight characters
 - Numbers and letters
 - Upper and lower case letters
 - Special characters
 ", "CHECKING_DESIRED_EMAIL": "The selected email address is being checked", "DESIRED_EMAIL_AVAILABLE": "Awesome!
Classified as: Password Requirement
Prediction: 59.12
(59.12% you cannot use Arabic letters)
- **Website Sentence:** ", "INFOBOX_RECOVERY_PANEL_GUARANTEE": "We will only use your contact information for password recovery or for service messages related to your account.
Classified as: General Policy
Prediction: 26.05
(26.05% you cannot use Arabic letters)
- **Website Sentence:** ", "M_TAN_VERIFICATION_BUTTON": "Agree and continue", "SHOW_PASSWORD": "Show password", "HIDE_PASSWORD": "Hide password", "PLACEHOLDER_DD": "DD", "PLACEHOLDER_MM": "MM", "PLACEHOLDER_YYYY": "YYYY", "MAX_LENGTH_ALLOWED": "Use maximum {{count}} characters", "MIN_LENGTH_REQUIRED": "Use at least {{count}} characters", "PASSWORD_RECOVERY_SPACE_USED": "Blank spaces are not permitted: please correct your submission", "STATE_REQUIRED": "Select a state", "EXAMPLE_DMY": "e.g.
Classified as: Password Requirement
Prediction: 50.19

Figure 4.5: GMX extracted requirements

GMX Registration

3 of 5

Password ⓘ

Use a unique password for each and every service.
A secure password consists of

- At least eight characters
- Numbers and letters
- Upper and lower case letters
- Special characters

Choose a password

لاأؤى3

⚠ Please choose another password

Figure 4.6: GMX rejection

As shown in the examples above, service providers rejected passwords containing Arabic characters, regardless of whether this restriction was explicitly mentioned in their requirements. The Password Requirements Extractor retrieves the exact requirements statements from the JS code, providing clarity as shown in Figure 4.7. While using non-English characters in passwords is not common, several service providers accept Arabic letters as shown in Table 4.1.

Table 4.1: Arabic password acceptance

Service Providers	Passwords
Facebook	ر2عل>ل
Instagram	ر2عل>ل
Amazon	ر2عل>ل
Twitter	ر2عل>لr
Github	ر2عل>لr

Enter URL:

 Enter a word:

Not similar to the dataset.

Enter URL:

 Enter a word:

No specified words found in the JavaScript code.

Enter URL:

 Enter a word:

Not similar to the dataset.

Figure 4.7: Requirements Extractor of Arabic password acceptance

4.3 Arab Password Generator Examination

The Password Generator generates two passwords by modifying the base word chosen by the user. The modification process is random, ensuring that the user receives different passwords with each click on the "Modify" button. Table 4.2 provides a sample of generated passwords, which may include Arabic characters, Arabizi, numbers, symbols, and Latin characters.

Table 4.2: Arab password samples

Base word	Arabic passwords	Latin passwords	Memorization	
كلمة مرور	لم: @ م) & ر^ ل ر& (و كلمة ر& (و مة ل^ ل ر(و مرهل ل^ ل ر(و لمة مر ل^ ل ر(و ة م ل^ ل	;glm lv,v ;glm o)&v	ك	L^
			ل	L
			م	o
			ة	@:
			ر	)
			و	& g
رؤيعادل	ر>ل>ع(ر^) ر>ل>ع1(ر^)	v2n3h]g vcn3h]g v2nuh]g	ؤ	^g ^و
			ى	[)
			ع	3

	31>L^و ر		ا	1 1
	3اد^و		د	>
قصة حياتي	89:]]:(ة: حيا	r9m 7dhjd	ق	8
	ج:))[:ل:]]:ق9@:	rwm 7dhjd	ص	9
	1:]]:ق9@:ج:	rwm pdhjd	ح	7
	89:ة:ج		ي	] :) :
			ت	:] :) :
قائمة سوداء	ءاقا2م@:ء&د	rhzlms,]hx	ئ	2
	8^ائم@:سودا		س	?
	812^مة?و<ا		ء	^

4.3.1 Uniqueness

In the absence of leaked password datasets containing Arabic-characters, the popularity of the base word is gauged using Google search engine. The number of search results is checked after each modification on the base word to ensure that random alterations do not compromise the uniqueness of the password (Figure 4.8). Additionally, regardless of whether the user chooses a meaningful or meaningless word as the base, the process takes advantage of the auto-reordering of characters by mixing right-to-left Arabic letters with numbers, Latin letters, and symbols. This increases the time required for dictionary or guessing attack. The input characters remain consistent with the base word while entering the password into service providers. Figure 4.9 illustrates the steps for typing the password by pressing keyboard buttons.

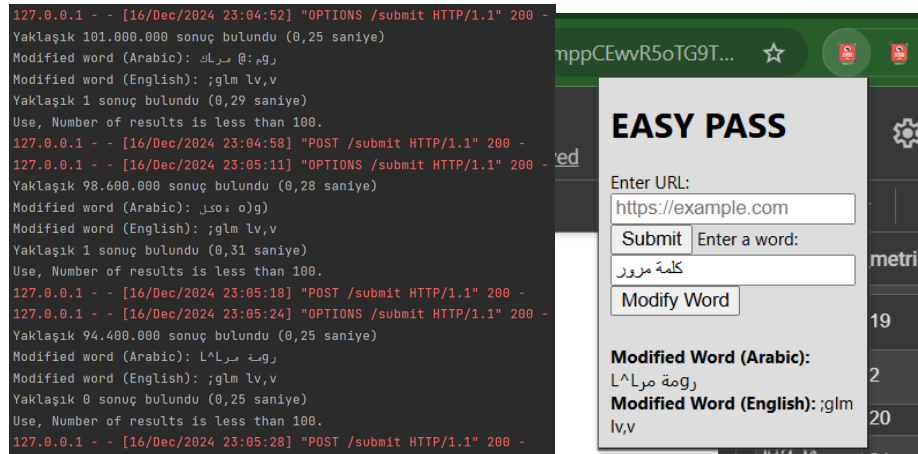


Figure 4.8: Verified number of search results

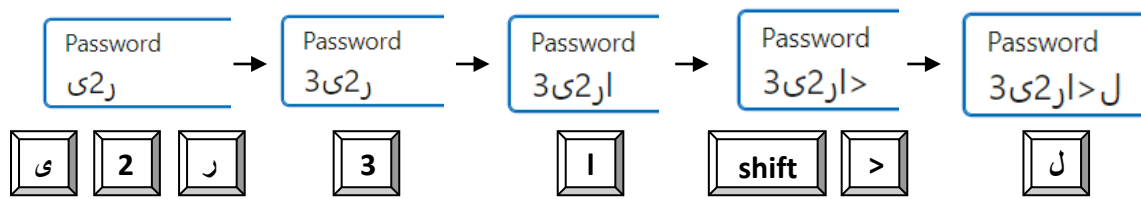


Figure 4.9: Steps of pressing keyboard buttons to type the password

4.3.2 Memorization

Diffusion, randomness, and confusion are essential when defending against password-cracking attempts. Generating passwords based on users' chosen words ensures both user-friendliness and memorability. While these passwords appear random, they are not truly random. A key factor in memorizing these passwords is the similar appearance between the Arabic characters and the replacement symbols, numbers, and Latin letters, especially for users familiar with the Arabizi writing style. By representing Arabic characters with similar-looking replacements, such as "ك" represented as "L^", making the password visually easier to memorize. Furthermore, representing a single character with two symbols or combining Latin characters with symbols effectively increases the password's length, enhancing security.

4.3.3 Password Entropy

To measure the strength of a password against guessing or cracking attempts, entropy provides a numerical value representing the upper bound of the password's security. The calculation of entropy depends on two factors: the character set (charset) and the password length. The character set includes the following:

- Digits (0,1,2,3,4,5,6,7,8,9) = 10 characters
- Lowercase letters (a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z) = 26 characters
- Uppercase letters (A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z) = 26 characters
- Special characters (!@#\$%^&*(),.?:;'"[<>_-=÷×+{}") = 30 characters
- Arabic letters (ا ب ت ث ج ح خ د ذ ر ز س ش ص ض ط ظ ع غ ف ق ك ل م ن ه و ي ئ ؤ ء إ آ) = 35 characters

The total number of possible characters (N) = 10 + 26 + 26 + 30 + 35 = 127

Entropy = $\log_2$ (Number of possible characters)^{Password length}

For a password of length (L) = 10, Entropy = $\log_2(127)^{10} \approx 73.97 \text{ bits}$

While the entropy of the same length password (L) = 10, without Arabic letters:

the total number of possible characters (N) = 10 + 26 + 26 + 30 = 92

Entropy = $\log_2(92)^{10} \approx 49.1 \text{ bits}$.

Passwords with higher entropy have a larger search space, meaning an attacker would have to try more possible combinations. Therefore, higher entropy translates to greater resistance to brute-force attacks and other forms of password cracking.

4.3.4 Brute-Force Attack

Brute force is a method used to crack passwords or other types of security tokens by systematically trying every possible combination of characters until the correct one is found. In a brute force attack, all possible combinations of a password are attempted, ranging from simple numbers or letters to complex combinations of symbols, numbers, and letters. This approach relies on trial and error, often with the help of automated tools to speed up the process.

The scenario here involves an attack carried out with prior knowledge of the algorithms used to generate both the Arabic and English passwords, in order to compare the time taken to crack each password under identical conditions. The same base word was used to generate both the Arabic and English passwords, allowing for a direct comparison of cracking time between the two.

Platforms typically store passwords as hashed values rather than in plain text, using algorithms such as MD5, SHA256, SHA512. Assuming the attacker was able to obtain the hash of the password and identify the hashing algorithm using a hash identifier, they could then initiate the cracking process with a word list.

For this examination, password hashes were saved in files to be used in the cracking attack, and the time taken to crack them was compared. To optimize the process, and due to the limited computing capabilities available, a word list was generated with the exact length and character set of the passwords being tested.

To brute force the Arabic password that contains the Arabic letter (arL), uppercase letter (upL), lowercase letter (loL), digit (D), and symbol (sy) " $L^{\wedge}Jm0$ ", which is

generated from the base word " كلمة ", the attacker must know that the Arabic letters are used in the password, which is not common. Therefore, a special word list must be generated that includes Arabic letters in addition to the uppercase and lowercase Latin characters, numbers, and symbols.

An Arabic word list (arwordlist.txt) was generated that has 3125 password combinations using the tool Crunch in Kali-Linux, to examine the Arabic password. Crunch did not accept Arabic letters (Unicode) until the letters were copy-pasted from an Arabic website. In Figure 4.10, a specified arwordlist.txt was generated with a length of 5 characters, and the character set (charset) consists of 5 types: ' m, L, 0, ^, ل ' for the password that will be cracked to test the duration of cracking. Figure 4.11 shows a sample of the arwordlist.txt.

```
(kali@ruaahamad)-[~]
$ crunch 5 5 mL0^J -o arwordlist.txt
Notice: Detected unicode characters. If you are piping crunch output
to another program such as john or aircrack please make sure that program
can handle unicode input.

Do you want to continue? [Y/n] y
Crunch will now generate the following amount of data: 21875 bytes
0 MB
0 GB
0 TB
0 PB
Crunch will now generate the following number of lines: 3125

crunch: 100% completed generating output
```

Figure 4.10: Generating arwordlist.txt using crunch

arwordlist.txt	
1097 L^JL	1116 L^J^m
1098 L^^0J	1117 L^J^L
1099 L^^J^	1118 L^0^J
1100 L^^JL	1119 L^J^^
1101 L^Jmm	1120 L^J^J
1102 L^JmL	1121 L^JmL
1103 L^Jm0	1122 L^JL
1104 L^Jm^	1123 L^0JL
1105 L^JmJ	1124 L^JL^
1106 L^JLm	1125 L^JL
1107 L^JLL	1126 L^Jmmm
1108 L^JL0	1127 L^JmmL
1109 L^JL^	1128 L^Jmm0
1110 L^JLJ	1129 L^Jmm^
1111 L^0Jm	1130 L^JmmJ
1112 L^0JL	1131 L^JmLm
1113 L^00J	1132 L^JmL0
1114 L^0J^	1133 L^JmL0
1115 L^0J0J	1134 L^JmL^
1116 L^J^m	1135 L^JmLJ
	1136 L^Jm0m
	1137 L^Jm0L

Figure 4.11: Sample of the arwordlist.txt password combinations

0 specifies Attack mode 0 = Straight). As shown in figure 4.14, in this command, md5.txt represents the file containing the hash, and the arwordlist.txt contains the Arabic word list.

```
(kali@ruaahamad)~$ hashcat -m 0 md5.txt arwordlist.txt
hashcat (v6.1.1) starting...

OpenCL API (OpenCL 1.2 pocl 1.6, None+Asserts, LLVM 9.0.1, RELOC, SLEEP, DISTRO, POCL_DEBUG) - Platform #1
[The pocl project]

* Device #1: pthread-Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz, 1423/1487 MB (512 MB allocatable), 2MCU

Minimum password length supported by kernel: 0
Maximum password length supported by kernel: 256

Hashes: 1 digests; 1 unique digests, 1 unique salts
Bitmaps: 16 bits, 65536 entries, 0x0000ffff mask, 262144 bytes, 5/13 rotates
Rules: 1
```

Figure 4.14: Hashcat cracking the md5 hash of the Arabic password

Figure 4.15 shows the result of cracking the MD5 hash of the Arabic password " ل^Jm0 ". With a wordlist containing 3125 passwords, the cracking process took 3 minutes and 8 seconds, or a total of 188 seconds.

```
[s]tatus [p]ause [b]ypass [c]heckpoint [q]uit => Dictionary cache built:
* Filename..: arwordlist.txt
* Passwords.: 3125
* Bytes.....: 21875
* Keyspace..: 3125
* Runtime...: 0 secs

ebfd32984589ba6898f0bca1cc7e29a7:L^Jm0

Session.....: hashcat
Status.....: Cracked
Hash.Name.....: MD5
Hash.Target.....: ebfd32984589ba6898f0bca1cc7e29a7
Time.Started.....: Tue Apr 30 12:17:19 2024 (0 secs)
Time.Estimated...: Tue Apr 30 12:17:19 2024 (0 secs)
Guess.Base.....: File (arwordlist.txt)
Guess.Queue.....: 1/1 (100.00%)
Speed.#1.....: 5152 H/s (0.48ms) @ Accel:1024 Loops:1 Thr:1 Vec:8
Recovered.....: 1/1 (100.00%) Digests
Progress.....: 2048/3125 (65.54%)
Rejected.....: 0/2048 (0.00%)
Restore.Point....: 0/3125 (0.00%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:0-1
Candidates.#1....: mmmmm -> ^LL0J

Started: Tue Apr 30 12:14:13 2024
Stopped: Tue Apr 30 12:17:21 2024
```

Figure 4.15: Hashcat cracking results of the Arabic password

Figure 4.16 shows cracking the SHA 256 hash of the same password " ل^Jm0 " using the arwordlist.txt file by executing the command `hashcat -m 1400 -a 0 sha512ar.txt arwordlist.txt`, where `-m 1400` specifies the hash type

(1400 = SHA256) and `-a 0` indicates Attack Mode 0 (Straight). In this command, `sha512ar.txt` refers to the file containing the hash, and `arwordlist.txt` contains the Arabic word list file.

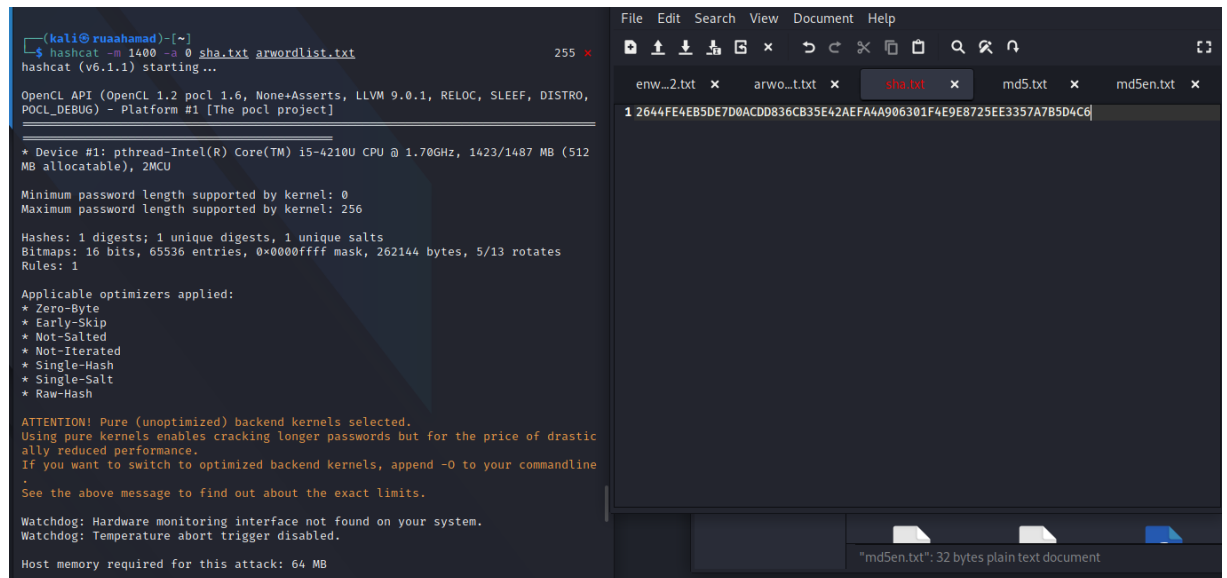


Figure 4.16: Hashcat cracking the sha256 hash of the Arabic password

Figure 4.17 shows the result of cracking the SHA-256 hash of the Arabic password " `L^Jm0` ". Using a word list containing 3125 passwords, the cracking process took 1 minute and 52 seconds, or a total of 112 seconds.

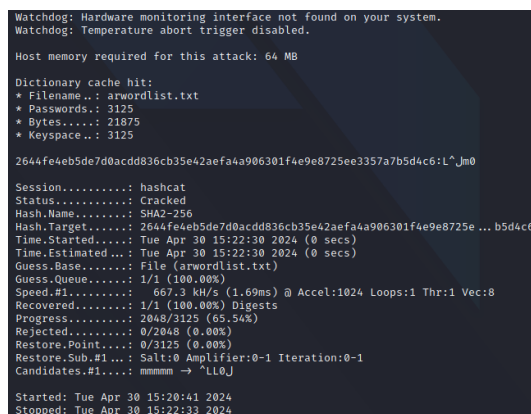


Figure 4.17: Hashcat cracking results of the Arabic password

As shown in Figure 4.18, to crack the English password " `K1^0` ", which contains an uppercase letter (upL), lowercase letter (loL), a digit (D), and a symbol (sy), an English word list named (`enwordlist.txt`) was generated. This list contains 1024 password combinations created using the Crunch tool in kali-linux. The specified

enwordlist.txt was generated with a length of 5 characters and a character set (charset) that includes four types: ' K ', ' l ', ' 0 ', ' ^ '. The aim is to test the duration of the cracking process.

Figure 4.19 shows a sample of the enwordlist.txt. This word list was generated using the most character set types that could be used to the English password (upL, loL, D, and sy).

```
(kali@ruaahamad)-[~]
$ crunch 5 5 lk0^ -o enwordlist.txt
Crunch will now generate the following amount of data: 6144 bytes
0 MB
0 GB
0 TB
0 PB
Crunch will now generate the following number of lines: 1024
crunch: 100% completed generating output
```

Figure 4.18: Generating enwordlist.txt using crunch

307 kl^l0	327 kklk0
308 kl^l^	328 kklk^
309 kl^kl	329 kkl0l
310 kl^kk	330 kkl0k
311 kl^k0	331 kkl00
312 kl^k^	332 kkl0^
313 kl^0l	333 kkl^l
314 kl^0k	334 kkl^k
315 kl^00	335 kkl^0
316 kl^0^	336 kkl^^
317 kl^^l	337 kklkl
318 kl^^k	338 kklkl
319 kl^^0	339 kklkl
320 kl^^^	340 kklkl^
321 kklkl	341 kklkl
322 kklkl	342 kklkl
323 kklkl	343 kklkl
324 kklkl^	344 kklkl^
325 kklkl	345 kklkl
326 kklkl	346 kklkl
327 kklkl	347 kklkl

Figure 4.19: Sample of the enwordlist.txt password combinations

First, the hashes of the password " **kl^^0** " were generated using a hash generator. The **MD5** hash is saved in the file md5en.txt. As shown in figure 4.20 to crack the md5 hash, the Hashcat tool in kali-linux was employed with the command `hashcat -m 0 -a 0 md5en.txt enwordlist.txt`, where `-m 0` specifies Attack mode 0 (Straight), `md5en.txt` is the file containing the hash, and `enwordlist.txt` is the English word list file.

```
(kali@ruaahamad)-[~]
$ hashcat -m 0 -a 0 md5en.txt enwordlist.txt
hashcat (v6.1.1) starting ...

OpenCL API (OpenCL 1.2 pocl 1.6, None+Asserts, LLVM 9.0.1, RELOC, SLEEF, DISTRO, POCL_DEBUG) -
Platform #1 [The pocl project]

* Device #1: pthread-Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz, 1423/1487 MB (512 MB allocatable), 2MCU

Minimum password length supported by kernel: 0
Maximum password length supported by kernel: 256
```

Figure 4.20: Hashcat cracking the md5 hash of the English password

Figure 4.21 shows the result of cracking the md5 hash of the English password " **kl^^0** ". Using a word list containing 1024 passwords, the cracking process took 6 seconds.

```
Dictionary cache hit:
* Filename..: enwordlist.txt
* Passwords.: 1024
* Bytes.....: 6144
* Keyspace..: 1024

The wordlist or mask that you are using is too small.
This means that hashcat cannot use the full parallel power of your device(s).
Unless you supply more work, your cracking speed will drop.
For tips on supplying more work, see: https://hashcat.net/faq/morework

Approaching final keyspace - workload adjusted.

473e3d46a3e3f8b0721cac5bb6fce3b8:kl^^0

Session.....: hashcat
Status.....: Cracked
Hash.Name.....: MD5
Hash.Target.....: 473e3d46a3e3f8b0721cac5bb6fce3b8
Time.Started.....: Tue Apr 30 16:20:01 2024 (0 secs)
Time.Estimated...: Tue Apr 30 16:20:01 2024 (0 secs)
Guess.Base.....: File (enwordlist.txt)
Guess.Queue.....: 1/1 (100.00%)
Speed.#1.....: 471.4 kH/s (0.70ms) @ Accel:1024 Loops:1 Thr:1 Vec:8
Recovered.....: 1/1 (100.00%) Digests
Progress.....: 1024/1024 (100.00%)
Rejected.....: 0/1024 (0.00%)
Restore.Point....: 0/1024 (0.00%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:0-1
Candidates.#1....: lllll → ^^^^^

Started: Tue Apr 30 16:19:57 2024
Stopped: Tue Apr 30 16:20:03 2024
```

Figure 4.21: Hashcat cracking results of the English password

The duration difference between cracking the Arabic password and the English password was significant. To further examine the cracking process, a larger English wordlist named " enwordlist2.txt " was generated, containing the same number of password combinations 3125 as the Arabic word list (Figure 4.22).

```

Dictionary cache hit:
* Filename..: enwordlist2.txt
* Passwords.: 3125
* Bytes.....: 18750
* Keyspace..: 3125

```

Figure 4.22: The size of enwordlist2.txt English password combinations

Cracking the SHA-256 hash of the same password " **K1^0** " was performed using the enwordlist2.txt file by executing the command `hashcat -m 1400 -a 0 sha256enn.txt enwordlist2.txt`, where `-m 1400` specifies the hash type 1400 = SHA-256) and `-a 0` indicates Attack mode 0 (Straight). In this command, `sha256enn.txt` refers to the file containing the hash, `enwordlist2.txt` is the larger English word list file (Figure 4.23).

```

(kali@ruaahamad:~)
$ hashcat -m 1400 -a 0 sha256enn.txt enwordlist2.txt
hashcat (v6.1.1) starting...

OpenCL API (OpenCL 1.2 pocl 1.6, None+Asserts, LLVM 9.0.1, RELOC, SLEEF, DISTRO, POCL_DEBUG) -
Platform #1 [The pocl project]

* Device #1: pthread-Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz, 1423/1487 MB (512 MB allocatabl
e), 2MCU

Minimum password length supported by kernel: 0
Maximum password length supported by kernel: 256

Hashes: 1 digests; 1 unique digests, 1 unique salts
Bitmaps: 16 bits, 65536 entries, 0x0000ffff mask, 262144 bytes, 5/13 rotates
Rules: 1

Applicable optimizers applied:
* Zero-Byte
* Early-Skip
* Not-Salted
* Not-Iterated
* Single-Hash
* Single-Salt
* Raw-Hash

```

Figure 4.23: Hashcat cracking the sha256 hash of the English password

Figure 4.24 shows the result of cracking the SHA-256 hash of the English password " **K1^0** ". Using a word list containing 3125 passwords, the cracking process again took 6 seconds.

```

Watchdog: Hardware monitoring interface not found on your system.
Watchdog: Temperature abort trigger disabled.

Host memory required for this attack: 64 MB

Dictionary cache hit:
* Filename..: enwordlist2.txt
* Passwords.: 3125
* Bytes.....: 18750
* Keyspace..: 3125

4ad2a266ecfe611441c8625fdc14aeba973ba0cca64b4ea68a00ed142773dec8:kl^0

Session.....: hashcat
Status.....: Cracked
Hash.Name.....: SHA2-256
Hash.Target.....: 4ad2a266ecfe611441c8625fdc14aeba973ba0cca64b4ea68a0 ... 73dec8
Time.Started.....: Tue Apr 30 16:31:16 2024 (0 secs)
Time.Estimated.....: Tue Apr 30 16:31:16 2024 (0 secs)
Guess.Base.....: File (enwordlist2.txt)
Guess.Queue.....: 1/1 (100.00%)
Speed.#1.....: 582.5 kH/s (2.32ms) @ Accel:1024 Loops:1 Thr:1 Vec:8
Recovered.....: 1/1 (100.00%) Digests
Progress.....: 2048/3125 (65.54%)
Rejected.....: 0/2048 (0.00%)
Restore.Point.....: 0/3125 (0.00%)
Restore.Sub.#1.....: Salt:0 Amplifier:0-1 Iteration:0-1
Candidates.#1.....: 11111 -> "klm0"

Started: Tue Apr 30 16:31:12 2024
Stopped: Tue Apr 30 16:31:18 2024

```

Figure 4.24: Hashcat cracking results of the English password

Table 4.3 summarizes the character sets used for each password and the corresponding cracking results, highlighting the significant differences in cracking duration.

Table 4.3: Results of cracking hashes

Password	Arabic password L^Jm0		English password K1^^0	
Length	5 char		5 char	
charset	arL	J	upL	K
	upL	L	loL	l
	loL	m	D	0
	D	0	sy	^
	sy	^		
Filename	arwordlist.txt	arwordlist.txt	enwordlist.txt	enwordlist2.txt
Size	3125 Passwords	3125 Passwords	1024 Passwords	3125 Passwords
Bytes	21875	21875	6144	18750
Keyspace	3125	3125	1024	3125
Hash.Name	MD5	SHA2-256	MD5	SHA2-256
Line	1103	1103	319	718
Started:	12:14:13 2024	15:20:41 2024	16:19:57 2024	16:31:12 2024
Stopped:	12:17:21 2024	15:22:33 2024	16:20:03 2024	16:31:18 2024
Time sec	188	112	6	6
Time min	3:08	1:52	0	0

By noticing the lines where each password is presented in the word lists, it was found that the Arabic password "L^Jm0" is located on line 1103 in the arwordlist.txt, while the English password "K1^^0" appears on line 319 in enwordlist.txt and line 718 in enwordlist2.txt. To conduct another comparison, an English password farther down the list was selected for cracking. This time, the SHA-512 hashing algorithm was used.

The Arabic password "L^0Jm" is located in line 1111 in the wordlist arwordlist.txt. It was hashed using the SHA-512 algorithm and saved in the sha512ar.txt file for cracking with the Hashcat command `hashcat -m 1700 -a 0 sha512ar.txt arwordlist.txt`.

Figure 4.25 shows the result of cracking the SHA-512 hash of the Arabic password "ل^0Jm". Using a word list containing 3125 passwords, the cracking process took 2 minutes and 30 seconds, or a total of 150 seconds.

```

Dictionary cache hit:
* Filename..: arwordlist.txt
* Passwords.: 3125
* Bytes.....: 21875
* Keyspace...: 3125

0c3a601b39985b46810379358a0088952bbe268c806f9ae8984808277241c92cd9ec598eb594879ae91eb0544f05b32a7b3cb28373
219fb104ccba5e13840954:L^0Jm

Session.....: hashcat
Status.....: Cracked
Hash.Name.....: SHA2-512
Hash.Target.....: 0c3a601b39985b46810379358a0088952bbe268c806f9ae8984 ... 840954
Time.Started....: Wed May 1 08:06:17 2024 (0 secs)
Time.Estimated...: Wed May 1 08:06:17 2024 (0 secs)
Guess.Base.....: File (arwordlist.txt)
Guess.Queue.....: 1/1 (100.00%)
Speed.#1.....: 474.0 kH/s (1.74ms) @ Accel:1024 Loops:1 Thr:1 Vec:4
Recovered.....: 1/1 (100.00%) Digests
Progress.....: 2048/3125 (65.54%)
Rejected.....: 0/2048 (0.00%)
Restore.Point....: 0/3125 (0.00%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:0-1
Candidates.#1....: mmmmm → ^LL0J

Started: Wed May 1 08:03:49 2024
Stopped: Wed May 1 08:06:19 2024

```

Figure 4.25: Hashcat cracking results of the Arabic password

The English password "Okk1m" is located on line 1405 in the word list enwordlist2.txt, It was hashed using the SHA-512 algorithm and saved in the sha512en.txt file for cracking with the Hashcat command `hashcat -m 1700 -a 0 sha512en.txt enwordlist2.txt`. Figure 4.26 shows the result of cracking the SHA-512 hash of the English password. Using a word list containing 3125 passwords, the cracking process took 6 seconds.

```

Dictionary cache hit:
* Filename..: enwordlist2.txt
* Passwords.: 3125
* Bytes.....: 18750
* Keyspace...: 3125

5b4964ffd47cb6b4ac03cefb3be498e9b94d7a8141b86e3671270995dc79160473ac4466154e5c33719d73d3a874dd7cdd1dbd92c
e8b350576a379a18a9df0c:Okk1m

Session.....: hashcat
Status.....: Cracked
Hash.Name.....: SHA2-512
Hash.Target.....: 5b4964ffd47cb6b4ac03cefb3be498e9b94d7a8141b86e36712 ... a9df0c
Time.Started....: Wed May 1 08:13:44 2024 (0 secs)
Time.Estimated...: Wed May 1 08:13:44 2024 (0 secs)
Guess.Base.....: File (enwordlist2.txt)
Guess.Queue.....: 1/1 (100.00%)
Speed.#1.....: 561.8 kH/s (2.11ms) @ Accel:1024 Loops:1 Thr:1 Vec:4
Recovered.....: 1/1 (100.00%) Digests
Progress.....: 2048/3125 (65.54%)
Rejected.....: 0/2048 (0.00%)
Restore.Point....: 0/3125 (0.00%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:0-1
Candidates.#1....: lllll → ^kk0

Started: Wed May 1 08:13:41 2024
Stopped: Wed May 1 08:13:47 2024

```

Figure 4.26: Hashcat cracking results of the English password

The English password " ^0mk0" is located on line 2233 in the word list enwordlist2.txt. It was hashed using the SHA-512 algorithm and saved in the sha512en.txt file for cracking with the Hashcat command: hashcat -m 1700 -a 0 sha512en.txt enwordlist2.txt.

Figure 4.27 shows the result of cracking the SHA-512 hash of the English password. Using a word list containing 3125 passwords, the cracking process took 6 seconds.

```

Dictionary cache hit:
* Filename..: enwordlist2.txt
* Passwords.: 3125
* Bytes.....: 18750
* Keyspace...: 3125

Approaching final keyspace - workload adjusted.

7392a389793f43fb1544047352dae55cd8d8bc9c32b8a5395aa0ff163d84a19bd3baf1bc4ac75ca3c35fce4cc19f605eb2419eb7e9
feb22ccd976b7cb8975cf9:^0mk0

Session.....: hashcat
Status.....: Cracked
Hash.Name.....: SHA2-512
Hash.Target.....: 7392a389793f43fb1544047352dae55cd8d8bc9c32b8a5395aa ... 975cf9
Time.Started.....: Wed May 1 08:17:54 2024 (0 secs)
Time.Estimated...: Wed May 1 08:17:54 2024 (0 secs)
Guess.Base.....: File (enwordlist2.txt)
Guess.Queue.....: 1/1 (100.00%)
Speed.#1.....: 529.3 kH/s (1.37ms) @ Accel:1024 Loops:1 Thr:1 Vec:4
Recovered.....: 1/1 (100.00%) Digests
Progress.....: 3125/3125 (100.00%)
Rejected.....: 0/3125 (0.00%)
Restore.Point....: 2048/3125 (65.54%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:0-1
Candidates.#1....: ^kkm^ -> mmmmm

Started: Wed May 1 08:17:50 2024
Stopped: Wed May 1 08:17:56 2024

```

Figure 4.27: Hashcat cracking results of the English password

Table 4.4 provides a summary of the character sets used for each password, along with the results of cracking time, highlighting the notable differences in cracking durations.

Table 4.4: Results of cracking sha512 hashes

Password	Arabic password L^0Jm		English password 0kklm, ^0mk	
Length	5 char		5 char	
charset	arL	J	upL	K
	upL	L	loL	l
	loL	m	D	0
	D	0	sy	^
	sy	^		
Filename	arwordlist.txt		Enwordlist2.txt	enwordlist2.t xt

Size	3125 Passwords	3125 Passwords	3125 Passwords
Bytes	21875	18750	18750
Keyspace	3125	3125	3125
Hash.Name	SHA2-512	SHA2-512	SHA2-512
Line	1111	1405	2233
Started:	08:03:49 2024	08:13:41 2024	08:17:50 2024
Stopped:	08:06:19 2024	08:13:47 2024	08:17:56 2024
Time sec	150	6	6
Time min	2:30	0	0

In the following Table 4.5, an expanded comparison is presented that includes additional test passwords generated by regular users using the password generator tool. These user-suggested passwords were cracked (MD5, SHA256, and SHA512 hashing), and the time taken was compared between the Arabic and the English passwords. The same method of comparison was applied, and the average time taken to crack each set of passwords was calculated and documented.

Table 4.5: Results of cracking MD5, SHA512 and SHA256 hashes

Password	Arabic password				English password			
	.D 1 \$		\$.o1 D		1oDo\$1		1oDDoo	
Length	6		6		6		6	
charset	arL	ل	arL	ل				
	upL	D	upL	D	upL	D	upL	D
	loL		loL	o	loL	o	loL	o
	D	1	D	1	D	1	D	1
	sy	. \$	sy	. \$	sy	\$	sy	
Size (number of passwords)	46656		46656		46656		46656	
Bytes	373248		373248		373248		373248	
Hash.Name	MD5		MD5		MD5		MD5	
Started:	14:05:40 2024		14:05:40 2024		14:09:37 2024		14:09:37 2024	
Stopped:	14:05:54 2024		14:05:54 2024		14:09:43 2024		14:09:43 2024	
Time sec	14		14		6		6	
Screenshot	Figure 4.28				Figure 4.29			
Password	Arabic password				English password			
	eل^1س		ل^1سe		eeL^12		L2e^1h	
Length	6		6		6		6	
charset	arL	س ل	arL	ل س				
	upL	L	upL	L	upL	L	upL	L
	loL	e	loL	e	loL	e	loL	e h

	D sy	1 ^	D sy	^	D sy	2 ^	D sy	2 1 ^
Size (number of passwords)	46656		46656		46656		46656	
Bytes	419904		419904		326592		326592	
Hash.Name	SHA2-512		SHA2-512		SHA2-512		SHA2-512	
Started:	16:00:34 2024		16:00:34 2024		16:12:57 2024		16:12:57 2024	
Stopped:	16:01:02 2024		16:01:02 2024		16:13:03 2024		16:13:03 2024	
Time sec	28		28		6		6	
Screenshot	Figure 4.30				Figure 4.31			
Password	Arabic password				English password			
	ؤ%شH4		ؤ%شH4		@^5Ash		^5sA5	
Length	6		6		6		6	
charset	arL	ؤ ش	arL	ؤ ش				
	upL	H	upL	H	upL	A	upL	A
	loL	s	loL		loL	s h	loL	s
	D	4	D	4	D	5	D	5
	sy	%	sy	%	sy	@ ^	sy	\$ ^
Size (number of passwords)	46656		46656		46656		46656	
Bytes	419904		419904		326592		326592	
Hash.Name	SHA2-256		SHA2-256		SHA2-256		SHA2-256	
Started:	18:03:31 2024		18:03:31 2024		18:14:19 2024		18:14:19 2024	
Stopped:	18:03:49 2024		18:03:49 2024		18:14:25 2024		18:14:25 2024	
Time sec	18		18		6		6	
Screenshot	Figure 4.32				Figure 4.33			

```

Dictionary cache hit:
* Filename.: masha.txt
* Passwords.: 46656
* Bytes.....: 373248
* Keyspace..: 46656

af039a36fb0d7c5567e4bf8ef68246e5:.Dl1$
9bdf12784dbb80622171c5acedce8338:$..o1D

Session.....: hashcat
Status.....: Cracked
Hash.Name.....: MD5
Hash.Target.....: mashamd5.txt
Time.Started.....: Sun Aug 25 14:05:52 2024 (0 secs)
Time.Estimated...: Sun Aug 25 14:05:52 2024 (0 secs)
Guess.Base.....: File (masha.txt)
Guess.Queue.....: 1/1 (100.00%)
Speed.#1.....: 1021.7 kH/s (0.50ms) @ Accel:1024 Loops:1 Thr:1 Vec:8
Recovered.....: 2/2 (100.00%) Digests
Progress.....: 45056/46656 (96.57%)
Rejected.....: 0/45056 (0.00%)
Restore.Point...: 43008/46656 (92.18%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:0-1
Candidates.#1...: $1o11. $ <- l.lo

Started: Sun Aug 25 14:05:40 2024
Stopped: Sun Aug 25 14:05:54 2024

```

Figure 4.28: Cracking ".Dl1\$", "\$..o1D"

```

Dictionary cache hit:
* Filename.: masha.txt
* Passwords.: 46656
* Bytes.....: 373248
* Keyspace..: 46656

3203bcd6f65fadc29ff11aca31e36a636:1oDo$1
4d6018a781ebc051a4d6beb39f8e9d5f:1oDDoo

Session.....: hashcat
Status.....: Cracked
Hash.Name.....: MD5
Hash.Target.....: mashamd5.txt
Time.Started.....: Sun Aug 25 14:09:41 2024 (0 secs)
Time.Estimated...: Sun Aug 25 14:09:41 2024 (0 secs)
Guess.Base.....: File (masha.txt)
Guess.Queue.....: 1/1 (100.00%)
Speed.#1.....: 780.7 kH/s (0.83ms) @ Accel:1024 Loops:1 Thr:1 Vec:8
Recovered.....: 2/2 (100.00%) Digests
Progress.....: 26624/46656 (57.06%)
Rejected.....: 0/26624 (0.00%)
Restore.Point...: 24576/46656 (52.67%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:0-1
Candidates.#1...: 11 <- l. $lDlolo

Started: Sun Aug 25 14:09:37 2024
Stopped: Sun Aug 25 14:09:43 2024

```

Figure 4.29: Cracking "1oDo\$1", "1oDDoo"

```

Dictionary cache built:
* Filename...: arbpas2.txt
* Passwords...: 46656
* Bytes.....: 419904
* Keypairs...: 46656
* Runtime....: 10 secs

9e375f89767bf937f6fc453ca36706e2d7f13b77ba559d9a8766b77d5af90f7732d54d4ae2419366f7511c589897d6a866cf
0f060015ad9c9daacc7fe1e1اﺳﻰ
c9d96abb0735cd58d5ea338e20ef3fbd465458093003362ebfb9f5732d76e2b9c122c6b3f0eb3ae2054e5561a15b5bd7bad0383a13
56a5f4cd506701f39c98c1eاﻟﻨﺲ

Session.....: hashcat
Status.....: Cracked
Hash.Name....: SM2-S12
Hash.Target...: a3f52.txt
Time.Started...: Sun Aug 25 16:00:59 2024 (0 secs)
Time.Estimated...: Sun Aug 25 16:00:59 2024 (0 secs)
Guess.Base....: File (arbpas2.txt)
Guess.Queue...: 1/1 (100.00%)
Speed.#1.....: 554.0 kH/s (2.03ms) @ Accel:1024 Loops:1 Thr:1 Vec:4
Recovered.....: 2/2 (100.00%) Digests
Progress.....: 43008/46656 (92.18%)
Recovered.....: 0/43008 (0.00%)
Restore.Point...: 40960/46656 (87.79%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:0-1
Candidates.#1...: 1اﺳﻰ 1اﻟﻨﺲ

Started: Sun Aug 25 16:00:34 2024
Stopped: Sun Aug 25 16:01:02 2024

kali@ruaahamad:~$

```

Figure 4.30: Cracking "e1سL^", "^eسeلL"

```
Host memory required for this attack: 64 MB

Dictionary cache hit:
* Filename..: arpass5.txt
* Passwords.: 46656
* Bytes.....: 419904
* Keypases..: 46656

e12c660c461c443d016dfba6976c95aab0d96e7f4c6b41b5f2772681d12d9535:س4خسH4
9311b33fa62de730db44867cb8f35ba8709b4562b76b447e9aba93b567dc193:H4س4خسH4

Session.....: hashcat
Status.....: Cracked
Hash.Name.....: SHA2-256
Hash.Target.....: sha256.txt
Time.Started.....: Sun Aug 25 18:03:47 2024 (0 secs)
Time.Estimated...: Sun Aug 25 18:03:47 2024 (0 secs)
Guess.Base.....: File (arpass5.txt)
Guess.Queue.....: 1/1 (100.00%)
Speed.#1.....: 758.8 kH/s (1.49ms) @ Accel:1024 Loops:1 Thr:1 Vec:8
Recovered.....: 2/2 (100.00%) Digests
Progress.....: 30720/46656 (65.84%)
Rejected.....: 0/30720 (0.00%)
Restore.Point...: 28672/46656 (61.45%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:0-1
Candidates.#1...: H44س4خس → H4س4خسH4

Started: Sun Aug 25 18:03:31 2024
Stopped: Sun Aug 25 18:03:49 2024
```

Figure 4.32: Cracking "ش% sH4ؤ", "H4%ش4ؤ"

The cracking time required using the full search space (size) was calculated using the `password_metrics.csv` file, which was created to evaluate the threshold of the password popularity. The tested password list in figure 4.34 shows guesses $\geq 10^{14}$, indicating that these are strong passwords.

$$\text{Search Space} = \text{Character Set Size}^{\text{Password Length}}$$

The cracking time is estimated based on the number of guesses and the guessing speed:

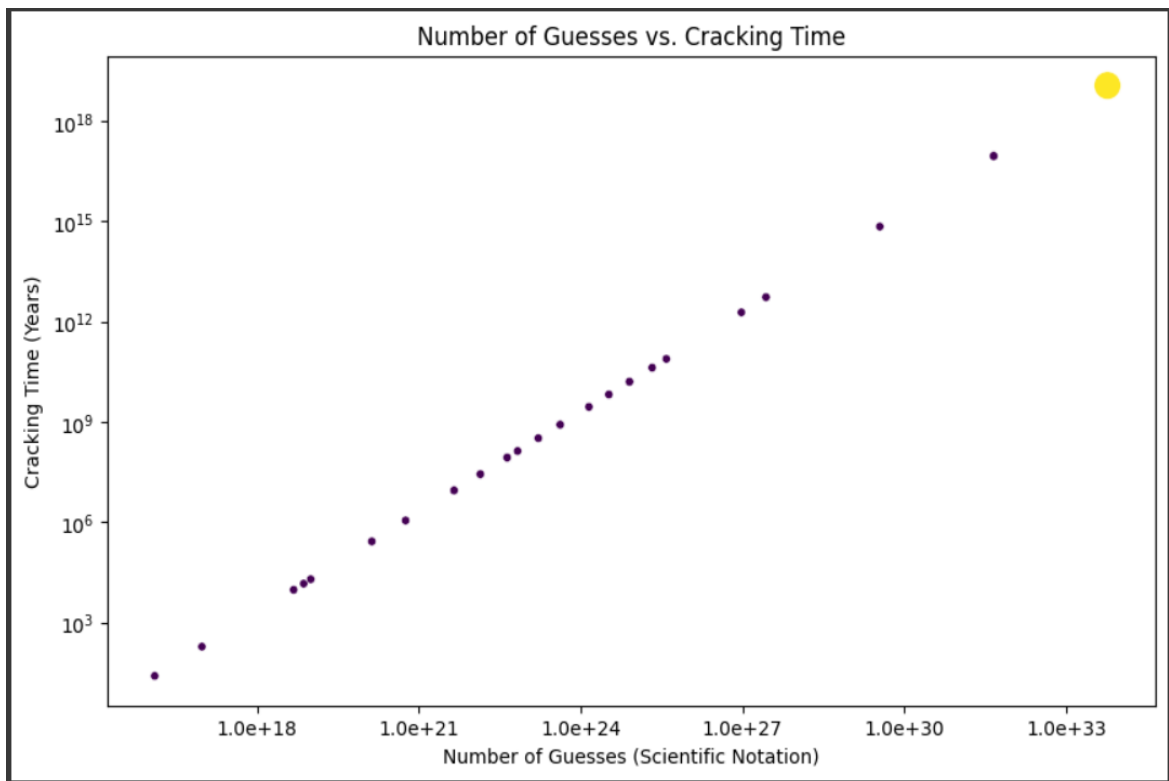


Figure 4.34 Number of guesses vs cracking time

4.3.5 Online password strength checkers

Another password evaluation is provided in this section using different password meters or password strength checkers to compare the effect of adding Arabic characters to passwords with known English passwords.

The Password Meter was used to assess the strength of passwords by counting the requirements, including the number of characters, uppercase letters, lowercase letters, numbers, symbols, and additional criteria. The Latin password "Ru@a0adL" and the Arabic password "Ru@a0adل" (with one added Arabic letter) were tested. Figure 4.35 shows the results: the English password received a strength score of 73 %, while the Arabic password received a score of 81 % . Notably, the Arabic letter "ل" was treated as a symbol.

Test Your Password		Minimum Requirements			
Password:	Ru@a0adL	- Minimum 8 characters in length - Contains 3/4 of the following items: - Uppercase Letters - Lowercase Letters - Numbers - Symbols			
Hide:	☐				
Score:	73%				
Complexity:	Strong				

Test Your Password		Minimum Requirements			
Password:	Ru@a0adL	- Minimum 8 characters in length - Contains 3/4 of the following items: - Uppercase Letters - Lowercase Letters - Numbers - Symbols			
Hide:	☐				
Score:	81%				
Complexity:	Very Strong				

Additions					Type	Rate	Count	Bonus
✓	Number of Characters				Flat	$+(n*4)$	8	+ 32
✓	Uppercase Letters				Cond/Incr	$+(len-n)*2$	2	+ 12
✓	Lowercase Letters				Cond/Incr	$+(len-n)*2$	4	+ 8
✓	Numbers				Cond	$+(n*4)$	1	+ 4
✓	Symbols				Flat	$+(n*6)$	1	+ 6
✓	Middle Numbers or Symbols				Flat	$+(n*2)$	2	+ 4
✓	Requirements				Flat	$+(n*2)$	5	+ 10
Deductions					Type	Rate	Count	Bonus
✓	Letters Only				Flat	-n	0	0
✓	Numbers Only				Flat	-n	0	0
⚠	Repeat Characters (Case Insensitive)				Comp	-	2	- 1
✓	Consecutive Uppercase Letters				Flat	$-(n*2)$	0	0
⚠	Consecutive Lowercase Letters				Flat	$-(n*2)$	1	- 2
✓	Consecutive Numbers				Flat	$-(n*2)$	0	0
✓	Sequential Letters (3+)				Flat	$-(n*3)$	0	0
✓	Sequential Numbers (3+)				Flat	$-(n*3)$	0	0
✓	Sequential Symbols (3+)				Flat	$-(n*3)$	0	0

Additions					Type	Rate	Count	Bonus
✓	Number of Characters				Flat	$+(n*4)$	8	+ 32
✓	Uppercase Letters				Cond/Incr	$+(len-n)*2$	1	+ 14
✓	Lowercase Letters				Cond/Incr	$+(len-n)*2$	4	+ 8
✓	Numbers				Cond	$+(n*4)$	1	+ 4
✓	Symbols				Flat	$+(n*6)$	2	+ 12
✓	Middle Numbers or Symbols				Flat	$+(n*2)$	2	+ 4
✓	Requirements				Flat	$+(n*2)$	5	+ 10
Deductions					Type	Rate	Count	Bonus
✓	Letters Only				Flat	-n	0	0
✓	Numbers Only				Flat	-n	0	0
⚠	Repeat Characters (Case Insensitive)				Comp	-	2	- 1
✓	Consecutive Uppercase Letters				Flat	$-(n*2)$	0	0
⚠	Consecutive Lowercase Letters				Flat	$-(n*2)$	1	- 2
✓	Consecutive Numbers				Flat	$-(n*2)$	0	0
✓	Sequential Letters (3+)				Flat	$-(n*3)$	0	0
✓	Sequential Numbers (3+)				Flat	$-(n*3)$	0	0
✓	Sequential Symbols (3+)				Flat	$-(n*3)$	0	0

Figure 4.35: The password meter

The Password Checker evaluates password strength and also assesses the vulnerability of passwords against two types of password-cracking methods: brute-force attack and dictionary attack. The same passwords from the previous test were used in this checker. Figure 4.36 shows the results. It was again observed that the Arabic letter " ﺝ " was treated as a symbol. The English password received a strength score of 47 %, while the Arabic password received a score of 64 %. The brute-force attack time estimation table shows a significant difference between the two passwords, with the dictionary attack results indicating that the English password is exposed, while the Arabic password remains secure.

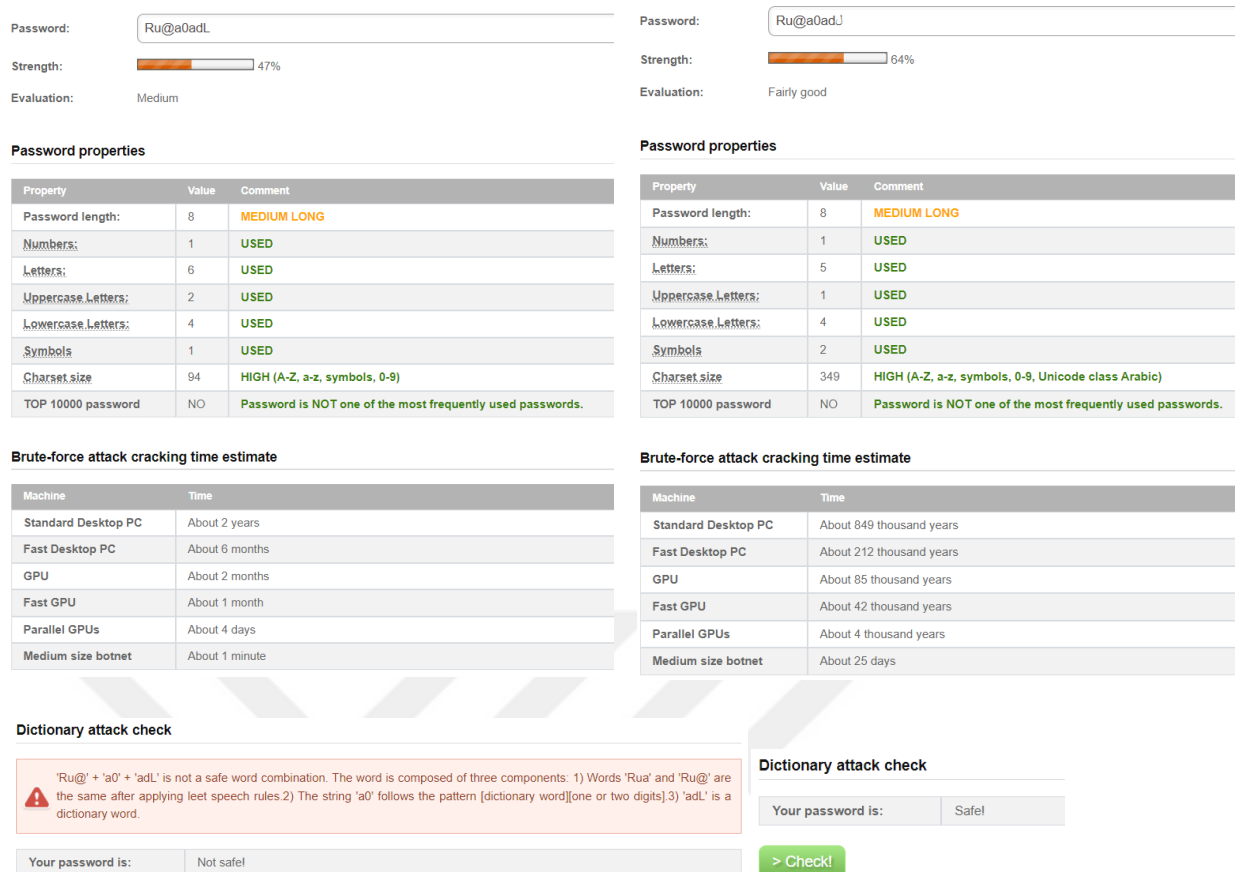


Figure 4.36: The Password Checker

4.4 Summary

This study examines a novel Password Generator and Password Requirements Extractor designed to enhance security, memorability, and usability. With the increasing complexity of password policies across various platforms, users often struggle to create strong yet memorable passwords.

The Password Requirements Extractor clearly provides the necessary requirements to the user, eliminating the need for multiple attempts. It displays the service providers requirements along with classifications and the probability of Arabic password rejection. In Table 4.6, the Windows live email service did not accept the Arabic-characters password " دلاوى3)", as conforms by the Extractor, which indicated that 93.81 % Arabic-characters passwords are not allowed. The Yahoo email service specifies a password length of 8 or more characters and listed the allowed symbols, as confirmed by the Extractor, Which provided a message reflecting the same requirements with prediction of 51.35% that Arabic-characters are not allowed. GMX mailbox service

clearly requires upper and lower case letters, which is the same as the extracted requirements prediction of 59.12%. Facebook, Instagram, and Amazon accept Arabic-characters' password " ر2عل>ل ", as indicated by the Extractor, which found no restrictions on such passwords. Twitter and Github accept Arabic-characters passwords like " رر2عل>ل ", as confirms by the Extractor.

Table 4.6: Extractor summary results

The password	Extractor	Service Provider	Outcome	Proven
دل اوى3)	"password": "Passwords must have at least 8 characters and contain at least two of the following\u003a uppercase letters, lowercase letters, numbers, and symbols."	Microsoft Windows	Password rejected	✓
دل اوى3)	"INVALID_CHARACTERS_IN_PASSWORD": "Can only contain letters, numbers and , ! \" # \$ % & ' () * + , - . \u002F :"	Yahoo	Password rejected	✓
دل اوى3)	A secure password consists of At least eight characters Numbers and letters Upper and lower case letters Special characters	GMX	Password rejected	✓
ر2عل>ل	Not similar to dataset	Facebook	Password accepted	✓
ر2عل>ل	Not similar to dataset	Instagram	Password accepted	✓
ر2عل>ل	Not similar to dataset	Amazon	Password accepted	✓
رر2عل>ل	No specified words found in js code	Twitter	Password accepted	✓
رر2عل>ل	No specified words found in js code	Github	Password accepted	✓

The Arab Password Generator creates both Arabic and English passwords by modifying a base word chosen by the user. The modification process is random, taking advantage of the auto-reordering of characters by mixing right-to-left Arabic characters, Arabizi, numbers, symbols, and Latin characters, without altering the typing order of the base word. For example, the base word " كلمة مرور " might generate passwords such as " L^&ر(م @:لم ", " ر&(و كلمة ", " L^Lمة و)&(", " L^مرهل (g ", " L^لمة مر ", " L^Lوم ر(ة g ", " ;glm lv,v " or " ;glm و)&v ". This diversity increases the time required for dictionary or guessing attack. To ensure that random alterations do not compromise the uniqueness of the password, the popularity (measured by the number of search

results) is checked after each modification. Despite their random appearance, these passwords are still memorable, especially due to the visual similarity appearance between the Arabic characters and the replacement symbols, numbers, and Latin letters, which is particularly useful for users familiar with the Arabizi writing style. Additionally, representing a single character with two symbols or a combination of Latin characters and symbol effectively increases the password's length, further enhancing its security.

Entropy measures the strength of a password by evaluating how difficult it would be to crack through guessing or brute-force attacks. A password using digits, lowercase and uppercase letters, special characters, and Arabic letters has a total character set of 127, leading to an entropy of 73.97 bits for a 10-character password. A similar password without Arabic letters (92-character set) results in lower entropy, around 49.1 bits. Higher entropy increases the number of possible combinations, making it harder for attackers to crack the password. Therefore, including a diverse set of characters, such as Arabic letters, improves security.

The same base word was used to generate both the Arabic and English passwords, allowing for a direct comparison of the time taken to crack each. Table 4.7 demonstrates that cracking Arabic passwords generally takes more time compared to their English counterparts.

Table 4.7: Results of cracking SHA512 hashes

Password		Length	Hash	Size	Bytes	Cracking time	
						Second	Minute
Ar	L^Jm0	5 char	MD5	3125	21875	188	3:08
			SHA256			112	1:52
En	Kl^^0		MD5	1024	6144	6	0
			SHA256	3125	18750		
Ar	L^0Jm	5 char	SHA512	3125	21875	150	2:30
En	0kk1m				18750	6	0
	^0mk						
Ar	.D 1 1\$	6 char	MD5	46656	37324	14	0
	\$.o1 D				8	6	0
En	1oDo\$1						

	1oDDoo						
Ar	e1سٲL^	6 char	SHA512	46656	41990 4	28	0
	^eسٲeٲL						
En	eeL^L2	6 char	SHA512	46656	32659 2	6	0
	L2e^1h						
Ar	شٲsH4	6 char	SHA256	46656	41990 4	18	0
	H4شٲ4						
En	shA^5@	6 char	SHA256	46656	32659 2	6	0
	5A5s@^						

Evaluation of password strength using different password meters to analyze the impact of adding Arabic characters to a password compared to using only English characters in Table 4.8. Two passwords were tested: the English password "Ru@a0adL" and the Arabic password "Ru@a0adل" (with one added Arabic letter).

Using **Password Meter**, the English password achieved a strength score of 73%, while the Arabic password scored 81%, with the Arabic character "ل" being categorized as a symbol.

Similarly, using **Password Checker**, the English password received a strength score of 47%, while the Arabic password scored 64%, with the Arabic character "ل" being recognized as a symbol. The brute-force attack analysis showed a substantial improvement in the Arabic password's resistance, estimating a cracking time of approximately 849 thousand years using a standard desktop PC, compared to two years for the English password. The dictionary attack results further indicated that the English password was vulnerable, whereas the Arabic password remained secure.

Table 4.8: Evaluation of password strength

Scale	Password	Strength	Complexity	Categorized as					
				Ar1	Upl	Lo1	D	Sy	Char set Size
Password Meter	Ru@a0adل	81 %	Very Strong	-	1	4	1	2	
	Ru@a0adL	73 %	Strong	-	2	4	1	1	
Password Checker	Ru@a0adل	64 %	Faily Good	-	1	4	1	2	349
	Ru@a0adL	47 %	Medium	-	2	4	1	1	94

CHAPTER 5

CONCLUSION & RECOMMENDATION

5.1 Conclusion

This study explores the advantages of incorporating Arabic characters into password creation through a newly developed Password Generator, alongside a Password Requirements Extractor designed to simplify password policy requirements for users. By analyzing the strength of both English and Arabic passwords using entropy calculations, brute-force attack simulations, and password strength meters and checkers, the results demonstrate a clear improvement in security. The inclusion of Arabic characters not only enhances password complexity but also helps users in generating memorable passwords.

The Password Requirements Extractor assists users by employing AI to analyze and extract password policies, providing clear guidelines to ensure that passwords meet the required criteria. This process eliminates the repetitive cycle of password rejection and retrying. Due to the lack of a non-English password rejection dataset, an AI language model was utilized to generate a custom dataset, enhancing the extractor's ability to handle multilingual passwords. The extractor identifies password requirements from JS code, using the `nltk.tokenize` module pre-trained. Then, requirements and dataset sentences are numerically represented, allowing for the calculation of similarity scores using TF-IDF, supported by ML. A similarity threshold of 60 or higher determines whether a non-English password will be rejected.

The outcomes of the Password Requirements Extractor depend on the clarity of the service provider's requirement sentences. The extracted requirements are analyzed and classified by the LR trained model with the prediction score (e.g. 77.60% "you can not use Arabic letters"). However, when the service specifies only general requirements, such as length or allowed special characters, without explicitly mentioning specific characters, the AI classifies these as General Policy. The extractor displays the requirements, the classifications, and the predictions to the user. Lastly, if the service provider has no strict password requirements, the extractor displays "No specified words found" or, if no similarity is detected with the dataset, "Not similar to the dataset" indicating that non-English passwords will likely be accepted.

As shown, the extractor is highly beneficial for users, but determining its overall success rate is challenging. Approximately 85% of its outputs or results are accurate, primarily because the results depend on how clearly the service provider discloses its password requirements. Additionally, the absence of a comprehensive dataset for non-English password rejection remains a limitation, as there are insufficient studies on the use of Arabic characters in passwords.

A comprehensive analysis of the Password Generator, comparing English and Arabic passwords, demonstrated that the incorporation of Arabic letters significantly enhances password strength and memorability. Additionally, the automatic shuffling of the password characters removes the need for the user to remember unordered characters. The Password Generator generates both Arabic and English passwords using a novel algorithm that checks the popularity of each password before generating it, ensuring the new password is stronger and less common.

The user's chosen base word is modified to include Arabic characters, Arabizi typing style, numbers, symbols, and Latin characters. This modification is random, so each click generates a new password, offering the user the flexibility to select a password that best suits their needs and is easier to memorize. The modification may also extend the password length and automatically reorder characters by combining right-to-left (Arabic) with left-to-right (English) scripts. This enhances the password's strength and randomness without sacrificing memorability particularly for Arabic-speaking users.

By measuring the entropy of both Arabic and English passwords and comparing the results, it was found that Arabic passwords show significantly higher entropy. Entropy is based on the character set and password length, and since the Arabic character set is combined with the commonly used character sets —lowercase letters, uppercase letters, digits, and special characters—it creates a larger search space. This, in turn, enhances resistance to brute-force attacks and password cracking attempts.

In brute-force attack simulations, Arabic and English password hashes were tested using the Hashcat tool in Kali-Linux with various hashing algorithms (MD5, SHA256, or SHA512). Arabic passwords included an Arabic letter (arL), uppercase letter (upL), lowercase letter (loL), digit (D), and symbol (sy), while English passwords included an uppercase letter (upL), lowercase letter (loL), digit (D), and symbol (sy).

Custom wordlists were generated for both using the Crunch tool in Kali-Linux, though Crunch did not natively accept Arabic letters (Unicode) until the letters were copied and pasted from an Arabic source. Although the Arabic and English passwords had the same length, hashing algorithm, and keyspace, the byte size was not always the same. It was observed that the Arabic wordlists contained more bytes, even with an identical keyspace. Comparing the time required to crack Arabic and English passwords, it became evident that Arabic passwords took significantly longer to crack than their English counterparts.

An online password strength evaluation compared Arabic and English passwords using password meters and checkers. Despite both passwords being of the same length, the inclusion of Arabic letters (often treated as symbols in these tools) led to stronger results for the Arabic passwords. However, due to the lack of precise online Arabic password strength evaluation tools, it is possible that the full strength of Arabic passwords is not always accurately reflected. Nevertheless, the Arabic passwords showed greater character diversity and a much longer cracking time, highlighting their superiority.

The tool's interface is easy to use, provided as a browser extension with input boxes for entering the sign-up link and typing the password, accompanied by submit buttons. The results are printed directly, offering a streamlined and efficient user experience.

Overall, this study demonstrates that incorporating Arabic characters into passwords greatly enhances security, and the combined tools of the Password Generator and Password Requirements Extractor offer users an innovative, user-friendly solution for creating strong, memorable, and compliant passwords.

5.2 Recommendations

This tool targeted the Arab community or anyone who understands and knows the Arabic letters. Due to the similarity between the Arabic alphabet, Persian, and Urdu, by adding their alphabet this tool could be helpful for the Iranians and Urdu-speaking regions.

Persian: ا ب پ ت ث ج چ ح خ د ذ ر ز ژ س ش ص ض ط ظ ع غ ف ق ک گ ل م ن و ه ی

Urdu: ا ب پ ت ث ج چ ح خ د ڈ ذ ر ژ س ش ص ض ط ظ ع غ ف ق ک گ ل م ن و ه ه ی

Expanding the Password Generator and Requirements Extractor to mobile platforms and making it cross-browser compatible could improve accessibility and usability for a wider audience. Given the increasing use of mobile devices for authentication, this enhancement could significantly benefit users.

Furthermore, as this tool generates passwords these passwords might be saved using the browser password manager, but another way to save the passwords is to add a password manager to the tool.

RESOURCES

- A. G. Reinhold. (2006, October) Diceware passphrase home page. [Online]. Available: <http://www.diceware.com/>.
- Alhamed, A., & Bhatia, S. (2021). VowPass: Novel method to generate secure and memorable passwords. *2021 4th International Conference on Signal Processing and Information Security (ICSPIS)*. <https://doi.org/10.1109/icspis53734.2021.9652188>.
- Alomari, R., Martín, M. V., MacDonald, S., Bellman, C., Liscano, R., & Maraj, A. (2017). What Your Brain Says About Your Password: Using Brain-Computer Interfaces to Predict Password Memorability. *Ieee*. <https://doi.org/10.1109/pst.2017.00024>.
- Alomari, R., & Thorpe, J. (2019). On password behaviours and attitudes in different populations. *Journal of Information Security and Applications*, 45, 79–89. <https://doi.org/10.1016/j.jisa.2018.12.008>.
- AlSabah, M., Oligeri, G., & Riley, R. (2018). Your culture is in your password: An analysis of a demographically-diverse password dataset. *Computers & Security*, 77, 427–441. <https://doi.org/10.1016/j.cose.2018.03.014>.
- Bergman, C. A. (1988). A password-protected Power Controller. *Proceedings of the IEEE Southern Tier Technical Conference*. <https://doi.org/10.1109/stier.1988.95486>.
- Billa, J. B., Nawar, A., Shakil, Md. M., & Das, A. K. (2019). Passman: A new approach of password generation and management without storing. *2019 7th International Conference on Smart Computing & Communications (ICSCC)*. <https://doi.org/10.1109/icsc.2019.8843591>.
- B. Ur, J. Bees, S. M. Segreti, L. Bauer, N. Christin, and L. F. Cranor, “Do users’ perceptions of password security match reality?” in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, ser. CHI ’16. San Jose, CA: Association for Computing Machinery, 2016, p. 3748–3760. [Online]. Available: <https://doi.org/10.1145/2858036.2858546>.
- B. Pal, T. Daniel, R. Chatterjee, and T. Ristenpart, “Beyond credential stuffing: Password similarity models using neural networks,” in *2019 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, 2019, pp. 417–434. [Online]. Available: <https://doi.org/10.1109/SP.2019.00056>.
- Chang, Hwang, & Buehrer. (1993). Using smart cards to authenticate passwords. *Proceedings of IEEE International Carnahan Conference on Security Technology CCST-94*. <https://doi.org/10.1109/ccst.1993.386808>.
- Coisel, I., Sanchez, I., & Galbally, J. (2017). Divide, recombine and conquer: Syntactic patterns-reassembly algorithm applied to password guessing process. *2017 International*

Carnahan Conference on Security Technology (ICCST).
<https://doi.org/10.1109/ccst.2017.8167849>.

Cui, X., Li, C., Qin, Y., & Ding, Y. (2020). A password strength evaluation algorithm based on sensitive personal information. 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom).
<https://doi.org/10.1109/trustcom50675.2020.00211>

Doucek, P., Pavlíček, L., Sedláček, J., & Nedomová, L. (2020). Adaptation of password strength estimators to a non-English environment—the Czech experience. *Computers & Security*, 95, 101757. <https://doi.org/10.1016/j.cose.2020.101757>.

Diffie, W., & Hellman, M. (1976). New Directions in cryptography. *IEEE Transactions on Information Theory*, 22(6), 644–654. <https://doi.org/10.1109/tit.1976.1055638>.

Davis, D. K., Chowdhury, M. M., & Rifat, N. (2022). Password security: What are we doing wrong? 2022 IEEE International Conference on Electro Information Technology (eIT). <https://doi.org/10.1109/eit53891.2022.9814059>.

Ezugwu, A., Ukwandu, E., Ugwu, C., Ezema, M., Olebara, C., Ndunagu, J., Ofusori, L., & Ome, U. (2023). Password-based authentication and the experiences of end users. *Scientific African*, 21. <https://doi.org/10.1016/j.sciaf.2023.e01743>

Furnell, S. (2018). Assessing website password practices – over a decade of progress? *Computer Fraud & Security*, 2018(7), 6–13. [https://doi.org/10.1016/s1361-3723\(18\)30063-0](https://doi.org/10.1016/s1361-3723(18)30063-0)

Furnell, S. (2022). Assessing website password practices – unchanged after fifteen years? *Computers & Security*, 120, 102790. <https://doi.org/10.1016/j.cose.2022.102790>

Gan, X., Li, D., & Chen, H. (2022). Analysis of words in passwords from three different countries. 2022 IEEE 10th Joint International Information Technology and Artificial Intelligence Conference (ITAIC). <https://doi.org/10.1109/itaic54216.2022.9836812>.

Garg, V., & Ahuja, L. (2019). Password guessing using Deep Learning. 2019 2nd International Conference on Power Energy, Environment and Intelligent Control (PEEIC). <https://doi.org/10.1109/peeic47157.2019.8976614>.

Glory, F. Z., Ul Aftab, A., Tremblay-Savard, O., & Mohammed, N. (2019). Strong password generation based on user inputs. 2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON). <https://doi.org/10.1109/iemcon.2019.8936178>

Guo, Y., Zhang, Z., & Ya-Jun, G. (2019). Optiwords: A new password policy for creating memorable and strong passwords. *Computers & Security*, 85, 423–435. <https://doi.org/10.1016/j.cose.2019.05.015>.

- Güven, E. Y., Boyacı, A., & Aydın, M. A. (2022). A novel password policy focusing on altering user password selection habits: A statistical analysis on breached data. *Computers & Security*, 113, 102560. <https://doi.org/10.1016/j.cose.2021.102560>.
- Hash Generator. (2024). <https://www.browserling.com/tools/all-hashes>.
- Han, W., Xu, M., Zhang, J., Wang, C., Zhang, K., & Wang, X. S. (2021). *transPCFG*: Transferring the grammars from short passwords to guess long passwords effectively. *IEEE Transactions on Information Forensics and Security*, 16, 451–465. <https://doi.org/10.1109/tifs.2020.3003696>.
- He, X., Cheng, H., Xie, J., Wang, P., & Liang, K. (2022). Passtrans: An improved password reuse model based on Transformer. ICASSP 2022 - 2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). <https://doi.org/10.1109/icassp43922.2022.9746731>
- Hubbard, J., Bendiab, G., & Shiaeles, S. (2022). IPASS: A novel open-source intelligence password scoring system. 2022 IEEE International Conference on Cyber Security and Resilience (CSR). <https://doi.org/10.1109/csr54599.2022.9850311>.
- Imamaliyev, A., & Khudoykulov, Z. (2021). Analysis Password-based Authentication Systems with Password Policy. *2021 International Conference on Information Science and Communications Technologies (ICISCT)*. <https://doi.org/10.1109/icisct52966.2021.9670312>.
- Jiang, X., Sun, X., & Liu, Q. (2022). Password guessing attack based on probabilistic context free algorithm. 2022 IEEE 8th International Conference on Computer and Communications (ICCC). <https://doi.org/10.1109/iccc56324.2022.10065766>.
- Kubariev, O., Piatykov, O., Pronina, O., & Levytska, T. (2023). The research on methods for generating random passwords. *2023 IEEE International Conference on Information and Telecommunication Technologies and Radio Electronics (UkrMiCo)*. <https://doi.org/10.1109/ukrmico61577.2023.10380416>.
- Lamport, L. (1981). Password authentication with insecure communication. *Communications of the ACM*, 24(11), 770–772. <https://doi.org/10.1145/358790.358797>.
- Leonhard, M. D., & Venkatakrishnan, V. N. (2007). A comparative study of three random password generators. *2007 IEEE International Conference on Electro/Information Technology*. <https://doi.org/10.1109/eit.2007.4374533>.
- Liu, Y., Xia, Z., Yi, P., Yao, Y., Xie, T., Wang, W., & Zhu, T. (2018). GENPass: A general deep learning model for password guessing with PCFG rules and adversarial generation. 2018 IEEE International Conference on Communications (ICC). <https://doi.org/10.1109/icc.2018.8422243>.

Li, S., Wang, Z., Zhang, R., Wu, C., & Luo, H. (2023). Mangling rules generation with density-based clustering for password guessing. *IEEE Transactions on Dependable and Secure Computing*, 20(5), 3588–3600. <https://doi.org/10.1109/tdsc.2022.3217002>.

Lyu, S., Yao, Q., & Song, J. (2022). AVOIDPWD: A mnemonic password generation strategy based on keyboard transformation. *China Communications*, 19(10), 92–101. <https://doi.org/10.23919/jcc.2022.00.027>.

Maoneke, P. B., Flowerday, S., & Isabirye, N. (2020). Evaluating the strength of a multilingual passphrase policy. *Computers & Security*, 92, 101746. <https://doi.org/10.1016/j.cose.2020.101746>.

Messier, R. (2021). *CEH V11 Certified Ethical Hacker Study Guide*. Sybex.

Mori, K., Watanabe, T., Zhou, Y., Akiyama Hasegawa, A., Akiyama, M., & Mori, T. (2019). Comparative analysis of three language spheres: Are linguistic and cultural differences reflected in password selection habits? *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 159–171. <https://doi.org/10.1109/eurospw.2019.00025>.

Murray, H., & Malone, D. (2018). Exploring the impact of password dataset distribution on guessing. *2018 16th Annual Conference on Privacy, Security and Trust (PST)*. <https://doi.org/10.1109/pst.2018.8514194>.

O’Gorman, L. (2003). Comparing passwords, tokens, and biometrics for user authentication. *Proceedings of the IEEE*, 91(12), 2021–2040. <https://doi.org/10.1109/jproc.2003.819611>.

Password Meter. (2024). <https://passwordmeter.com/>.

Password Checker. (2024). <http://password-checker.online-domain-tools.com/>.

Pereira, D., Ferreira, J. F., & Mendes, A. (2020). Evaluating the accuracy of password strength meters using off-the-shelf guessing attacks. *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. <https://doi.org/10.1109/issrew51248.2020.00079>.

R. Ganesan and C. Davies, “A new attack on random pronounceable password generators,” in *Proc. 17th NIST-NCSC National Computer Security Conference*, 1994.

R. L. Brotman, *Department of Defense Password Management Guideline, CSC-STD-002-85*, Department of Defense Computer Security Center, 1985.

Renaud, K., Otondo, R., & Warkentin, M. (2019). “this is the way ‘I’ create my passwords” ... does the endowment effect deter people from changing the way they

create their passwords? *Computers & Security*, 82, 241–260. <https://doi.org/10.1016/j.cose.2018.12.018>.

Sælensminde, K., & Boonjing, V. (2010). A simple password less authentication system for web sites. *2010 Seventh International Conference on Information Technology: New Generations*. <https://doi.org/10.1109/itng.2010.154>

Safdar, S., & Hassan, M. F. (2010). Moving towards two dimensional passwords. *2010 International Symposium on Information Technology*. <https://doi.org/10.1109/itsim.2010.5561566>

Shen, C., Yu, T., Xu, H., Yang, G., & Guan, X. (2016). User practice in password security: An empirical study of real-life passwords in the wild. *Computers & Security*, 61, 130–141. <https://doi.org/10.1016/j.cose.2016.05.007>

Sarkar, S., & Nandan, M. (2022). Password strength analysis and its classification by applying machine learning based techniques. *2022 Second International Conference on Computer Science, Engineering and Applications (ICCSEA)*. <https://doi.org/10.1109/iccsea54677.2022.9936117>.

Shin, Y., & Woo, S. S. (2022). PasswordTensor: Analyzing and explaining password strength using tensor decomposition. *Computers & Security*, 116, 102634. <https://doi.org/10.1016/j.cose.2022.102634>.

Song, J., Wang, D., Yun, Z., & Han, X. (2019). Alphapwd: A password generation strategy based on mnemonic shape. *IEEE Access*, 7, 119052–119059. <https://doi.org/10.1109/access.2019.2937030>.

Thai, B. L., & Tanaka, H. (2024). A statistical Markov-based password strength meter. *Internet of Things*, 25, 101057. <https://doi.org/10.1016/j.iot.2023.101057>.

Voyiatzis, A. G., Fidas, C. A., Serpanos, D. N., & Avouris, N. M. (2011). An empirical study on the web password strength in Greece. *2011 15th Panhellenic Conference on Informatics*. <https://doi.org/10.1109/pci.2011.6>

Weiss, K. P. (1990). When a password is not a password (Computer Resource Security). *IEEE International Carnahan Conference on Security Technology, Crime Countermeasures*. <https://doi.org/10.1109/ccst.1990.111393>.

Wu, Y., Wan, X., Guan, X., Ji, T., & Ye, F. (2022). PGTCN: A novel password-guessing model based on Temporal Convolution Network. *SSRN Electronic Journal*. <https://doi.org/10.2139/ssrn.4084713>.

Yang, S., Ji, S., & Beyah, R. (2018). DPPG: a Dynamic Password Policy Generation System. *IEEE Transactions on Information Forensics and Security*, 13(3), 545–558. <https://doi.org/10.1109/tifs.2017.2737971>.

- Yan, S.-H., & Ku, C. C.-Y. (2023). Using language-specific input methods and pronunciation rules to improve the guesses of passwords. *Journal of Information Security and Applications*, 77, 103588. <https://doi.org/10.1016/j.jisa.2023.103588>.
- Zhang, X., Zhang, X., Hu, J., & Zhu, Y. (2023). A new targeted online password guessing algorithm based on Old password. *2023 26th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*. <https://doi.org/10.1109/cscwd57460.2023.10152712>.
- Zhou, E., Peng, Y., Shao, G., Deng, F., Miao, Y., & Fan, W. (2024). Password cracking using chunk similarity. *Future Generation Computer Systems*, 150, 380–394. <https://doi.org/10.1016/j.future.2023.09.013>.
- Zhang, T., Cheng, Z., Qin, Y., Li, Q., & Shi, L. (2020). Deep learning for password guessing and password strength evaluation, a survey. *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. <https://doi.org/10.1109/trustcom50675.2020.00155>.
- Zimmermann, V., & Gerber, N. (2020). The password is dead, long live the password – a laboratory study on user perceptions of authentication schemes. *International Journal of Human-Computer Studies*, 133, 26–44. <https://doi.org/10.1016/j.ijhcs.2019.08.006>.

ANNEXES

7.1 Appendix A. Dataset and Codes

A.1 labeled arbarb.Json

```
[{"text": "Only use letters, numbers, and common punctuation characters.", "label": 1 },
  {"text": "Passwords must have at least 8 characters and contain at least two of the following: uppercase letters, lowercase letters, numbers, and symbols.", "label": 1 },
  {"text": "INVALID_IDENTIFIER:Error: Invalid identifier.", "label": 0 },
  {"text": "The quick brown fox jumps over the lazy dog.", "label": 0 },
  {"text": "I enjoy reading books during my free time.", "label": 0 },
  {"text": "This is a random sentence with no specific meaning.", "label": 0 },
  {"text": "The password contains characters that aren't allowed.", "label": 1 },
  {"text": "Make sure it's at least 15 characters OR at least 8 characters including a number and a lowercase letter.", "label": 1 },
  {"text": "Please use only letters, numbers, and common punctuation characters.", "label": 1 },
  {"text": "Please use English characters only.", "label": 1 },
  {"text": "Generate a robust password using a combination of letters, digits, and symbols.", "label": 1 },
  {"text": "Utilize only letters, digits, and standard punctuation marks.", "label": 1 },
  {"text": "She sells seashells by the seashore.", "label": 0 },
  {"text": "A journey of a thousand miles begins with a single step.", "label": 0 },
  {"text": "The early bird catches the worm.", "label": 0 },
  {"text": "To be or not to be, that is the question.", "label": 0 },
  {"text": "All that glitters is not gold.", "label": 0 },
  {"text": "Passwords need to be a minimum of 8 characters and must include at least two of the following: uppercase
```

```

letters, lowercase letters, digits, and symbols.", "label": 1
},
    {"text": "The password contains disallowed
characters.", "label": 1 },
    {"text": "TOO_MANY_REPEATED_CHARACTERS_IN_PASSWORD:Too
many repeating characters.", "label": 0 },
    {"text": "Ensure it's either 15 characters long OR a
minimum of 8 characters with at least one number and one
lowercase letter.", "label": 1 },
    {"text": "Kindly use only letters, digits, and standard
punctuation marks.", "label": 1 },
    {"text": "Please restrict the usage to English characters
only.", "label": 1 },
    {"text": "PASSWORD_LENGTH_INVALID:Your password isn't
strong enough, try making it longer.", "label": 0 },
    {"text": "You must create a secure password with a blend
of letters, numbers, and symbols.", "label": 1 },
    {"text": "Limit usage to letters, numbers, and common
punctuation marks exclusively.", "label": 1 },
    {"text": "Passwords must be a minimum of 8 characters and
include at least two of the following: capital letters,
lowercase letters, digits, and symbols.", "label": 1 },
    {"text": "The password contains unauthorized
characters.", "label": 1 },
    {"text": "It should be either 15 characters long OR at
least 8 characters with a number and a lowercase
letter.", "label": 1 },
    {"text": "Utilize exclusively English characters,
please.", "label": 1 },
    {"text": "Utilize only letters, digits, and common
punctuation marks.", "label": 1 },
    {"text": "Passwords must be at least 8 characters in
length and include at least two of the following: uppercase
letters, lowercase letters, digits, and symbols.", "label": 1
},
    {"text": "The password contains prohibited characters.",
"label": 1 },
    {"text": "PASSWORD_TOO_SHORT: Must contain at least 8
characters.", "label": 0 },
    {"text": "Ensure it's either 15 characters in length OR a
minimum of 8 characters with both a number and a lowercase
letter.", "label": 1 },
    {"text": "Please restrict usage to English characters
solely.", "label": 1 },
    {"text": "Kindly use English characters
exclusively.", "label": 1 },

```

```

    {"text": "Only utilize letters, numbers, and standard
punctuation.", "label": 1 },
    {"text": "Passwords need to be a minimum of 8 characters
and must include at least two of the following: uppercase,
lowercase, digits, and symbols.", "label": 1 },
    {"text": "PASSWORD_CONTAINS_USERNAME: Can't contain
personal details or usernames.", "label": 0 },
    {"text": "The password contains restricted characters.",
"label": 1 },
    {"text": "Ensure it's either 15 characters long OR a
minimum of 8 characters with a number and a lowercase
letter.", "label": 1 },
    {"text": "INVALID_PHONE_TYPE: We don't support this
number, provide another one.", "label": 0 },
    {"text": "Kindly employ only letters, numbers, and common
punctuation marks.", "label": 1 },
    {"text": "Please restrict usage to English characters
solely.", "label": 1 },
    {"text": "Limit usage to letters, numbers, and common
punctuation marks exclusively.", "label": 1 },
    {"text": "PASSWORD_CONTAINS_NAME: Can't contain personal
details or usernames.", "label": 0 },
    {"text": "An apple a day keeps the doctor away.",
"label": 0 },
    {"text": "Actions speak louder than words.", "label": 0 },
    {"text": "Passwords must be a minimum of 8 characters and
include at least two of the following: capital letters,
lowercase letters, digits, and symbols.", "label": 1 },
    {"text": "The password contains unauthorized
characters.",
"label": 1 },
    {"text": "It should be either 15 characters long OR at
least 8 characters with a number and a lowercase
letter.", "label": 1 },
    {"text": "Utilize exclusively English characters,
please.", "label": 1 },
    {"text": "Utilize only letters, digits, and common
punctuation marks.", "label": 1 },
    {"text": "Beauty is in the eye of the beholder.",
"label": 0 },
    {"text": "Better late than never.", "label": 0 },
    {"text": "Birds of a feather flock together.", "label": 0
},
    {"text": "INVALID_PHONE_NUMBER: This phone number doesn't
look right.", "label": 0 },
    {"text": "Passwords must be at least 8 characters in
length and include at least two of the following: uppercase

```

```

letters, lowercase letters, digits, and symbols.", "label": 1
},
    {"text": "The password contains prohibited characters.",
     "label": 1 },
    {"text": "Ensure it's either 15 characters in length OR a
minimum of 8 characters with both a number and a lowercase
letter.", "label": 1 },
    {"text": "WEAK_PASSWORD: Not strong enough. Try adding
characters.", "label": 0 },
    {"text": "Don't count your chickens before they hatch.",
"label": 0 },
    {"text": "Every cloud has a silver lining.", "label": 0
},
    {"text": "Good things come to those who wait.", "label":
0 },
    {"text": "Honesty is the best policy.", "label": 0 },
    {"text": "Please restrict usage to English characters
solely.", "label": 1 },
    {"text": "Kindly use English characters exclusively.",
"label": 1 },
    {"text": "Generate a robust password with a combination
of letters, numbers, and symbols.", "label": 1 },
    {"text": "Hope for the best, but prepare for the worst.",
"label": 0 },
    {"text": "It's always darkest before the dawn.", "label":
0 },
    {"text": "Only utilize letters, numbers, and standard
punctuation.", "label": 1 },
    {"text": "Passwords need to be a minimum of 8 characters
and must include at least two of the following: uppercase,
lowercase, digits, and symbols.", "label": 1 },
    {"text": "The password contains restricted characters.",
     "label": 1 },
    {"text": "MODERATE_PASSWORD: Adding characters can
increase strength.", "label": 0 },
    {"text": "Keep your friends close and your enemies
closer.",
"label": 0 },
    {"text": "Knowledge is power.", "label": 0 },
    {"text": "Please enter a unique username.", "label": 0 },
    {"text": "Usernames cannot contain special characters.",
"label": 0 },
    {"text": "Your username should be between 5 and 15
characters long.", "label": 0 },
    {"text": "Laughter is the best medicine.", "label": 0 },

```

```

    {"text": "Ensure it's either 15 characters long OR a
minimum of 8 characters with a number and a lowercase
letter.",
    "label": 1 },
    {"text": "Kindly employ only letters, numbers, and common
punctuation marks.", "label": 1 },
    {"text": "Please restrict usage to English characters
solely.", "label": 1 },
    {"text": "Limit usage to letters, numbers, and common
punctuation marks exclusively.", "label": 1 },
    {"text": "INVALID_EMAIL: Make sure you use your full email
address, including an \"@\" sign and a domain.", "label": 0 },
    {"text": "Passwords must be a minimum of 8 characters and
include at least two of the following: capital letters,
lowercase letters, digits, and symbols.", "label": 1 },
    {"text": "FIELD_TOO_LONG: This is too long.", "label": 0
},
    {"text": "The password contains unauthorized
characters.",
"label": 1 },
    {"text": "Usernames are case-sensitive.", "label": 0 },
    {"text": "Choose a username that is easy for you to
remember.", "label": 0 },
    {"text": "Your username cannot be the same as your
email.",
"label": 0 },
    {"text": "It should be either 15 characters long OR at
least 8 characters with a number and a lowercase letter.",
"label": 1 },
    {"text": "Utilize exclusively English characters,
please.",
"label": 1 },
    {"text": "Utilize only letters, digits, and common
punctuation marks.", "label": 1 },
    {"text": "Usernames must not contain spaces.", "label": 0
},
    {"text": "This username is already taken.", "label": 0 },
    {"text": "Make sure your username is not offensive.",
"label": 0 },
    {"text": "Usernames can include letters, numbers, and
underscores.", "label": 0 },
    {"text": "Passwords must be at least 8 characters in
length and include at least two of the following: uppercase
letters, lowercase letters, digits, and symbols.", "label": 1
},
    {"text": "The password contains prohibited characters.",
"label": 1 },

```

```

        {"text": "Please enter your date of birth in MM/DD/YYYY
format.", "label": 0 },
        {"text": "Your age must be above 18 to
register.", "label": 0 },
        {"text": "Birthdate fields cannot be left
empty.", "label": 0 },
        {"text": "The selected date is not valid.", "label": 0 },
        {"text": "Ensure the birth year is accurate.", "label": 0
},
        {"text": "We use your birthday to verify your identity.",
"label": 0 },
        {"text": "Your birthday cannot be in the future.",
"label": 0 },
        {"text": "Enter a valid birth date to proceed.", "label":
0 },
        {"text": "Ensure it's either 15 characters in length OR a
minimum of 8 characters with both a number and a lowercase
letter.", "label": 1 },
        {"text": "CANNOT_END_WITH_SPECIAL_CHARACTER: Your
username has to end with a letter or a number.", "label": 0
},
        {
            {"text": "Please restrict usage to English characters
solely.", "label": 1 },
            {"text": "Kindly use English characters exclusively.",
"label": 1 },
            {"text": "Only utilize letters, numbers, and standard
punctuation.", "label": 1 },
            {"text": "You must confirm your birthday for age
verification.", "label": 0 },
            {"text": "Passwords need to be a minimum of 8 characters
and must include at least two of the following: uppercase,
lowercase, digits, and symbols.", "label": 1 },
            {"text": "Date of birth should match the format
provided.", "label": 0 },
            {"text": "Please provide a valid email address.",
"label": 0 },
            {"text": "Emails must follow the format
username@domain.com.", "label": 0 },
            {"text": "The password contains restricted characters.",
"label": 1 },
            {"text": "Ensure it's either 15 characters long OR a
minimum of 8 characters with a number and a lowercase
letter.", "label": 1 },
            {"text": "CANNOT_HAVE_MORE_THAN_ONE_PERIOD: You can't
have more than one '.' in your username.", "label": 0 },

```

```

    {"text": "Kindly employ only letters, numbers, and common
punctuation marks.", "label": 1 },
    {"text": "Please restrict usage to English characters
solely.", "label": 1 },
    {"text": "Limit usage to letters, numbers, and common
punctuation marks exclusively.", "label": 1 },
    {"text": "Passwords must be a minimum of 8 characters and
include at least two of the following: capital letters,
lowercase letters, digits, and symbols.", "label": 1 },
    {"text": "Your email address will be used for account
recovery.", "label": 0 },
    {"text": "The email entered is already associated with
another account.", "label": 0 },
    {"text": "Ensure the email address does not contain
invalid characters.", "label": 0 },
    {"text": "This email domain is not supported.", "label": 0
},
    {"text": "Please confirm your email address.", "label": 0
},
    {"text": "Email addresses are case-sensitive.", "label": 0
},
    {"text": "An email verification link will be sent to your
inbox.", "label": 0 },
    {"text": "The password contains unauthorized
characters.", "label": 1 },
    {"text": "NEED_AT_LEAST_ONE_ALPHA: Please use at least
one letter in your username.", "label": 0 },
    {"text": "It should be either 15 characters long OR at
least 8 characters with a number and a lowercase letter.",
"label": 1 },
    {"text": "Utilize exclusively English characters,
please.", "label": 1 },
    {"text": "Do not use temporary email addresses for
registration.", "label": 0 },
    {"text": "Your profile information must be accurate.",
"label": 0 },
    {"text": "Please review and accept the terms and
conditions.", "label": 0 },
    {"text": "You can update your contact details at any
time.",
"label": 0 },
    {"text": "Provide a recovery email for account safety.",
"label": 0 },
    {"text": "Utilize only letters, digits, and common
punctuation marks.", "label": 1 },
    {"text": "Passwords must be at least 8 characters in
length and include at least two of the following: uppercase

```

```

letters, lowercase letters, digits, and symbols.", "label": 1
},
    {"text": "The phone number is optional but recommended.",
"label": 0 },
    {"text": "Your profile picture must not exceed 5MB.",
"label": 0 },
    {"text": "Use the settings menu to manage your account
preferences.", "label": 0 },
    {"text": "Two-factor authentication is recommended for
security.", "label": 0 },
    {"text": "Remember to save your changes before exiting.",
"label": 0 },
    {"text": "The password contains prohibited characters.",
"label": 1 },
    {"text":
"CANNOT_START_WITH_SPECIAL_CHARACTER_OR_NUMBER:Your username
has to start with a letter.", "label": 0 },
    {"text": "Ensure it's either 15 characters in length OR a
minimum of 8 characters with both a number and a lowercase
letter.", "label": 1 },
    {"text": "Please restrict usage to English characters
solely.", "label": 1 },
    {"text": "Provide a valid shipping address for order
fulfillment.", "label": 0 },
    {"text": "Your billing information will be encrypted for
security.", "label": 0 },
    {"text": "Add an alternate contact number for account
recovery.", "label": 0 },
    {"text": "Please upload a document for identity
verification.", "label": 0 },
    {"text": "Choose a memorable security question.",
"label": 0 },
    {"text": "Enter the OTP sent to your registered mobile
number.", "label": 0 },
    {"text": "Kindly use English characters exclusively.",
"label": 1 },
    {"text": "Only utilize letters, numbers, and standard
punctuation.", "label": 1 },
    {"text": "Passwords need to be a minimum of 8 characters
and must include at least two of the following: uppercase,
lowercase, digits, and symbols.", "label": 1 },
    {"text": "CONSECUTIVE_SPECIAL_CHARACTERS_NOT_ALLOWED: You
can't have more than one \. ' or \_ ' in a row.", "label": 0 },
    {"text": "The password contains restricted characters.",
"label": 1 },

```

```

    {"text": "Ensure it's either 15 characters long OR a
minimum of 8 characters with a number and a lowercase
letter.",
"label": 1 },
    {"text": "Kindly employ only letters, numbers, and common
punctuation marks.", "label": 1 },
    {"text": "The CAPTCHA verification is required to
proceed.",
"label": 0 },
    {"text": "Select your country from the dropdown menu.",
"label": 0 },
    {"text": "Ensure the postal code matches your address.",
"label": 0 },
    {"text": "Update your notification preferences in your
account settings.", "label": 0 },
    {"text": "Generate a robust password using a combination
of letters, numbers, and symbols.", "label": 1 },
    {"text":
"SOME_SPECIAL_CHARACTERS_NOT_ALLOWED_IN_EMAIL:Make sure you
use your full email address, including an "@" sign and a
domain.", "label": 0 },
    {"text": "Ensure your password includes letters, numbers,
and common punctuation.", "label": 1 },
    {"text": "Your password should consist of a minimum of 8
characters and include two of the following: uppercase
letters, lowercase letters, numbers, or symbols.", "label": 1
},
    {"text": "Your password includes disallowed characters.",
"label": 1 },
    {"text": "Ensure your password is either 15 characters
long or at least 8 characters with a number and lowercase
letter.",
"label": 1 },
    {"text": "COMMON_PASSWORD: Can be guessed easily. Try
something else.", "label": 0 },
    {"text": "Utilize only letters, numbers, and common
punctuation in your password.", "label": 1 },
    {"text": "Please restrict your password to English
characters only.", "label": 1 },
    {"text": "Craft a strong password with a blend of
letters, digits, and symbols.", "label": 1 },
    {"text": "Use a password that comprises letters, numbers,
and standard punctuation marks.", "label": 1 },
    {"text": "INVALID_BIRTHDATE: That doesn't look right,
please re-enter your birthday.", "label": 0 },

```

```

    {"text": "Your password must be at least 8 characters in
length and contain at least two of the following: uppercase
letters, lowercase letters, digits, or symbols.",
    "label": 1 },
    {"text": "Your password contains prohibited characters.",
"label": 1 },
    {"text": "Ensure your password meets the criteria of
being 15 characters or 8 characters with a digit and a
lowercase letter.", "label": 1 },
    {"text": "Utilize solely letters, digits, and standard
punctuation.", "label": 1 },
    {"text": "Kindly use only English characters in your
password.", "label": 1 },
    {"text": "Develop a robust password incorporating
letters, numbers, and symbols.", "label": 1 },
    {"text": "BIRTHDATE_TOO_OLD: That doesn't look right,
please re-enter your birthday.", "label": 0 },
    {"text": "Make sure your password includes letters,
numbers, and common punctuation symbols.", "label": 1 },
    {"text": "Passwords must be a minimum of 8 characters and
include at least two of the following: uppercase letters,
lowercase letters, digits, or symbols.", "label": 1 },
    {"text": "Invalid characters found in your password.",
"label": 1 },
    {"text": "Ensure your password meets the requirements of
being either 15 characters or 8 characters with a number and
a lowercase letter.", "label": 1 },
    {"text": "Please restrict your password to letters,
numbers, and common punctuation.", "label": 1 },
    {"text": "ERROR_MISSING_FIELD_PHONE_ONLY:Your number is
needed for account security and won't be seen by others.",
"label": 0 },
    {"text": "Only English characters are allowed in your
password.", "label": 1 },
    {"text": "Formulate a strong password using letters,
numbers, and symbols.", "label": 1 },
    {"text": "BIRTHDATE_IN_FUTURE: That doesn't look right,
please re-enter your birthday.", "label": 0 },
    {"text": "Your password should incorporate letters,
numbers, and standard punctuation marks.", "label": 1 },
    {"text": "Passwords must contain a minimum of 8
characters and include at least two of the following:
uppercase letters, lowercase letters, numbers, or symbols.",
"label": 1 },
    {"text": "Your password includes unauthorized
characters.",
"label": 1 },

```

```

    {"text": "Ensure your password satisfies the condition of
being either 15 characters or 8 characters with a number and
a lowercase letter.", "label": 1 },
    {"text": "Use only letters, numbers, and standard
punctuation marks in your password.", "label": 1 },
    {"text": "UNDERAGE_USER: User is under age.", "label": 0
},
    {"text": "Please use exclusively English characters in
your password.", "label": 1 },
    {"text": "Create a secure password comprising letters,
numbers, and symbols.", "label": 1 },
    {"text": "Ensure your password consists of letters,
numbers, and common punctuation marks.", "label": 1 },
    {"text": "Passwords must be at least 8 characters long
and include at least two of the following: uppercase letters,
lowercase letters, digits, or symbols.", "label": 1 },
    {"text": "OVERAGE_USER: That doesn't look right, please
re-enter your birthday.", "label": 0 },
    {"text": "Your password contains restricted characters.",
"label": 1 },
    {"text": "Ensure your password meets the requirement of
either being 15 characters or 8 characters with a number and
a lowercase letter.", "label": 1 },
    {"text": "Use solely letters, numbers, and standard
punctuation in your password.", "label": 1 },
    {"text": "INVALID_NAME_LENGTH: That name is too long.",
"label": 0 },
    {"text": "Please ensure your password contains only
English characters.", "label": 1 },
    {"text": "Develop a strong password using a combination
of letters, numbers, and symbols.", "label": 1 },
    {"text": "LENGTH_TOO_SHORT: That email address is too
short, please use a longer one.", "label": 0 },
    {"text": "Ensure your password incorporates letters,
numbers, and common punctuation.", "label": 1 },
    {"text": "Your password must be a minimum of 8 characters
and include at least two of the following: uppercase letters,
lowercase letters, digits, or symbols.", "label": 1 },
    {"text": "Characters not allowed in your password.",
"label": 1 },
    {"text": "Ensure your password is either 15 characters or
8 characters with a number and a lowercase letter.", "label":
1 },
    {"text": "LENGTH_TOO_LONG: That email address is too
long, please use a shorter one.", "label": 0 },
    {"text": "Please use only letters, numbers, and common
punctuation in your password.", "label": 1 },

```

```

    {"text": "English characters are the only ones permitted
in your password.", "label": 1 },
    {"text": "ERROR_MISSING_FIELD:This is required.",
"label": 0 },
    {"text": "Design a robust password with letters, numbers,
and symbols.", "label": 1 },
    {"text": "Ensure your password includes letters, numbers,
and standard punctuation.", "label": 1 },
    {"text": "ERROR_MULTIPLE_AT_SIGN: We see you like the '@'
sign! Use only one in your email address, please.,"label": 0
},
    {"text": "Passwords must have a minimum of 8 characters
and include at least two of the following: uppercase letters,
lowercase letters, numbers, or symbols.", "label": 1 },
    {"text": "Prohibited characters detected in your
password.",
"label": 1 },
    {"text": "Make sure your password meets the criteria of
either being 15 characters or 8 characters with a number and
a lowercase letter.", "label": 1 },
    {"text": "Use only letters, numbers, and standard
punctuation symbols in your password.", "label": 1 },
    {"text": "ERROR_NO_DOMAIN: What about the stuff after the
'@'?.", "label": 0 },
    {"text": "Please restrict your password to English
characters exclusively.", "label": 1 },
    {"text": "Formulate a strong password with a mix of
letters, numbers, and symbols.", "label": 1 },
    {"text": "Your password should contain letters, numbers,
and common punctuation marks.", "label": 1 },
    {"text": "Passwords must be at least 8 characters long
and include at least two of the following: uppercase letters,
lowercase letters, digits, or symbols.", "label": 1 },
    {"text": "Unauthorized characters present in your
password.", "label": 1 },
    {"text": "Ensure your password meets the condition of
being either 15 characters or 8 characters with a number and
a lowercase letter.", "label": 1 },
    {"text": "ERR_SELECT_GOOGLE_ACCOUNT:Please select an
account.", "label": 0 },
    {"text": "Utilize solely letters, numbers, and standard
punctuation in your password.", "label": 1 },
    {"text": "Please use only English characters for your
password.", "label": 1 },
    {"text": "ELECTION_SPECIFIC_WORD_PRESENT: Not available,
try something else.", "label": 0 },

```

```

    {"text": "Create a secure password incorporating letters,
numbers, and symbols.", "label": 1 },
    {"text": "Ensure your password includes letters, numbers,
and standard punctuation marks.", "label": 1 },
    {"text": "ERROR_NOTFOUND:Oops, something went wrong.",
"label": 0 },
    {"text": "Passwords must have a minimum of 8 characters
and include at least two of the following: uppercase letters,
lowercase letters, numbers, or symbols.", "label": 1 },
    {"text": "Characters not permitted in your password.",
"label": 1 },
    {"text": "RISKY_PASSWORD : This password is not safe to
use. Attackers can easily find this password online and
attempt to access your account with it.", "label": 0 },
    {"text": "Ensure your password satisfies the requirement
of being either 15 characters or 8 characters with a number
and a lowercase letter.", "label": 1 },
    {"text": "Use only letters, numbers, and common
punctuation in your password.", "label": 1 },
    {"text": "EMAIL_DOMAIN_NOT_ALLOWED:This email isn't
available. Try a different one.", "label": 0 },
    {"text": "Please ensure your password contains solely
English characters.", "label": 1 },
    {"text": "Design a strong password with a blend of
letters, numbers, and symbols.", "label": 1 },
    {"text": "Ensure your password consists of letters,
numbers, and common punctuation.", "label": 1 },
    {"text": "EXCEEDED_ACCOUNTS_PER_PHONE_MASS_REG:Account
limit reached for this number. Please use a different one to
continue.", "label": 0 },
    {"text": "Passwords must be at least 8 characters and
include at least two of the following: uppercase letters,
lowercase letters, numbers, or symbols.", "label": 1 },
    {"text": "Unauthorized characters found in your
password.",
"label": 1 },
    {"text": "INVALID_BIRTHYEAR: That doesn't look right,
please re-enter your birth year.", "label": 0 },
    {"text": "Make sure your password meets the condition of
either being 15 characters or 8 characters with a number and
a lowercase letter.", "label": 1 },
    {"text": "Utilize only letters, numbers, and standard
punctuation marks in your password.", "label": 1 },
    {"text": "Please restrict your password to English
characters only.", "label": 1 },
    {"text": "Develop a robust password using a combination
of letters, numbers, and symbols.", "label": 1 },

```

```

    {"text": "EXCEEDED_ACCOUNTS_PER_PHONE:There are already
accounts associated with this phone number. Please use a
different one.", "label": 0 },
    {"text": "DOMAIN_NOT_SUPPORTED: This domain is not
supported.", "label": 0 },
    {"text": "Ensure your password incorporates letters,
numbers, and common punctuation.", "label": 1 },
    {"text": "Your password should be a minimum of 8
characters and include at least two of the following:
uppercase letters, lowercase letters, digits, or symbols.",
"label": 1 },
    {"text": "Prohibited characters detected in your
password.",
"label": 1 },
    {"text": "NAME_CONTAINS_URL: You can't use this name.",
"label": 0 },
    {"text": "Ensure your password meets the requirement of
either being 15 characters or 8 characters with a number and
a lowercase letter.", "label": 1 },
    {"text": "Use solely letters, numbers, and standard
punctuation symbols in your password.", "label": 1 },
    {"text": "Please use only English characters in your
password.", "label": 1 },
    {"text": "EXCEEDED_ACCOUNTS_PER_EMAIL:Your email is
listed on too many Yahoo accounts. Please use a different
email.",
"label": 0 },
    {"text": "Formulate a strong password with a mix of
letters, numbers, and symbols.", "label": 1 },
    {"text": "Your password should contain letters, numbers,
and common punctuation marks.", "label": 1 },
    {"text": "Passwords must be at least 8 characters long
and include at least two of the following: uppercase letters,
lowercase letters, digits, or symbols.", "label": 1 },
    {"text": "Characters not allowed in your password.",
"label": 1 },
    {"text": "EXCEEDED_ACCOUNTS_PER_CONTACT:Your mobile
number is listed on too many Yahoo accounts. Please use a
different number.", "label": 0 },
    {"text": "Ensure your password meets the condition of
being either 15 characters or 8 characters with a number and
a lowercase letter.", "label": 1 },
    {"text": "Utilize only letters, numbers, and common
punctuation in your password.", "label": 1 },
    {"text": "Please ensure your password contains solely
English characters.", "label": 1 },

```

```

    {"text": "Does not contain any similarities to your
login.",
"label": 0 },
    {"text": "Create a secure password incorporating letters,
numbers, and symbols.", "label": 1 },
    {"text": "Ensure your password consists of letters,
numbers, and standard punctuation marks.", "label": 1 },
    {"text": "You can only use letters, numbers.", "label": 1
},
    {"text": "IDENTIFIER_NOT_AVAILABLE:This email address is
not available for sign up, try something else.", "label": 0 },
    {"text": "You can only use letters, numbers, periods
(`.`), and underscores (`_`) in your username.", "label": 1
},
    {"text": "8 character min. with a number, symbol, upper,
lower case letter; characters cannot be repeated more than
twice in a row.", "label": 1 },
    {"text": "Does not contain character sequences (for
example, 11111, 12345, qwerty and so on).", "label": 1 },
    {"text": "root.regdata.messages = {IDENTIFIER_EXISTS: A
Yahoo account already exists with this email address.",
"label": 0 },
    {"text": "Is not a previously used password.", "label": 1
},
    {"text": "Is not a previously used password.", "label": 0
},
    {"text": "Does not contain any similarities to your
login.",
"label": 1 },
    {"text": "USER_DATA_CORRUPT:A Yahoo account already
exists with this email address.", "label": 0 },
    {"text": "Has a minimum length of 7 characters.",
"label": 1 },
    {"text": "Commonly known weak passwords are not accepted
(password123, etc.).", "label": 1 },
    {"text": "Your can define a minimum password length of 1
characters up to a maximum of 128 characters.", "label": 1 },
    {"text":
"REPLACE_SIGNIN_LINK.,DANGLING_IDENTIFIER_EXISTS:A Yahoo
account already exists with this email address.",
"label": 0 },
    {"text": "You can define password complexity to contain a
combination (up to a total of four) of the following: Digits
(0 to 9), Uppercase characters (A to Z), Lowercase characters
(a to z), Special characters (!@#$%^&*()), Unicode.",
"label": 1 },

```

```

        {"text": "Must contain at least 8 characters.,"label": 1
    },
    {"text": "Must contain uppercase, lowercase letters, and
numbers.,"label": 1 },
    {"text": "USER_DATA_CORRUPT:A Yahoo account already
exists with this email address.,"label": 0 },
    {"text": "INVALID_CHARACTERS_IN_PASSWORD: Can only
contain letters, numbers and , !","label": 1 },
    {"text": "If less than 12 characters, must contain a
special character.,"label": 1 }
]

```

A.2 Main.py

```

from flask import Flask, request, jsonify, render_template
import requests
from bs4 import BeautifulSoup
from nltk.tokenize import sent_tokenize
from flask_cors import CORS
import joblib
import random
import nltk
nltk.download('punkt')

# Load the saved model and vectorizer
log_reg = joblib.load("logistic_regression_model.pkl")
vectorizer = joblib.load("tfidf_vectorizer.pkl")

app = Flask(__name__)
CORS(app)
def get_search_results(user_input):
    url = f"https://www.google.com.tr/search?q={user_input}"
    headers = {
        "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64;
x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/91.0.4472.124 Safari/537.36"
    }
    # getting the number of search results from google
    response = requests.get(url, headers=headers)
    soup = BeautifulSoup(response.text, 'html.parser')

    result_stats = soup.find("div", {"id": "result-stats"})

    if result_stats:
        return result_stats.text
    else:
        return "No result stats found"

```

```

def random_replace(word, replacement_dict):
    modified_word = ""
    for letter in word:
        if letter in replacement_dict:
            replacement_options = replacement_dict[letter]
            random_value = random.choice(replacement_options)
            modified_word += random_value
        else:
            modified_word += letter
    return modified_word
# choose random letters and replace it

replacement_dict = {
    'ا': ['1', 'ا', '1'],
    'أ': ['2', 'أ', 'i'],
    'إ': ['!', 'إ'],
    'آ': ['1', 'آ', '1'],
    'ب': [',', 'ب', 'b'],
    'ت': [':', 'ت', 't'],
    'ث': [':', 'ث', 't'],
    'ج': ['>', 'ج'],
    'ح': ['7', 'ح'],
    'خ': ['`7', 'خ', '5'],
    'د': ['>', 'د'],
    'ذ': ['4', 'ذ', '>'],
    'ر': [')', 'ر'],
    'ز': ['`)', 'ز'],
    'س': ['?', 'س'],
    'ش': ['$ ', 'ش'],
    'ص': ['9', 'ص'],
    'ض': ['`9', 'ض'],
    'ط': ['6', 'ط'],
    'ظ': ['`6', 'ظ'],
    'ع': ['3', 'ع'],
    'غ': ['`3', 'غ'],
    'ف': [';', 'b.', 'ف'],
    'ق': ['8', 'ق'],
    'ك': ['L^', 'ك'],
    'ل': ['L', 'ل'],
    'م': ['o', 'م'],
    'ن': [',', 'ن', 'n'],
    'ه': ['@', 'ه'],
    'و': ['&', 'و', 'g'],
    'ؤ': ['^g', 'ؤ', '^', '^و'],
    'ي': [':', 'ي', 'y'],
    'ى': [' ', 'ى']
}

```

```

        'ئ': ['2', 'ئ'],
        'ء': ['^', 'ء'],
        'ة': [':@', 'ة']

    }

def random_engreplace(word, replace_eng_dict):
    modified_word = ""
    for letter in word:
        if letter in replace_eng_dict:
            replacement_options = replace_eng_dict[letter]
            random_value = random.choice(replacement_options)
            modified_word += random_value
        else:
            modified_word += letter
    return modified_word

# Example english usage
replace_eng_dict = {
    "ا": ["h"],
    "ا": ["H"],
    "ا": ["Y"],
    "ا": ["N"],
    "ب": ["f"],
    "ت": ["j"],
    "ث": ["e"],
    "ج": ["[",
    "ح": ["7", "p"],
    "خ": ["5", "`7", "o"],
    "د": ["]"],
    "ذ": ["4", "`"],
    "ر": ["v"],
    "ز": ["."],
    "س": ["s"],
    "ش": ["a"],
    "ص": ["9", "w"],
    "ض": ["`9", "q"],
    "ط": ["6", "'"],
    "ظ": ["`6", "/"],
    "ع": ["3", "u"],
    "غ": ["`3", "y"],
    "ف": ["t"],
    "ق": ["8", "r"],
    "ك": [";"],
    "ل": ["g"],
    "م": ["l"],
    "ن": ["k"],
    "ه": ["i"],

```

```

        "و": [",", "],
        "ؤ": ["2", "c"],
        "ي": ["d"],
        "ى": ["n"],
        "ئ": ["z"],
        "ء": ["x"],
        "ë": ["m"]
    }

def modify_word_with_eng(user_input, eng_replace_dict):
    modified_word = random_engreplace(user_input,
eng_replace_dict)
    return modified_word

def modify_word(user_input):
    while True:
        results = get_search_results('"' + user_input + '"')
        print(results)

        num_results = int(results.split()[1].replace('.', '
'))

        if num_results < 100:
            print(f"Use. Number of results is less than
100.")
            return user_input
        else:
            modified_word_ar = random_replace(user_input,
replacement_dict)
            modified_word_en =
modify_word_with_eng(user_input, replace_eng_dict)
            print(f"Modified word (Arabic):
{modified_word_ar}")
            print(f"Modified word (English):
{modified_word_en}")

            if modified_word_ar == user_input:
                print("No further modifications possible.
Exiting.")
                return user_input

            user_input = modified_word_ar # modified_word

# Function to extract JavaScript code from a URL
def extract_js_from_url(url):
    try:
        # Fetch the content of the webpage
        response = requests.get(url)

```

```

        response.raise_for_status()

        # Extract JavaScript code using BeautifulSoup
        soup = BeautifulSoup(response.content, 'html.parser')
    #or .text
        js_code = [script.text for script in
soup.find_all('script')]

        return js_code

    except requests.exceptions.HTTPError as http_err:
        print(f"HTTP error occurred: {http_err}")
    except Exception as err:
        print(f"An error occurred: {err}")

# Function to search for specific words in JavaScript code
and return full sentences
def search_specific_sentences(js_code):
    specific_words = "common", "lower", "lowercase", "lower-
case", "upper", "uppercase", "upper-case", "english", "only
use", "use only", "at least", "at most", "lower_case",
"upper_case", "letters", "symbols", "special characters",
"special-characters", "special_characters", "mixed case",
"mixed-case", "mixed_case", "must contain", "should contain",
"cannot contain", "should include", "must include", "cannot
include",
"password", "only contain"
]

    found_sentences = []

    for code in js_code:
        sentences = sent_tokenize(code) # Tokenize code into
sentences
        for sentence in sentences:
            for word in specific_words:
                if word in sentence:
                    found_sentences.append(sentence)

    for code in js_code:
        sentences = code.split(';') # Split code into
sentences
        for sentence in sentences:
            for word in specific_words:
                if word in sentence:
                    found_sentences.append(sentence.strip())

```

```

    return found_sentences

def classify_sentence(sentence, model, vectorizer,
threshold=0.5):
    vectorized_text = vectorizer.transform([sentence])
    probabilities = model.predict_proba(vectorized_text)[0]
    predicted_label = 1 if probabilities[1] > threshold else
0
    # Convert the label to a human-readable format
    label_text = "Password Requirement" if predicted_label ==
1 else "General Policy"
    return label_text, probabilities

# Function to find similar sentences and compute similarity
def find_similar_sentence(found_sentences):
    similar_sentences = []

    for website_sentence in found_sentences:
        label, probabilities =
classify_sentence(website_sentence, log_reg, vectorizer)
        confidence_label_1 = round(probabilities[1] * 100, 2)
        similar_sentences.append((website_sentence, label,
confidence_label_1))

    return similar_sentences

# Function to fetch content from a URL
def fetch_content(url):
    response = requests.get(url)
    return response.text

# Route for the home page
@app.route('/')
def home():
    return render_template('index.html')

@app.route('/submit', methods=['POST'])
def submit():
    data = request.get_json()

    if 'url' in data:
        # Case 1: Processing URL
        user_url = data.get('url')
        js_code = extract_js_from_url(user_url)

```

```

        found_sentences = []
        if js_code:
            found_sentences =
search_specific_sentences(js_code)
            if found_sentences:
                similar_sentences =
find_similar_sentence(found_sentences)
                if similar_sentences:
                    return jsonify({'result':
similar_sentences})
            else:
                print(f"\nFound the following sentences
in the JavaScript code:")
                for sentence in found_sentences:
                    print(sentence)
                return jsonify({'found_sentences':
found_sentences, 'message': "Not similar to the dataset."})
            else:
                return jsonify({'message': "No specified
words found in the JavaScript code."})
            else:
                return jsonify({'message': "No JavaScript code
found in the provided URL."})

    elif 'input' in data:
        # Case 2: Modifying Word
        user_input = data.get('input')
        modified_word_ar = modify_word(user_input) #
modified_word
        modified_word_en = modify_word_with_eng(user_input,
replace_eng_dict)
        return jsonify({'result_arabic': modified_word_ar,
'result_english': modified_word_en})

    else:
        return jsonify({'message': "Invalid request."})

if __name__ == '__main__':

    app.run(debug=True, host='0.0.0.0', port=5000)

```

A.3 Index.html

```

<!DOCTYPE html>
<html lang="en">

```

```

<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <title>Similar Sentences Finder</title>
</head>
<body>
  <h1>Similar Sentences Finder</h1>
  <p>Enter a URL to find similar sentences in the
JavaScript code:</p>
  <form action="/submit" method="post">
    <label for="url">URL:</label>
    <input type="text" id="url" name="url" required>
    <button type="submit">Submit</button>
  </form>
</body>
</html>

```

A.4 Results.html

```

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <title>Results</title>
</head>
<body>
  <h1>Results</h1>

  {% if similar_sentences %}
    <h2>Similar Sentences Found:</h2>
    <ul>
      {% for sentence_pair in similar_sentences %}
        <li>
          <p><strong>Website Sentence:</strong> {{
sentence_pair.0 }}</p>
          <p><strong>Classified as:</strong> {{
sentence_pair.1 }}</p>
          <p><strong>Prediction:</strong> {{
sentence_pair.2 }}%</p>
        </li>
      {% endfor %}
    </ul>

```

```

    {% else %}
        {% if found_sentences %}
            <h2>Found Sentences in the JavaScript code:</h2>
            <ul>
                {% for sentence in found_sentences %}
                    <li>{{ sentence }}</li>
                {% endfor %}
            </ul>
        {% else %}
            <p>{{ message }}</p>
        {% endif %}
    {% endif %}

</body>
</html>

```

A.5 popup.html

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Popup</title>
    <style>
        body {
            background-color: #dddddd; /* Set the background
color to gray */
        }
    </style>
</head>
<body>
    <h1>EASY PASS</h1>

    <!-- Input field and button for URL -->
    <label for="urlInput">Enter URL:</label>
    <input type="text" id="urlInput"
placeholder="https://example.com">
    <button id="urlSubmitButton">Submit</button>

    <!-- Display area for results -->
    <div id="resultsContainer" style="margin-top: 20px;
display: none;">
        <h2>Results:</h2>
        <ul id="resultsList"></ul>
    </div>
</body>
</html>

```

```

</div>

<!-- Input field and button for modifying word -->
<label for="userInput">Enter a word:</label>
<input type="text" id="userInput" placeholder="Enter a
word">
<button id="wordSubmitButton">Modify Word</button>

<!-- Display area for modified word -->
<div id="resultDisplay" style="margin-top: 20px; display:
none;"></div>

<!-- Message container for displaying errors or messages
-->
<div id="messageContainer" style="color: red; margin-top:
20px; display: none;"></div>

<script src="popup.js"></script>
</body>
</html>

```

A.6 Popup.js

```

document.addEventListener('DOMContentLoaded', function () {
    var urlSubmitButton =
document.getElementById('urlSubmitButton');
    var wordSubmitButton =
document.getElementById('wordSubmitButton');
    var urlInput = document.getElementById('urlInput');
    var userInput = document.getElementById('userInput');
    var resultsContainer =
document.getElementById('resultsContainer');
    var resultDisplay =
document.getElementById('resultDisplay');

    // Ensure messageContainer is defined before using it
    var messageContainer =
document.getElementById('messageContainer');
    if (!messageContainer) {
        // If messageContainer is not found, create it
        dynamically
        messageContainer = document.createElement('div');
        messageContainer.id = 'messageContainer';
        document.body.appendChild(messageContainer);
    }

    // Ensure resultsList is defined before using it

```

```

var resultsList = document.getElementById('resultsList');
if (!resultsList) {
    // If resultsList is not found, create it dynamically
    resultsList = document.createElement('ul');
    resultsList.id = 'resultsList';
    document.body.appendChild(resultsList);
}
// Function to display found sentences as an alert
function displayFoundSentences(foundSentences) {
    // Create a string containing all the found sentences
    var alertMessage = "Found Sentences in JS code:\n";
    foundSentences.forEach(function(sentence) {
        alertMessage += "- " + sentence + "\n";
    });

    // Display the alert message
    alert(alertMessage);
}

urlSubmitButton.addEventListener('click', function () {
    console.log('Button clicked');
    var url = urlInput.value.trim();

    if (url) {
        console.log('Sending request to Flask
server...');
        // Triggering a fetch request to the Flask server
        for URL
        fetch('http://localhost:5000/submit', {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
            },
            body: JSON.stringify({ url: url }),
        })
        .then(response => response.json())
        .then(data => {
            console.log('Data received from Flask
server:', data);

            // Handle the response data here for URL
            // ...
            // Example: Displaying results in the
            popup UI

            if (data.result) {
                console.log('Similar Sentences:');
                data.result.forEach(result => {
                    console.log('Website Sentence:',

```

```

result[0]);
result[1]);
result[2]);

down
// Display results logic here

// Calculate and print percentage
var percentage = result[2]; //
Convert similarity to percentage *100
var percentageStr =
percentage.toFixed(2) + "%"; // Format to two decimal places
console.log(percentageStr + " you
cannot use Arabic letters");

// Display results in the popup
UI
var listItem =
document.createElement('li');
listItem.innerHTML =
`<strong>Website Sentence:</strong> ${result[0]}<br>
<strong>Classified as:</strong> ${result[1]}<br>
<strong>Prediction:</strong> ${result[2]}<br>
(${percentageStr} you cannot use Arabic letters)`;
resultsList.appendChild(listItem);
});

// Show the results container and
hide the message container
resultsContainer.style.display =
'block';
messageContainer.style.display =
'none';

} else if (data.message) {
console.log('Error:', data.message);
// Display error message in the
message container
messageContainer.textContent =
data.message;
messageContainer.style.display =
'block';
resultsContainer.style.display =
'none';
}

```

```

        // Display found sentences if available
        if (data.found_sentences) {
            console.log('Found Sentences:',
data.found_sentences);
        }
        displayFoundSentences(data.found_sentences);
    }
    })
    .catch(error => {
        console.error('Error:', error);
    });
} else {
    console.log('Please enter a URL. ');
    // Display a message in the message container
    messageContainer.textContent = 'Please enter a
URL. ';
    messageContainer.style.display = 'block';
    resultsContainer.style.display = 'none';
}
});

wordSubmitButton.addEventListener('click', function () {
    var input = userInput.value.trim();

    if (input) {
        // Triggering a fetch request to the Flask server
for word
        fetch('http://localhost:5000/submit', {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
            },
            body: JSON.stringify({ input: input }),
        })
        .then(response => response.json())
        .then(data => {
            // Handle the response data here for word
            // ...

            // Example: Displaying modified word in
the popup UI
            if (data.result_arabic &&
data.result_english) {
                // Display modified word logic here
                resultDisplay.innerHTML =
`<strong>Modified Word (Arabic):</strong>

```

```

    ${data.result_arabic}<br> <strong>Modified Word
    (English):</strong> ${data.result_english}`;
        resultDisplay.style.display =
'block';
        resultsContainer.style.display =
'none';
        messageContainer.style.display =
'none';
        } else if (data.message) {
            // Display error message in the
message container
            messageContainer.textContent =
data.message;
            messageContainer.style.display =
'block';
        }
    })
    .catch(error => {
        console.error('Error:', error);
    });
    } else {
        console.log('Please enter a word.');
```

word. ';

```

        // Display a message in the message container
        messageContainer.textContent = 'Please enter a
        messageContainer.style.display = 'block';
    }
    });
});

```