Multi-Purpose AI Password Analysis Tool

Osama Khalid – 27971
Sardar M. Zeeshan khan – 27969
Salman Ali – 27667

Supervised by:
Mr. Osamah Ahmed
Dr. Mansoor Alam

Faculty of Computing
Riphah International University, Islamabad
Spring 2024

A Dissertation Submitted To

Faculty of Computing,

Riphah International University, Islamabad

As a Partial Fulfillment of the Requirement for the Award of

the Degree of

Bachelors of Science in Cyber Security

Faculty of Computing
Riphah International University, Islamabad

Date: 23/6/2024

Final Approval

This is to certify that we have read the report submitted by ***Osama Khalid 27971, Sardar Muhammad Zeeshan Khan 27969, Salman Ali 27667***, for the partial fulfillment of the requirements for the degree of the Bachelors of Science in Cyber Security (BSCY). It is our judgment that this report is of sufficient standard to warrant its acceptance by Riphah International University, Islamabad for the degree of Bachelors of Science in Cyber Security (BSCY).

Committee:

1 ________________________________

Mr. Osama Ahmad & Dr. Mansoor Alam
(Supervisors)

2 ________________________________

Dr. Musharraf Ahmed
(Head of Department/chairman)

Declaration

We hereby declare that this document **"Multi-Purpose AI Password Analysis Tool"** neither as a whole nor as a part has been copied out from any source. It is further declared that we have done this project with the accompanying report entirely on the basis of our personal efforts, under the proficient guidance of our teachers, especially our supervisors **Mr. Osamah Ahmad** and **Dr. Mansoor Alam.** If any part of the system is proved to be copied out from any source or found to be reproduction of any project from anywhere else, we shall stand by the consequences.

s

Osama Khalid
27971

Sardar M. Zeeshan khan

27969

Salman Ali

27667

Dedication

This project is dedicated to our friends, mentors, and educators, whose unwavering support and encouragement have been a constant source of strength throughout this journey. Their belief in our capabilities has fueled our determination to succeed. Additionally, we dedicate this work to our colleagues and the academic community, whose guidance and knowledge have been instrumental in shaping our professional growth. Their commitment to excellence has inspired us to push the boundaries of our potential. Thank you for being our pillars of support and for always believing in us.

Acknowledgement

First of all, we are obliged to Allah the Almighty the Merciful, the Beneficent and the source of all Knowledge, for granting us the courage and knowledge to complete this Project.

We would like to express our sincere gratitude to the following individuals whose guidance and support were instrumental in the successful completion of this project:

- **Project Supervisor:** We are particularly grateful to our project supervisors, **Mr. Osama Ahmad & Dr. Mansoor Alam**, for their invaluable expertise, insightful feedback, and unwavering encouragement throughout the entire project development process. Their dedication and support played a pivotal role in shaping the direction and success of our work.
- **Faculty Advisors:** We extend our sincere thanks to our faculty advisors, **Dr. Muhammad Mansoor Alam**, whose passion for AI inspired us to pursue this project, and **Dr. Jawaid Iqbal**, whose knowledge and guidance were invaluable throughout the research phase.
- **Technical Support Staff:** We appreciate the assistance provided by **Mr. Haseeb Ahmed**, **Mr. Ihtesham Ullah**, and **Mr. Tajammul Hussain**. Their technical expertise and willingness to help were crucial in overcoming technical challenges.
- **Project Convener:** We acknowledge the leadership of **Mr. Muhammad Ahmad Nawaz** as the project convener, whose guidance ensured the project's smooth execution within the academic framework.
- **Colleagues and Friends:** Finally, we would like to thank our colleagues, **Mr. Osama Raza** and **Mr. Awais Nawaz**, for their encouragement, collaboration, and support throughout the course of this project. Their positive spirit and willingness to assist contributed significantly to our progress.

Osama Khalid
27971

Sardar M. Zeeshan khan
27969

Salman Ali
27667

Abstract

In an increasingly digitized world, the security of personal and organizational data is paramount. This project presents a comprehensive solution in the form of a multi-purpose AI-driven password analysis tool. By harnessing the power of artificial intelligence, this tool offers dual functionality, serving both offensive and defensive purposes.

On the offensive front, it employs advanced algorithms to analyze and crack passwords, providing insights into their vulnerabilities and potential points of exploitation. This capability enables security professionals to understand the weaknesses inherent in various password configurations, thereby facilitating the development of more robust defense strategies.

Simultaneously, the tool acts as a guardian on the defensive front, evaluating the strength of passwords and identifying potential breaches through sophisticated pattern recognition and analysis. By proactively identifying compromised credentials, it empowers users to take preemptive action, mitigating the risks associated with data breaches and unauthorized access.

Through its intuitive interface and customizable features, this AI-powered tool becomes an indispensable asset for individuals and organizations seeking to fortify their digital security posture. By enabling users to test the resilience of their passwords and stay ahead of emerging threats, it serves as a proactive safeguard in an ever-evolving cybersecurity landscape.

TABLE OF CONTENTS

System Design Figures

Table of Figures

Chapter 1: Introduction

Chapter 1: Introduction

1.1 Introduction

1.2 Opportunities and Stakeholders

1.3 Motivations and Challenges

1.4 Goals and Objectives

1.5 Solution Overview

1.6 Report Outline

Chapter 1: Introduction

Project Title: Multi-Purpose AI Password Analysis Tool

1.1 Introduction

In today's world, where everything is becoming digital and online threats are on the rise, having strong password security is crucial. That's where the AI Multipurpose Password Analysis Tool comes in. It's a game-changing software that's here to revolutionize how we manage passwords. By combining cutting-edge artificial intelligence with traditional password analysis tools, this tool offers a complete solution for keeping our digital accounts safe. Whether it's cracking passwords or assessing password's strength, this tool covers multiple aspects of password security. And it doesn't stop there - it seamlessly works with popular tools like Hashcat, making it even more powerful. So, as we step into the future of password management, the AI Multipurpose Password Analysis Tool is leading the way, providing both analysis and cracking abilities to protect our sensitive information from the ever-evolving threats online.

1.2 Opportunity & Stakeholders

The key opportunities and primary stakeholders associated with this project are outlined as follows:

Opportunities

- **Enhanced Security Assessments:** Businesses and individuals can leverage the tool for penetration testing and vulnerability assessments, identifying potential weaknesses in their password security policies and practices.
- **AI-driven Offensive and Defensive Capabilities:** Utilizing AI for both offensive (password cracking) and defensive (password strength assessment) purposes provides a comprehensive approach to password security management.

Stakeholders

- **Businesses:** Companies aim to improve their overall cybersecurity posture by identifying and mitigating password-related vulnerabilities.
- **Security Professionals:** Penetration testers and security consultants who can use the tool for vulnerability assessments and ethical hacking activities.
- **Law Enforcement:** Agencies might leverage the tool for lawful investigations adhering to court orders and legal guidelines.

1.3 Problem Statement

With increasing password complexity requirements, brute-force attacks become significantly slower and less efficient.

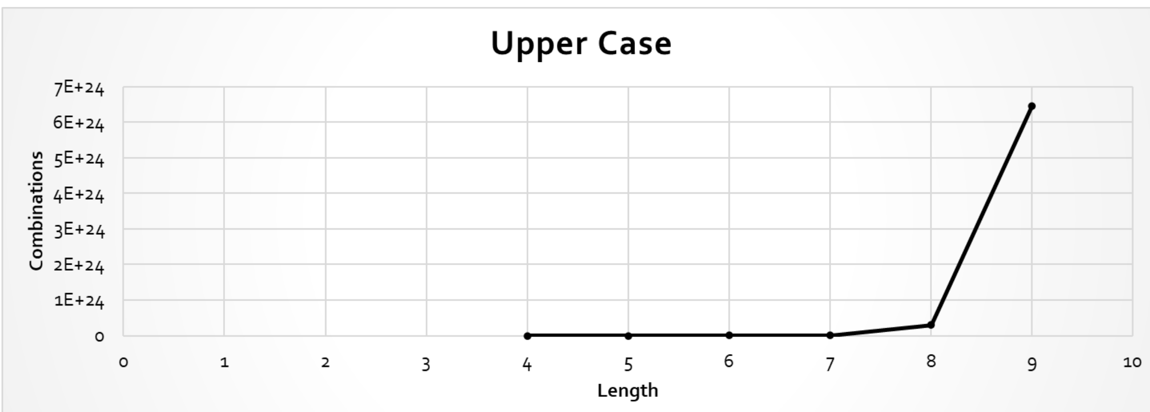

Figure 1: Upper Case

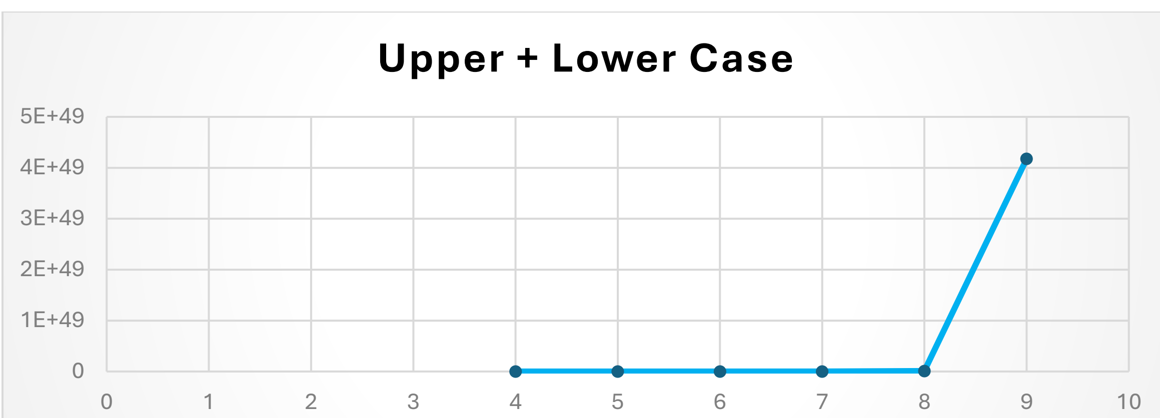

Figure 2 :Upper and Lowercase

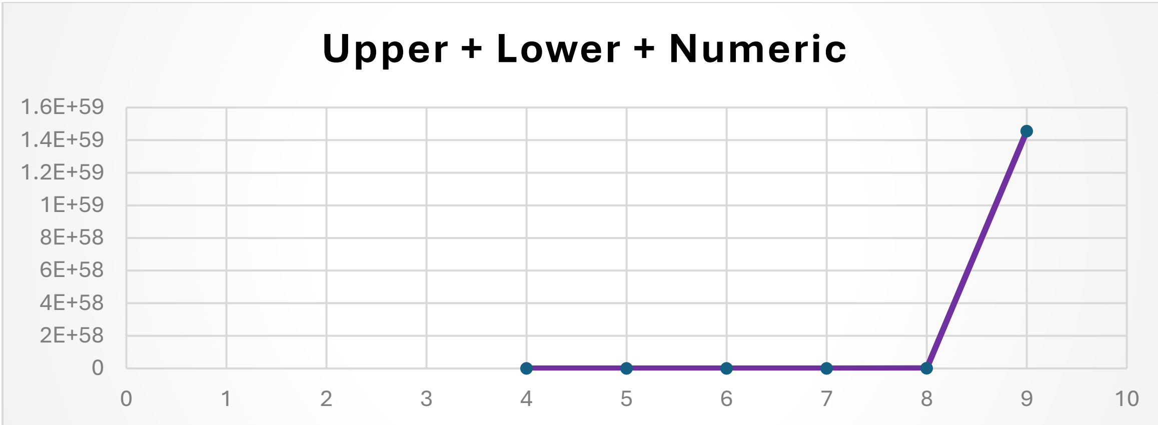

Figure 3:Upper, Lower and Numeric

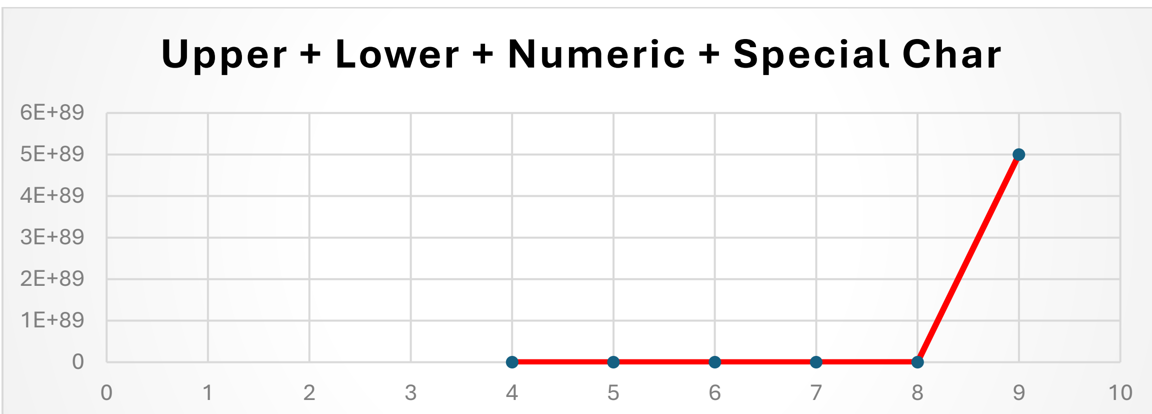

Figure 4:Upper, Lower, Numeric, and Special Char

Users often employ longer and more complex passwords (e.g., 8+ characters, combination of uppercase, lowercase, symbols) making traditional brute-force methods impractical.

Length	**Combination**
4^{26+26}	2220446049250313080847263336181640625
$4^{26+26+10}$	21267647932558653966460912964485513216
5^{26+26}	29098125988731506183153025616435306561536
$5^{26+26+10}$	21684043449710088680149056017398834228515625

Table 1:**No. of iterations for each length of passwords**

Length	Upper Case	Upper + Lower Case	Upper + Lower + Numeric	Upper + Lower + Numeric + Special Char
4	4.5036E+15	2.02824E+31	2.12676E+37	3.92319E+56
5	1.49012E+18	2.22045E+36	2.1684E+43	5.04871E+65
6	1.70582E+20	2.90981E+40	1.75945E+48	1.40029E+73
7	9.38748E+21	8.81248E+43	2.48931E+52	2.74926E+79
8	3.02231E+23	9.13439E+46	9.80797E+55	7.77068E+84
9	6.46108E+24	4.17456E+49	1.45558E+59	4.998E+89

Table 2:**Password Combinations**

1.4 Motivation and Challenges

Outlined below are the motivations and challenges associated with our AI-driven password analysis tool.

Motivation

In the digital age, robust password security is more critical than ever. The need for an advanced, AI-driven password analysis tool arises from several key motivations.

Increasing Complexity of Passwords:

- As cyber threats evolve, users are encouraged to create more complex passwords to enhance security. However, the complexity often leads to users adopting predictable patterns, which can be exploited.
- Our tool aims to address this by leveraging AI to recognize and exploit these patterns, thereby improving both offensive and defensive security measures.

Advancements in AI and Machine Learning:

- The development of sophisticated AI algorithms offers new possibilities for password analysis and cracking. By integrating AI, our tool can stay ahead of traditional methods, providing more efficient and effective solutions.
- Utilizing state-of-the-art AI techniques ensures that our tool can handle the increasing complexity and diversity of modern passwords.

Comprehensive Security Assessments:

- Businesses and security professionals need comprehensive tools that can provide both offensive and defensive capabilities. By integrating password strength analysis with AI-driven password cracking, our tool offers a holistic approach to password security.
- This dual functionality supports thorough security assessments, identifying vulnerabilities and testing the robustness of password policies.

Growing Number of Data Breaches:

- The frequency and scale of data breaches are increasing, exposing millions of passwords. Our tool can analyze these breaches to understand common patterns and weaknesses, enhancing its cracking capabilities and providing insights into better password practices.
- By incorporating breach-checking features, our tool can alert users if their passwords have been compromised, enabling proactive security measures.

Enhancing Cybersecurity Awareness:

- Educating users about the importance of strong passwords and the risks of weak ones is crucial. Our tool can demonstrate the ease with which weak passwords can be cracked, promoting better password practices.
- Security professionals can use our tool to raise awareness and provide tangible evidence of the need for robust password policies.

Challenges

Data Collection and Quality:

- Obtaining a diverse and representative dataset of passwords is crucial for training the AI model. However, ethical and legal considerations must be adhered to when sourcing data from online leaks and breaches.

- Ensuring the dataset is comprehensive and free from sensitive or personally identifiable information requires rigorous data cleaning and preprocessing.

Balancing Offensive and Defensive Capabilities:

- Integrating both offensive (password cracking) and defensive (password strength analysis) functionalities in a single tool presents a significant challenge. The tool must be designed to operate ethically, with proper authorization and adherence to legal guidelines.
- Ensuring that the tool's offensive capabilities do not compromise its defensive functionalities, and vice versa, requires careful design and implementation.

Adapting to Evolving Password Practices:

- Password patterns and user behaviors are continuously evolving. Traditional methods often struggle to keep up with these changes, and our AI-driven approach must be flexible and adaptive to remain effective.
- Continuous updates and retraining of the AI model are necessary to handle new password trends and behaviors.

Efficiency and Scalability

- As passwords become more complex, traditional brute-force methods become impractical due to their time and resource-intensive nature. Our AI-powered tool must significantly reduce the search space and improve efficiency.
- Ensuring that the tool can scale effectively to handle large datasets and complex passwords without compromising performance is a critical challenge.

Security and Privacy Concerns:

- Implementing features such as breach checking involves integrating with online databases, which must be done securely to prevent exposure to sensitive information.

- Ensuring that the tool itself is secure and does not become a vector for attacks is paramount. This includes safeguarding against misuse by malicious actors.

1.5 Goals and Objectives

Goals

- Enhance Password Security
- Efficient Password Cracking
- Advanced Password Strength Analysis
- Comprehensive Vulnerability Assessments

Objectives

- Focus cracking efforts on statistically probable password patterns.
- Analyze password strength based on complexity metrics and AI-driven pattern recognition.
- Reduced cracking time.
- Improved vulnerability assessment.

1.6 Solution Overview

Our proposed AI-powered tool enhances password security by leveraging advanced algorithms and human-like password data. It focuses on both offensive and defensive capabilities:

Focused Cracking Efforts: Utilizes AI to identify statistically probable password patterns, reducing the search space and accelerating the cracking process compared to traditional brute-force methods.

Password Strength Analysis: Employs AI-driven pattern recognition and complexity metrics to analyze and identify weaknesses in complex passwords, aiding in targeted cracking attempts.

Reduced Cracking Time: Enhances efficiency by concentrating on likely password patterns, significantly reducing the time required for cracking.

Improved Vulnerability Assessment: Provides comprehensive assessments of password vulnerabilities, helping to identify and address weak points proactively.

Scope of the Project

Defensive capabilities:

Password strength assessment: Analyze password complexity, identify dictionary words, and check for patterns using AI.

<u>**Optional**</u>

- **Breach checking:** Integrate with online databases to verify if entered passwords and email addresses have been exposed in known data breaches.

Offensive capabilities: (For Ethical purposes only)

AI-powered password cracking: Utilize correlation of password data and advanced algorithms to efficiently crack passwords within a specified scope and with proper authorization.

1.7 Report Outline

The project report for the "Multi-Purpose AI Password Analysis Tool" begins with an introduction, providing a comprehensive background on password security and the impetus for developing this tool. It identifies key opportunities and stakeholders who will benefit from enhanced password analysis. The report then reviews existing password analysis tools such as John the Ripper, Brutus, and Wfuzz, highlighting their limitations and setting the context for the problem statement and proposed solution. Our proposed solution leverages AI to focus on statistically probable password patterns and complexity metrics, aiming to reduce cracking time and improve vulnerability assessment. The objectives and scope of the project are clearly defined, emphasizing defensive capabilities like password strength assessment and breach checking, as well as ethical offensive capabilities like AI-powered password cracking. The evaluation plan outlines a comprehensive assessment using diverse datasets from real-world breaches, password dictionaries, and synthetic data. The development and training section details the process of acquiring and preparing relevant datasets, training and optimizing the AI model, and selecting appropriate algorithms. Implementation focuses on core functionalities for password strength analysis and breach checking, with optional automated password cracking. The report also discusses the design and development of a user-friendly interface, ensuring seamless integration with the AI model and functionalities. Thorough testing and refinement processes are documented to ensure the tool's effectiveness and usability. Finally, the report includes references and a list of figures to support the content and provide visual clarity.

Chapter 2:
Market Review

Chapter 2: Market Review

Chapter 2: Market Review

2.1 Background

Traditional password cracking tools have relied on brute force techniques or precomputed dictionaries to decipher passwords. While these methods have been effective to some extent, they suffer from several limitations and challenges.

1. **Limited Efficiency**: Brute force attacks iterate through all possible combinations of characters, making them time-consuming and resource-intensive, especially for complex passwords.
2. **Dependence on Dictionaries**: Dictionary attacks rely on precompiled lists of commonly used passwords or words found in dictionaries. However, these lists may not encompass all possible variations, especially when users employ complex or unique passwords.
3. **Lack of Adaptability**: Traditional tools often struggle with adaptive password generation techniques, such as adding special characters or changing letter cases, making them less effective against modern password practices.
4. **Inability to Detect Breached Credentials**: Many password cracking tools do not have built-in mechanisms to cross-reference passwords with known breaches, leaving users unaware if their passwords have already been compromised.
5. **Scalability Issues**: As passwords become longer and more complex to enhance security, traditional methods face scalability challenges in terms of processing power and memory requirements.

2.2 Market Review / Technologies Overview

The foundation of password security lies in its ability to resist unauthorized access, primarily achieved through encryption and hashing techniques. Over the years, several tools and methodologies have been developed to analyze and crack passwords, each with its unique strengths and weaknesses.

Existing Systems

The following are the Existing systems that exist in the market for about more than a decade.

John the Ripper

Limitations:

1. Relies on traditional methods of password cracking such as dictionary attacks, brute-force attacks, and rainbow tables.
2. While powerful, it may not effectively adapt to evolving password patterns and behaviors without continuous updates and modifications.

Problems:

1. Lack of advanced AI integration for targeted password list generation based on learned patterns.
2. Limited defensive capabilities in analyzing password strength beyond basic dictionary checks.

Brutus

Limitations:

1. Primarily designed for online password cracking through network protocols like HTTP, FTP, SMB, etc.
2. May lack advanced AI-driven capabilities for generating targeted password lists.

Problems:

1. Limited applicability for offline password analysis and cracking scenarios.
2. May not integrate well with the defensive aspects of the proposed tool, such as analyzing password strength and checking for breached credentials.

Wfuzz

Limitations:

1. Primarily focuses on web application security testing, including fuzzing and brute-forcing directories and files.
2. May lack specialized features for password cracking and analysis.

Problems:

1. Not specifically tailored for password analysis and cracking tasks, thus requiring significant adaptation to fit into the proposed project's scope.
2. Limited or no integration with AI-driven password analysis and cracking techniques.

Comparison Tables

Tool	John the Ripper	RainbowCrack	OphCrack
Type	Password Cracker	Password Cracker	Password Cracker
Supported Platforms	Windows, Linux, macOS	Windows, Linux	Windows, Linux
Password Hashes Supported	Various (Unix, Windows, etc.)	LM, NTLM, MD5, SHA1, SHA256, SHA512	LM, NTLM
Attack Methods	Dictionary, Brute Force, Hybrid	Precomputed Hash Tables	Rainbow Tables, Brute Force
Speed	Fast	Depends on Rainbow Table size	Moderate
User Interface	Command Line	Command Line	GUI
License	Open Source	Freeware	Open Source
Usage	Penetration Testing, Password Auditing	Password Cracking	Password Recovery

Table 3:Comparsion Tables of tools

Tool	L0phtCrack	Aircrack-ng
Type	Password Cracker	Wi-Fi Network Security Tool
Supported Platforms	Windows	Linux, macOS
Password Hashes Supported	LM, NTLM	WEP, WPA, WPA2
Attack Methods	Dictionary, Brute Force	Dictionary, Brute Force, WPS PIN
Speed	Fast	Depends on hardware and complexity of the password
User Interface	GUI	Command Line
License	Commercial	Open Source
Usage	Password Cracking	Wi-Fi Network Security Testi

Table 4:Comparison Table

2.3 Summary

The evolution of password security technologies reflects the ongoing battle between cybersecurity professionals and cybercriminals. Traditional tools like John the Ripper, Brutus, and Wfuzz continue to play crucial roles in password cracking and security testing. However, the advent of AI and machine learning has ushered in a new era of password security, offering more efficient and effective methods for analyzing and cracking passwords.

The integration of human-like passwords data with advanced algorithms represents a significant leap forward, enabling more targeted and successful password cracking attempts. This literature review underscores the need for continuous innovation in password security, highlighting the potential of AI-powered tools to enhance our defenses against evolving cyber threats.

By understanding the strengths and limitations of existing tools and technologies, this review sets the stage for the development of a comprehensive AI-powered password analysis tool. Such a tool aims to address the current gaps in password security, providing a more robust and reliable means of protecting sensitive data in an increasingly digital world.

Chapter 3:
System Requirements

Chapter 4: System Requirements

3.1 Hardware Requirements
3.2 Software Requirements

Chapter 3: System Requirements

3.1 Hardware Requirements

3.1.1 Workstation: HP Z840

The HP Z840 workstation is a high-performance desktop designed for demanding tasks such as AI and machine learning model training. It offers robust processing power, extensive memory capacity, and advanced graphics capabilities, making it ideal for computationally intensive applications.

3.1.2 Processor: Intel Xeon E5-2680 V4 Dual

The Intel Xeon E5-2680 V4 is a high-end server processor with 14 cores and 28 threads, offering a base clock speed of 2.4 GHz and a turbo boost up to 3.3 GHz. Utilizing two of these processors in a dual configuration provides a total of 28 cores and 56 threads, significantly enhancing parallel processing capabilities crucial for training complex AI models.

3.1.3 RAM: 64GB DDR4 ECC

64GB of DDR4 Error-Correcting Code (ECC) RAM ensures high data integrity and system stability, which is essential for handling large datasets and preventing data corruption during intensive computations. This amount of memory supports the simultaneous operation of multiple AI models and applications without performance degradation.

3.1.4 GPU: RTX 3070TI 8GB GDDR6x

The NVIDIA RTX 3070TI graphics card, with 8GB of GDDR6x memory, provides powerful parallel processing capabilities and accelerated performance for deep learning tasks. Its CUDA cores and Tensor cores are optimized for AI workloads, significantly speeding up the training and inference processes of neural networks.

3.1.5 SSD: 512GB

A 512GB Solid-State Drive (SSD) offers fast read/write speeds and quick access to data, reducing loading times and improving overall system responsiveness. The SSD is used for storing the operating system, software applications, and project files, ensuring efficient data retrieval and storage.

3.2 Software Requirements

3.2.1 Operating System: Windows 11 64bit

Windows 11 is the latest version of Microsoft's operating system, providing a modern interface, enhanced security features, and support for the latest hardware and software technologies. The 64-bit version is necessary to leverage the full potential of the workstation's hardware, especially the large amount of RAM and dual processors.

3.2.2 Integrated Development Environment (IDE): Visual Studio

Visual Studio is a comprehensive development environment from Microsoft, offering tools and features for coding, debugging, and deploying applications. It supports multiple programming languages and integrates well with various frameworks and libraries, making it suitable for developing and testing the AI password analysis tool.

3.2.3 Programming Language: Python 3.12

Python 3.12 is the latest stable release of the Python programming language, known for its simplicity, readability, and extensive library support. It is widely used in AI and machine learning projects due to its powerful frameworks and community support. Python serves as the primary language for developing the AI models and integrating different components of the tool.

3.2.4 Machine Learning Framework: TensorFlow

TensorFlow is an open-source machine learning framework developed by Google, offering comprehensive tools for building and deploying machine learning models. It supports deep learning and neural network training, providing high performance on both CPUs and GPUs. TensorFlow's flexibility and scalability make it a preferred choice for developing complex AI applications.

3.2.5 Deep Learning Library: PyTorch

PyTorch is an open-source deep learning library developed by Facebook's AI Research lab, known for its dynamic computational graph and ease of use. It is widely adopted for research and production in AI, offering strong support for GPU acceleration and integration with other machine learning tools. PyTorch is used for experimenting with and deploying AI models in the password analysis tool.

3.2.6 GPU-Accelerated Library: cuDNN

cuDNN (CUDA Deep Neural Network library) is a GPU-accelerated library developed by NVIDIA, designed to enhance the performance of deep learning frameworks. It provides highly optimized implementations of standard routines such as forward and backward convolution, pooling, normalization, and activation layers. cuDNN is essential for maximizing the efficiency of TensorFlow and PyTorch models on the RTX 3070TI GPU.

Chapter 4:
Preparing Datasets

Chapter 4: Preparing Datasets

4.1 Introduction

4.2 Acquire or prepare relevant password datasets for training

Chapter 5: Preparing Datasets

4.1 Introduction

In the development of any machine learning or artificial intelligence project, the quality and preparation of datasets play a pivotal role in determining the success and efficacy of the final model. This chapter delves into the essential processes and considerations involved in acquiring, preparing, and managing datasets specifically tailored for our multipurpose AI password analysis tool. The robustness of our AI models, such as Generative Adversarial Networks (GANs), Recurrent Neural Networks (RNNs), and Transformers, hinges on the richness and accuracy of the data they are trained on.

4.2 Acquire or prepare relevant password datasets for training

Getting a dataset ready for password generation involves a series of steps aimed at making sure the data accurately represents different password characteristics and is suitable for training. Here's a simplified breakdown of what we need to do:

4.2.1 Data Collection

Get a mix of passwords from various sources, like:

- Online leaks and breaches (follow ethical and legal guidelines).
- Making sure our dataset covers a wide range of password types, including different lengths, complexities, and compositions.

4.2.1.1 Collected Datasets Table

No	Name	Size
1	rockyou2021.txt dictionary from kys234 on RaidForums	12.7 GB
2	36.4GB-18_in_1.lst_2.7z	4.50 GB
3	ASLM.txt.7z	127 MB
4	b0n3z_dictionary-SPLIT-BY-LENGTH-34.6GB_2.7z	3.29 GB
5	b0n3z-wordlist-sorted_REPACK-69.3GB_3.7z	9.07 GB
6	bad-passwords-master.zip	1.34 MB
7	crackstation.txt.gz	4.19 GB
8	dictionaries-master.zip	19.3 MB
9	Password lists.zip	336 MB
10	password-list-main.zip	291 MB
11	password-lists-master.zip	8.86 MB
12	pastePasswordLists-main_2.zip	54.6 MB
13	PowerSniper-master.zip	0.3 MB
14	pwlist-master.zip	8.02 MB
15	rockyou.zip	41.7 MB
16	SecLists-master.zip	554 MB
17	statistically-likely-usernames-master.7z	9.07 MB
18	vietnam-password-lists-master.zip	5.14 MB
19	wpa-passwords-master.zip	5.79 MB
20	WPA-PSK WORDLIST 3 Final (13 GB).rar	4.49 GB
21	cyclone.hashesorg.hashkiller.combined.7z	6.58 GB

Table 5:Datasets Table

4.2.1.2 Dataset Downloaded Screen Shot

Name	Date modified	Type	Size
rockyou2021.txt dictionary from kys234 on RaidForums	7/15/2022 4:56 AM	File folder	
36.4GB-18_in_1.lst_2.7z	3/10/2024 11:28 AM	7z Archive	4,726,867 KB
ASLM.txt.7z	12/11/2021 12:16 PM	7z Archive	130,261 KB
b0n3z_dictionary-SPLIT-BY-LENGTH-34.6GB_2.7z	3/10/2024 12:47 PM	7z Archive	3,460,722 KB
b0n3z-wordlist-sorted_REPACK-69.3GB_3.7z	3/10/2024 12:26 PM	7z Archive	9,511,223 KB
bad-passwords-master.zip	3/10/2024 10:02 AM	zip Archive	1,376 KB
crackstation.txt.gz	3/10/2024 4:57 PM	gz Archive	4,395,271 KB
dictionaries-master.zip	3/10/2024 10:02 AM	zip Archive	19,766 KB
Password lists.zip	5/25/2024 1:21 AM	zip Archive	344,979 KB
password-list-main.zip	3/10/2024 10:26 AM	zip Archive	298,841 KB
password-lists-master.zip	3/10/2024 9:59 AM	zip Archive	8,894 KB
pastePasswordLists-main_2.zip	3/10/2024 10:26 AM	zip Archive	55,928 KB
PowerSniper-master.zip	3/10/2024 10:02 AM	zip Archive	372 KB
pwlist-master.zip	3/10/2024 10:02 AM	zip Archive	8,220 KB
rockyou.zip	5/25/2024 1:24 AM	zip Archive	42,728 KB
SecLists-master.zip	3/10/2024 10:50 AM	zip Archive	567,740 KB
statistically-likely-usernames-master.7z	5/25/2024 1:14 AM	7z Archive	9,298 KB
vietnam-password-lists-master.zip	3/10/2024 10:00 AM	zip Archive	5,269 KB
wpa-passwords-master.zip	3/10/2024 10:02 AM	zip Archive	5,937 KB
WPA-PSK WORDLIST 3 Final (13 GB).rar	3/10/2024 10:56 AM	rar Archive	4,710,749 KB
cyclone.hashesorg.hashkiller.combined.7z	5/25/2024 1:42 AM	7z Archive	6,902,263 KB

Figure 5:**Datasets Screenshots**

4.2.1.3 Data Cleaning

Filter the password of length of 4 to 10 to keep the datasets clean and manageable. Get rid of any weird or corrupted entries that could mess up training.

4.2.1.3.4 Pseudocode: Data Filtration

pseudocode Data Filteration

pseudocode Data Filteration

```
START Main Script
    Parse command-line arguments
    IF word lengths not provided
        Set default word lengths to [5, 6, 7, 8]
    CALL parallel_filter_words with parsed arguments
END Main Script

FUNCTION parallel_filter_words(directory, word_lengths, num_chunks, subchunk_size, encoding, output_directory)
    Create multiprocessing pool
    IF output directory does not exist
        Create output directory

    Open log file in output directory

    FOR each file in directory
        IF file does not end with '.txt'
            Continue to next file

        Get file path
        Get file chunks based on num_chunks

        FOR each word length in word_lengths
            Create directory for current word length if not exists
            Create output filename for filtered words

            Initialize list to store results from multiprocessing

            FOR each chunk (chunk_start, chunk_size) in file chunks
```

Figure 6:Pseudocode DataFilteration-1

```
            FOR each chunk (chunk_start, chunk_size) in file chunks
                Apply process_chunk asynchronously with pool
                Add result to results list

            Open output file for writing
            FOR each result in results list
                Get filtered words from result
                IF filtered words exist
                    Write filtered words to output file

            Write log entry for processed file and output file

    Close and join multiprocessing pool

    PRINT message indicating where filtered words are saved
END FUNCTION

FUNCTION process_chunk(chunk_start, chunk_size, lengths, filename, subchunk_size, encoding)
    Initialize empty list for filtered words

    Open file with specified encoding and seek to chunk_start position

    WHILE chunk_size > 0
        Read subchunk from file (size: min(subchunk_size, chunk_size))
```

Figure 7:Pseudocode DataFilteration-2

```
    WHILE chunk_size > 0
        Read subchunk from file (size: min(subchunk_size, chunk_size))
        Split subchunk into lines

        Filter lines by specified lengths
        Add filtered words to list

        Reduce chunk_size by subchunk_size
        IF no more lines in subchunk
            Break loop

    RETURN list of filtered words
END FUNCTION

FUNCTION filter_words_by_length(chunk, lengths)
    Initialize empty list for filtered words

    FOR each word in chunk
        IF length of word is in lengths
            Add word to filtered words list

    RETURN filtered words list
END FUNCTION

FUNCTION get_file_chunks(filename, num_chunks)
    Get size of file
```

Figure 8: Pseudocode DataFileration-3

```
    Calculate chunk size as file size divided by num_chunks
    Initialize empty list for chunks

    FOR i from 0 to num_chunks - 1
        Calculate chunk start as i * chunk_size
        Add (chunk_start, chunk_size) to list of chunks

    RETURN list of chunks
END FUNCTION
```

Figure 9: Pseudocode DataFilteration-4

4.2.1.3.5 Flowchart

Figure 10: DataFilteration Flowchart

Total number of Input passwords	8459063135
Length	**Total Count of Filtered Passwords**
5	10,751,871
6	563,608,354
7	465,597,114
8	1,357,729,013

Table 6: No. of Password as per Length

4.2.1.4 Splitting the Dataset

We divided our dataset into multiple 2 GB files for efficient processing and management. Our approach involves processing these chunks independently, which helps in handling large datasets without the need for significant memory resources. Each chunk is processed, and the results are aggregated to ensure comprehensive analysis. This method allows us to train, validate, and test our model effectively while managing data size constraints.

4.2.1.4.1 Pseudocode: Data Splitting

DataSpliter.py | Psuedocode Data Splitter

Psuedocode Data Splitter

```
START Main Script
    Set input directory
    Set output directory
    Set log file path
    Set max chunk size (default: 2GB)

    CALL process_files with input directory, output directory, log file, and max chunk size
END Main Script

FUNCTION process_files(input_dir, output_dir, log_file, max_chunk_size)
    IF output directory does not exist
        Create output directory

    Initialize file_counter to 1
    Initialize buffer as empty list
    Initialize current_size to 0

    FOR each file in input directory and its subdirectories
        Get full input file path
        IF file has already been processed (check log file)
            Continue to next file

        Get input file size
        Open input file and initialize progress bar

        WHILE reading chunks from input file
            Read chunk up to max_chunk_size
            IF no more data in chunk
```

Figure 11: Pseudocode DataSplitting-1

```
                Break loop

            FOR each line in chunk
                TRY
                    Add line to buffer
                    Increment current_size by line length + 1 (for newline character)
                    IF current_size exceeds max_chunk_size
                        Write buffer to new output file
                        Increment file_counter
                        Clear buffer
                        Reset current_size to 0
                EXCEPT error
                    Print error message
                    Continue to next line

            Update progress bar with chunk size
            Free memory after processing chunk

        Log processed file in log file

    IF buffer is not empty
        Write remaining buffer to new output file
END FUNCTION
```

Figure 12: Pseudocode DataSplitting-2

```
FUNCTION create_new_output_file(output_dir, file_counter)
    Create output file path using file_counter with zero padding
    Open output file for writing in UTF-8 encoding
    RETURN output file and its path
END FUNCTION

FUNCTION log_processed_file(log_file, input_file_path)
    Open log file for appending in UTF-8 encoding
    Write input file path to log file
    Close log file
END FUNCTION

FUNCTION is_file_processed(log_file, input_file_path)
    IF log file does not exist
        RETURN False

    Open log file for reading in UTF-8 encoding
    Read all processed file paths
    Close log file

    IF input file path is in processed file paths
        RETURN True

    RETURN False
END FUNCTION
```

Figure 13: Pseudocode Data Splitting-3

4.2.1.4.2 Flowchart:

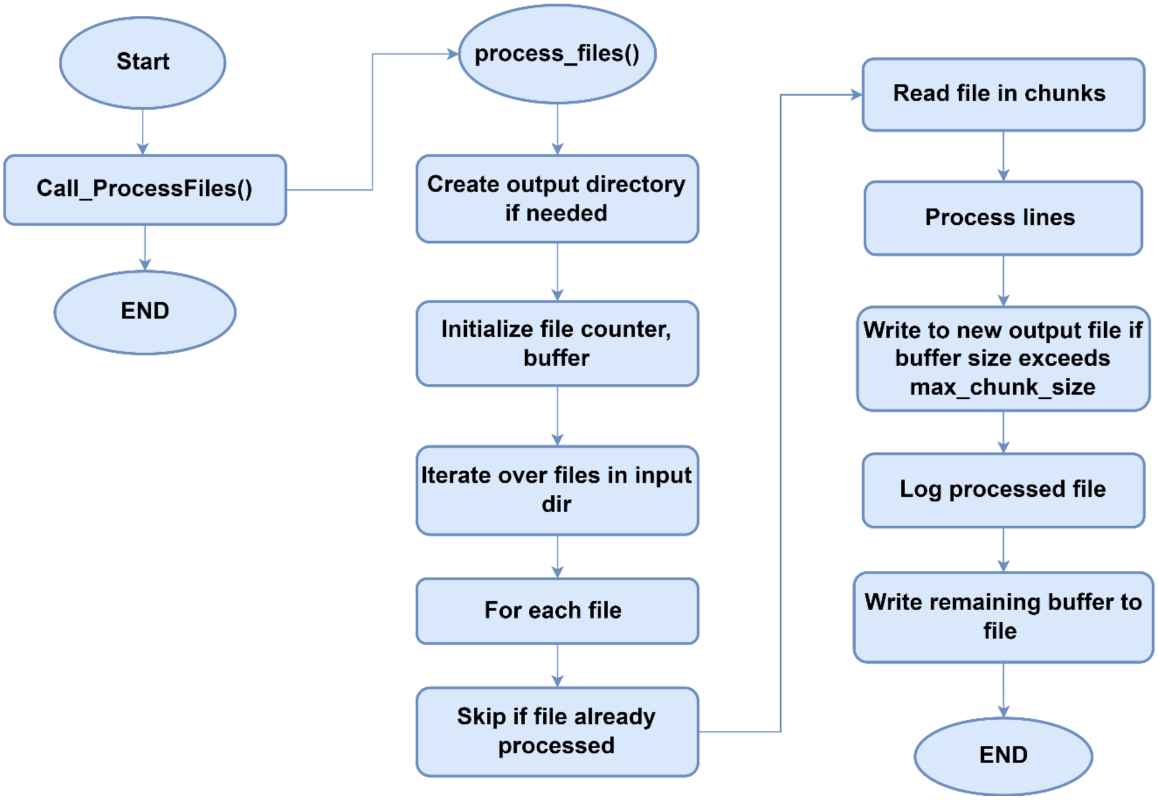

Figure 14: DataSplitting Flowchart

4.2.1.5 Save the Preprocessed Dataset

As the files contain datasets of huge volumes i.e. 120GB, 90GB, 100GB. We couldn't run these files. So, we split each file into multiples files which were approx. 2 GB and saved them as txt files and used the Data filtration technique to filter the passwords for training.

Chapter 5:
Develop and Train the AI Model

Chapter 5: Develop and Train the AI Model

5.1 Introduction

5.2 Research and select appropriate AI algorithms

5.3 Train and optimize the AI model with the selected datasets

Chapter 5: Develop and Train the AI Model

5.1 Introduction

The development and training of the AI model form the core of our multipurpose AI password analysis tool. This chapter focuses on the intricate process of building, fine-tuning, and optimizing various AI models to effectively analyze and predict password strength, generate secure passwords, and detect potential password vulnerabilities.

Utilizing cutting-edge technologies such as Generative Adversarial Networks (GANs), Recurrent Neural Networks (RNNs), and Transformers, the chapter outlines the step-by-step approach to model development. Starting with the selection of appropriate architectures and frameworks, we delve into the implementation details using TensorFlow, PyTorch, and cuDNN. The chapter emphasizes the importance of leveraging the computational power of the RTX 3070TI GPU to handle the intensive training processes efficiently.

5.2 Research and select appropriate AI algorithms

List of AI algorithms suitable for training on a password dataset and generating a dictionary file.

- Markov Models
- Generative Adversarial Networks (GANs)
- Variational Autoencoders (VAEs)
- Recurrent Neural Networks (RNNs)
- Transformer Models
- Probabilistic Context-Free Grammars (PCFGs)

5.2.1 Algorithms Overview

5.2.1.1 Markov Models

Markov Models are statistical models used to describe the probabilistic transitions between different states in a sequence of data. In the context of password generation, a Markov Model can be used to learn the statistical patterns and transitions between characters or character sequences in passwords.

Training Phase

a. Input

A dataset of passwords.

b. Preprocessing

The dataset is preprocessed to extract relevant features, such as character sequences or n-grams (sequences of n characters).

c. Model Construction

The Markov Model is constructed based on the observed transitions between characters or character sequences in the dataset.

d. Transition Probabilities

The model calculates transition probabilities between characters or character sequences based on their frequencies in the dataset. For example, the probability of transitioning from the character 'a' to 'b' or from the sequence 'ab' to 'cd' is estimated.

Generation Phase

a. Seed

A starting point (seed) is chosen to initiate the generation process. This can be a randomly chosen character or character sequence or a predefined starting point.

b. Random Walk

The model performs a random walk through the states (characters or character sequences) based on the learned transition probabilities. At each step, it selects the next state probabilistically according to the transition probabilities from the current state.

c. Generation Stop Criterion

Generation continues until a predefined stopping criterion is met, such as reaching a maximum password length or generating a certain number of passwords.

d. Output

The generated passwords are outputted as the result of the Markov Model.

5.2.1.2 Generative Adversarial Networks (GANs)

Generative Adversarial Networks (GANs) are a class of deep learning models consisting of two neural networks, the generator and the discriminator, which are trained simultaneously through a competitive process. GANs have been widely used for generating synthetic data that closely resembles real data distributions. In the context of password generation, GANs can be trained on a dataset of passwords to learn the underlying distribution of passwords and generate new passwords that mimic the characteristics of real passwords.

Training Phase

a. **Input**

   A dataset of passwords.

b. **Generator Network**

   The generator takes random noise as input and learns to generate synthetic passwords. Initially, the generator produces random noise that bears no resemblance to real passwords.

c. **Discriminator Network**

   The discriminator is trained to distinguish between real passwords from the dataset and fake passwords generated by the generator. It learns to differentiate between real and synthetic passwords.

d. **Adversarial Training**

   The generator and discriminator are trained simultaneously in a competitive manner. The generator aims to produce synthetic passwords that are indistinguishable from real passwords, while the discriminator aims to correctly classify real and fake passwords.

e. **Backpropagation**

   The errors from the discriminator are backpropagated to update the weights of both the generator and discriminator networks using techniques such as stochastic gradient descent (SGD) or variants like Adam.

Generation Phase

a. **Random Noise Input**

During the generation phase, the generator takes random noise vectors as input.

b. **Synthetic Password Generation**

The generator generates synthetic passwords by transforming the random noise vectors into meaningful password representations.

c. **Output**

The generated passwords are outputted as the result of the GAN.

Evaluation

The quality of the generated passwords can be evaluated using metrics such as similarity to real passwords, diversity, and entropy. Additionally, human evaluators can assess the usability and security of the generated passwords.

5.2.1.2.1 Pseudocode: (For Generating Password)

Psuedocode GAn generate > ...

```python
import torch
import torch.nn as nn
import numpy as np

# Define Generator class for word generation
class Generator(nn.Module):
    def __init__(self, latent_size, hidden_size, num_layers, num_classes):
        super(Generator, self).__init__()
        # Define architecture: LSTM -> Linear
        self.lstm = nn.LSTM(latent_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_classes)

    def forward(self, x, h):
        out, h = self.lstm(x, h)
        out = self.fc(out)
        return out, h

    def init_hidden(self, batch_size):
        # Initialize hidden state for LSTM
        return (torch.zeros(self.num_layers, batch_size, self.hidden_size).to(device),
                torch.zeros(self.num_layers, batch_size, self.hidden_size).to(device))

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Load vocabulary
char_to_idx = torch.load('char_to_idx.pth')
idx_to_char = torch.load('idx_to_char.pth')
chars = sorted(char_to_idx.keys())

latent_size = 100
hidden_size = 128
num_layers = 2
max_length = 10  # Maximum length of generated words

# Load trained Generator model
G = Generator(latent_size, hidden_size, num_layers, len(chars)).to(device)
G.load_state_dict(torch.load('gan_generator.pth'))

def generate_word(G, length):
    G.eval()
    z = torch.randn(1, length, latent_size).to(device)
```

Figure 15: Pseudocode GAN for Generation passwords

```python
    h = G.init_hidden(1)
    with torch.no_grad():
        generated_data, _ = G(z, h)
        generated_data = torch.softmax(generated_data, dim=-1)
        generated_data = torch.argmax(generated_data, dim=-1).squeeze().cpu().numpy()

    word = ''.join([idx_to_char[idx] for idx in generated_data])
    return word

# Example usage
start_char = 'a'  # Starting character for word generation
word_length = 5   # Length of the word to generate
new_word = generate_word(G, word_length)
print(f'Generated word: {new_word}')

```

Figure 16: Pseudocode GAN for Generating Passwords-2

5.2.1.2.2 Flowchart

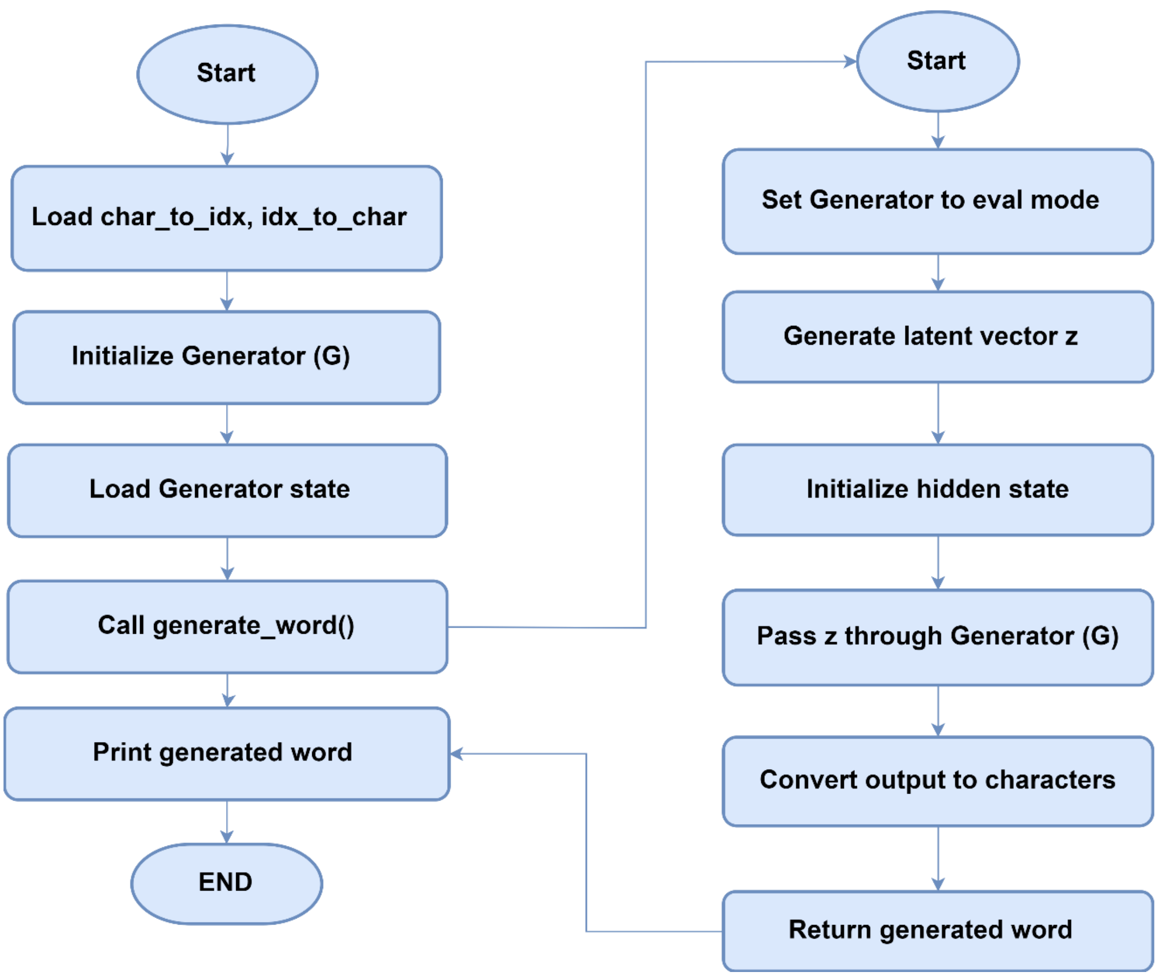

Figure 17: Flowchart of GAN for Generating Passwords

5.2.1.2.3 Pseudocode (For Training)

```python
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np

# Define Generator and Discriminator classes
class Generator(nn.Module):
    def __init__(self, latent_size, hidden_size, num_layers, num_classes):
        super(Generator, self).__init__()
        # Define architecture: LSTM -> Linear
        self.lstm = nn.LSTM(latent_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_classes)

    def forward(self, x, h):
        out, h = self.lstm(x, h)
        out = self.fc(out)
        return out, h

    def init_hidden(self, batch_size):
        # Initialize hidden state for LSTM
        return (torch.zeros(self.num_layers, batch_size, self.hidden_size).to(device),
                torch.zeros(self.num_layers, batch_size, self.hidden_size).to(device))

class Discriminator(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers):
        super(Discriminator, self).__init__()
        # Define architecture: LSTM -> Linear
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, 1)

    def forward(self, x, h):
        out, h = self.lstm(x, h)
        out = self.fc(out[:, -1, :])
        return torch.sigmoid(out)

    def init_hidden(self, batch_size):
        # Initialize hidden state for LSTM
        return (torch.zeros(self.num_layers, batch_size, self.hidden_size).to(device),
                torch.zeros(self.num_layers, batch_size, self.hidden_size).to(device))
```

Figure 18: Pseudocode for GAN's Training

```
# Training parameters
latent_size = 100
hidden_size = 128
num_layers = 2
num_epochs = 10
learning_rate = 0.0002
batch_size = 64

# Initialize models, criterion, optimizers
G = Generator(latent_size, hidden_size, num_layers, len(chars)).to(device)
D = Discriminator(len(chars), hidden_size, num_layers).to(device)
criterion = nn.BCELoss()
G_optimizer = optim.Adam(G.parameters(), lr=learning_rate)
D_optimizer = optim.Adam(D.parameters(), lr=learning_rate)

# Training loop
for epoch in range(num_epochs):
    for i in range(0, len(sequences), batch_size):
        # Train Discriminator
        D_optimizer.zero_grad()
        real_data = sequences[i:i+batch_size]
        h_real = D.init_hidden(batch_size)
        outputs = D(real_data, h_real)
        D_loss_real = criterion(outputs, real_labels)
        D_loss_real.backward()

        z = torch.randn(batch_size, max_length, latent_size).to(device)
        h_fake = G.init_hidden(batch_size)
        fake_data, _ = G(z, h_fake)
        fake_data = torch.softmax(fake_data, dim=-1)

        h_fake = D.init_hidden(batch_size)
        outputs = D(fake_data.detach(), h_fake)
        D_loss_fake = criterion(outputs, fake_labels)
        D_loss_fake.backward()

        D_optimizer.step()

        # Train Generator
        G_optimizer.zero_grad()
```

Figure 19: **Pseudocode for GAN's Training-2**

```
        h_fake = D.init_hidden(batch_size)
        outputs = D(fake_data, h_fake)
        G_loss = criterion(outputs, real_labels)
        G_loss.backward()

        G_optimizer.step()

    print(f'Epoch [{epoch+1}/{num_epochs}], D Loss: {D_loss.item():.4f}, G Loss: {G_loss.item():.4f}')

# Save models and vocabulary
torch.save(G.state_dict(), 'gan_generator.pth')
torch.save(D.state_dict(), 'gan_discriminator.pth')
torch.save(char_to_idx, 'char_to_idx.pth')
torch.save(idx_to_char, 'idx_to_char.pth')
```

Figure 20: Pseudocode for GAN's Training-3

5.2.1.2.4 Flowchart

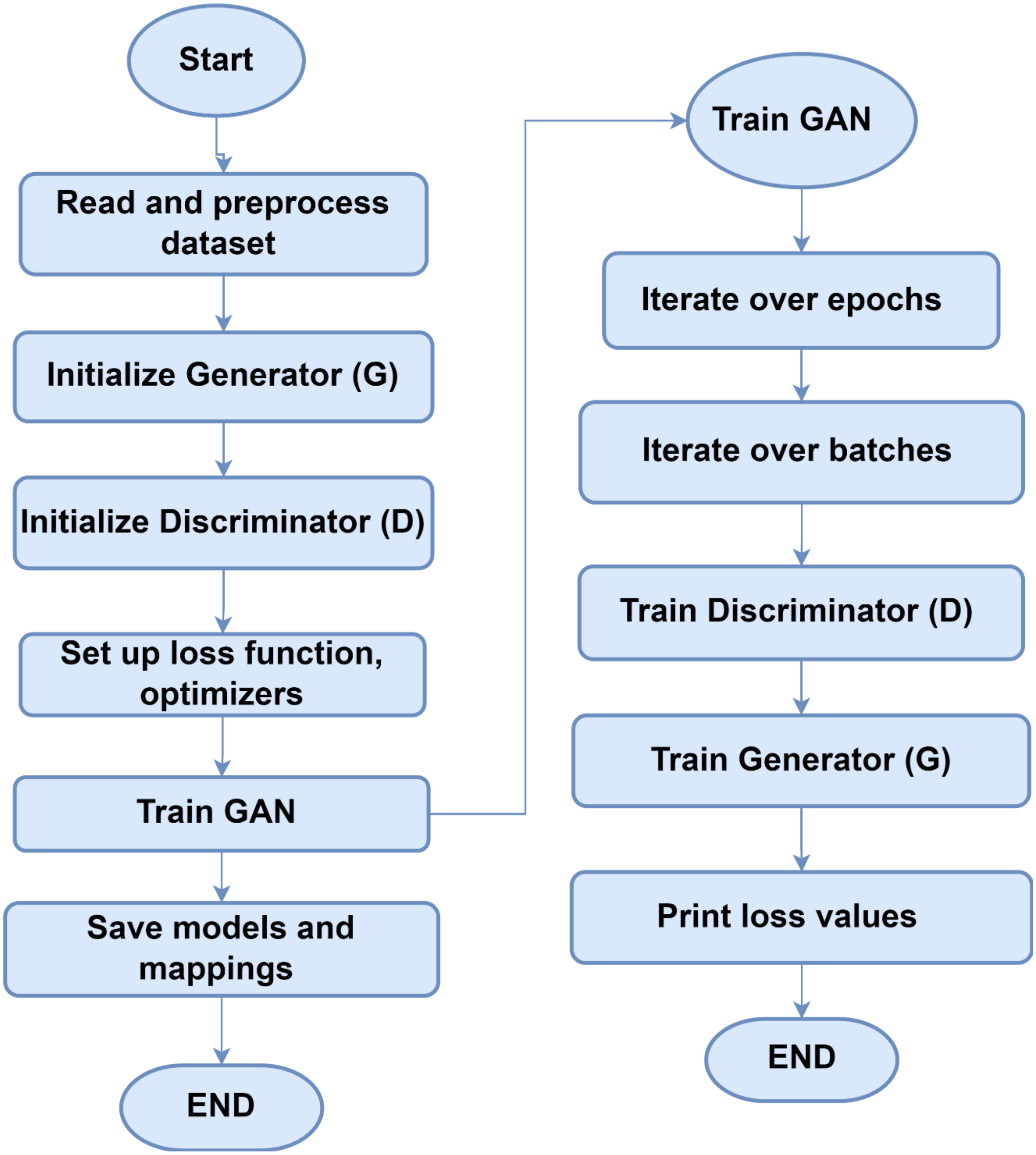

Figure 21: Flowchart of GAN's Training Module

5.2.1.3 Variational Autoencoders (VAEs)

Variational Autoencoders (VAEs) are a type of generative model that learns to encode input data into a lower-dimensional latent space and decode it back into the original data distribution. VAEs are trained to capture the underlying structure of the data distribution, allowing them to generate new samples that closely resemble the input data. In the context of password generation, VAEs can be trained on a dataset of passwords to learn the distribution of passwords and generate new passwords that exhibit similar characteristics.

Training Phase

a. **Input**

   A dataset of passwords.

b. **Encoder Network**

   The encoder takes input passwords and maps them to a lower-dimensional latent space, where each point represents a compressed representation of a password.

c. **Decoder Network**

   The decoder takes points from the latent space and reconstructs them into passwords. The decoder aims to reconstruct passwords that are similar to the input passwords.

d. **Variational Inference**

   VAEs incorporate variational inference, which involves learning a probabilistic distribution over the latent space. During training, the encoder learns to approximate this distribution, while the decoder learns to reconstruct passwords from samples drawn from this distribution..

e. **Reconstruction Loss**

   The VAE is trained to minimize the reconstruction loss, which measures the difference between the original passwords and the reconstructed passwords.

f. **Regularization**

   VAEs also incorporate a regularization term, such as the Kullback-Leibler (KL) divergence, to encourage the learned latent space to follow a prior distribution (e.g., Gaussian distribution).

Generation Phase

a. **Sampling**

During the generation phase, random points are sampled from the latent space.

b. **Decoder Output**

The decoder takes these sampled points and generates new passwords by reconstructing them into the password space.

c. **Output**

The generated passwords are output as the result of the VAE.

Evaluation

The quality of the generated passwords can be evaluated using metrics such as similarity to real passwords, diversity, and entropy. Human evaluators can also assess the usability and security of the generated passwords.

5.2.1.4 Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs) are a class of neural networks designed to handle sequential data by maintaining a hidden state that captures information about previous inputs in the sequence. RNNs are well-suited for tasks where the input data's temporal order is important, making them a suitable choice for password generation, where the order of characters matters.

Training Phase

a. **Input**

A dataset of passwords represented as sequences of characters.

b. **Architecture**

The RNN consists of recurrent units (such as LSTM - Long Short-Term Memory, or GRU - Gated Recurrent Unit) that process one character at a time while maintaining a hidden state. This hidden state captures information about previous characters in the password sequence.

c. **Sequence Learning**

The RNN is trained to predict the next character in the sequence given the previous characters. This is done by feeding each character in the sequence into the RNN and comparing the predicted next character with the actual next character in the training data.

d. **Backpropagation Through Time (BPTT)**

   The errors are backpropagated through time to update the weights of the RNN, allowing it to learn the patterns and dependencies present in the password dataset.

e. **Loss Function**

   The RNN is trained to minimize a loss function, such as categorical cross-entropy, which measures the difference between the predicted and actual characters in the sequence.

Generation Phase

a. **Seed**

   During the generation phase, a seed sequence is provided as input to the RNN to initiate the generation process. This seed sequence can be randomly chosen or predefined.

b. **Character-by-Character Generation**

   RNN generates new characters one at a time by feeding the previous character and the current hidden state back into the network. The output character is sampled from the predicted probability distribution over the character vocabulary.

c. **Output**

   The generated characters are concatenated to form a new password sequence.

Evaluation

The quality of the generated passwords can be evaluated using metrics such as similarity to real passwords, diversity, and entropy. Human evaluators can also assess the usability and security of the generated passwords.

5.2.1.4.1 Pseudocode (For Generating Passwords)

Pseudocode RNN Generate > RNNModel > init_hidden

```python
import torch
import torch.nn as nn
import numpy as np

# Define RNN model for word generation
class RNNModel(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, num_classes):
        super(RNNModel, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.rnn = nn.RNN(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_classes)

    def forward(self, x, h):
        out, h = self.rnn(x, h)
        out = self.fc(out.reshape(out.size(0) * out.size(1), out.size(2)))
        return out, h

    def init_hidden(self, batch_size):
        return torch.zeros(self.num_layers, batch_size, self.hidden_size).to(device)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Load vocabulary
char_to_idx = torch.load('char_to_idx.pth')
idx_to_char = torch.load('idx_to_char.pth')
chars = sorted(char_to_idx.keys())

input_size = len(chars)
hidden_size = 64
num_layers = 2

# Load trained RNN model
model = RNNModel(input_size, hidden_size, num_layers, len(chars)).to(device)
model.load_state_dict(torch.load('rnn_model.pth'))

def generate_word(model, start_char, length):
    model.eval()
    h = model.init_hidden(1)
    input = torch.eye(len(chars))[char_to_idx[start_char]].unsqueeze(0).unsqueeze(0).to(device)
    word = start_char
```

Figure 22: Pseudocode RNN for Generating Passwords

```python
    word = start_char

    with torch.no_grad():
        for _ in range(length - 1):
            output, h = model(input, h)
            _, top_idx = torch.topk(output[-1], 1)
            generated_idx = top_idx.item()

            if generated_idx not in idx_to_char:
                break

            next_char = idx_to_char[generated_idx]
            word += next_char
            input = torch.eye(len(chars))[char_to_idx[next_char]].unsqueeze(0).unsqueeze(0).to(device)

    return word

# Example usage
start_char = 's'  # Starting character for word generation
word_length = 9   # Length of the word to generate
new_word = generate_word(model, start_char, word_length)
print(f'Generated word: {new_word}')
```

Figure 23: Pseudocode RNN for Generating Passwords-1

5.2.1.4.2 Flowchart

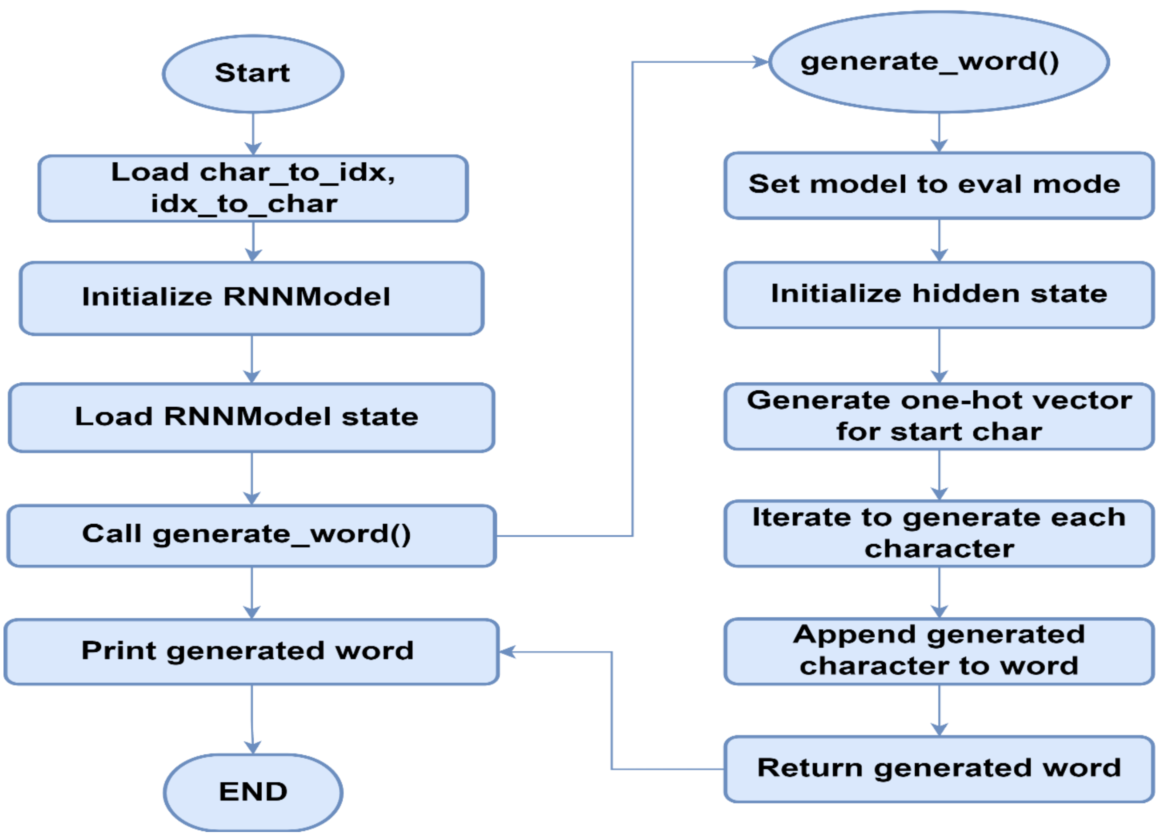

Figure 24: flowchart of RNN for Generating Passwords

5.2.1.4.3 Pseudocode (For Training)

```python
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
from tqdm import tqdm

# Parameters
hidden_size = 64
num_layers = 2
num_epochs = 10
learning_rate = 0.003

# Read dataset
def read_words(file_path):
    with open(file_path, 'r', encoding='utf-8') as file:
        words = file.read().splitlines()
    return words

words = read_words('path_to_your_dataset.txt')

# Create character vocabulary
chars = sorted(list(set(''.join(words))))
char_to_idx = {ch: i for i, ch in enumerate(chars)}
idx_to_char = {i: ch for i, ch in enumerate(chars)}

input_size = len(chars)

# Preprocess dataset
def preprocess(words, char_to_idx):
    sequences = []
    for word in words:
        sequences.append([char_to_idx[char] for char in word])
    return sequences

sequences = preprocess(words, char_to_idx)

# Define RNN model
class RNNModel(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, num_classes):
        super(RNNModel, self).__init__()
        self.hidden_size = hidden_size
```

Figure 25: Pseudocode for RNN's Training

```python
        self.num_layers = num_layers
        self.rnn = nn.RNN(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_classes)

    def forward(self, x, h):
        out, h = self.rnn(x, h)
        out = self.fc(out.reshape(out.size(0) * out.size(1), out.size(2)))
        return out, h

    def init_hidden(self, batch_size):
        return torch.zeros(self.num_layers, batch_size, self.hidden_size).to(device)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = RNNModel(input_size, hidden_size, num_layers, len(chars)).to(device)

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Training loop
for epoch in range(num_epochs):
    model.train()
    h = model.init_hidden(1)
    loss_avg = 0
    with tqdm(total=len(sequences), desc=f'Epoch {epoch+1}/{num_epochs}') as pbar:
        for seq in sequences:
            inputs = torch.eye(len(chars))[seq[:-1]].unsqueeze(0).to(device)
            targets = torch.tensor(seq[1:], dtype=torch.long).to(device)

            h = h.detach()
            outputs, h = model(inputs, h)

            loss = criterion(outputs, targets.view(-1))
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

```

Figure 26: Pseudocode for RNN's Traning-2

```python
            loss_avg += loss.item()
            pbar.update(1)

    print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss_avg/len(sequences):.4f}')

# Save model and vocabulary
torch.save(model.state_dict(), 'rnn_model.pth')
torch.save(char_to_idx, 'char_to_idx.pth')
torch.save(idx_to_char, 'idx_to_char.pth')

```

Figure 27: Pseudocode for RNN's Traning-3

5.2.1.4.4 Flowchart

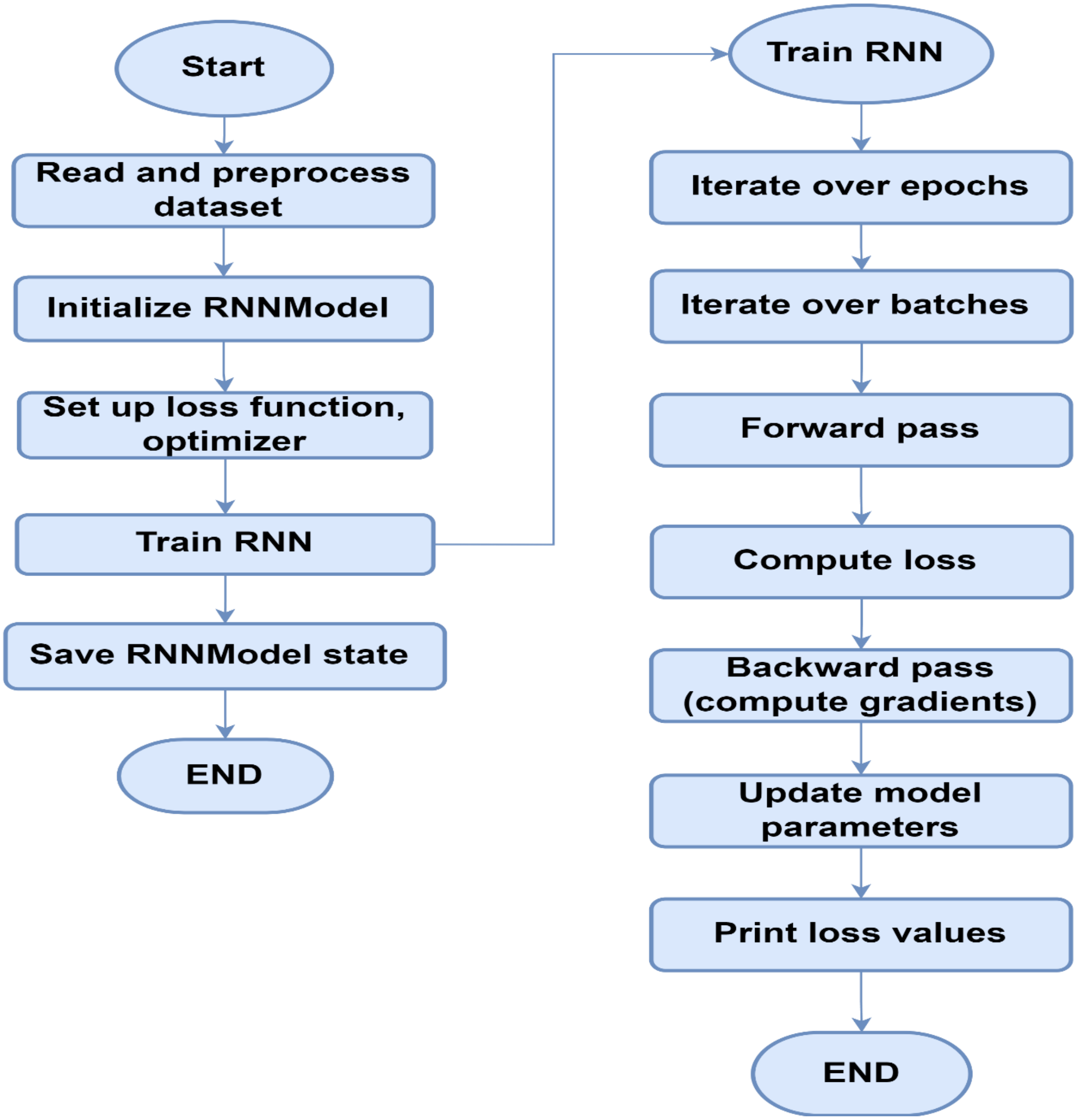

Figure 28: Flowchart of RNN's Training

5.2.1.5 Transformer Models

Transformer models are a type of neural network architecture that has gained significant popularity in natural language processing tasks due to its ability to capture long-range dependencies in sequential data efficiently. Transformer models, such as OpenAI's GPT (Generative Pre-trained Transformer) and Google's BERT (Bidirectional Encoder Representations from Transformers), are powerful language models capable of generating coherent text based on input sequences. In the context of password generation, transformer models can be fine-tuned on a dataset of passwords to learn the distribution of passwords and generate new passwords that exhibit similar characteristics.

Training Phase

a. **Input**

   A dataset of passwords represented as sequences of characters.

b. **Architecture**

   The transformer model consists of encoder and decoder layers, each composed of self-attention mechanisms and feed-forward neural networks. The encoder processes the input sequence, while the decoder generates the output sequence.

c. **Pre-Training**

   The transformer model is pre-trained on a large corpus of text data using unsupervised learning objectives, such as language modeling or masked language modeling. This pre-training step allows the model to learn general language patterns and representations.

d. **Fine-Tuning**

   The pre-trained transformer model is fine-tuned on the password dataset using supervised learning. During fine-tuning, the model learns to generate passwords by minimizing a loss function that measures the difference between the predicted and actual passwords in the dataset.

e. **Tokenization**

   The input passwords are tokenized into subword or character-level tokens to represent them as numerical inputs to the transformer model.

Generation Phase

a. **Seed**

   During the generation phase, a seed sequence is provided as input to the transformer model to initiate the generation process. This seed sequence can be randomly chosen or predefined.

b. Autoregressive Generation

The transformer model generates new characters or tokens autoregressively, meaning that it predicts each token based on the previously generated tokens. At each step, the model outputs a probability distribution over the vocabulary of characters or tokens, from which the next token is sampled.

c. Output

The generated characters or tokens are concatenated to form a new password sequence.

Evaluation

The quality of the generated passwords can be evaluated using metrics such as similarity to real passwords, diversity, and entropy. Human evaluators can also assess the usability and security of the generated passwords.

5.2.1.5.1 Pseudocode (For Generation Passwords)

Pseudocode transformers Generate passwords > ...

```python
import torch
from transformers import GPT2Tokenizer, GPT2LMHeadModel

# Check for GPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")

# Load model and tokenizer
print("Loading model and tokenizer...")
model = GPT2LMHeadModel.from_pretrained('path_to_saved_model_directory')  # Update with your saved model path
tokenizer = GPT2Tokenizer.from_pretrained('path_to_saved_model_directory')  # Update with the same path as above

# Move model to GPU if available
model.to(device)
model.eval()
print("Model and tokenizer loaded.")

# Generate new words
def generate_words(seed_text, max_length=50):
    print(f"Generating new words for seed text: '{seed_text}'")
    input_ids = tokenizer.encode(seed_text, return_tensors='pt')
    input_ids = input_ids.to(device)  # Move input to GPU
    output = model.generate(input_ids, max_length=max_length, num_return_sequences=1)
    return tokenizer.decode(output[0], skip_special_tokens=True)

# Example usage
seed_text = "your_seed_text_here"  # Replace with your desired seed text
generated_text = generate_words(seed_text)
print("Generated text:", generated_text)

```

Figure 29: Pseudocode of Transformers for Generating Passwords

5.2.1.5.2 Flowchart

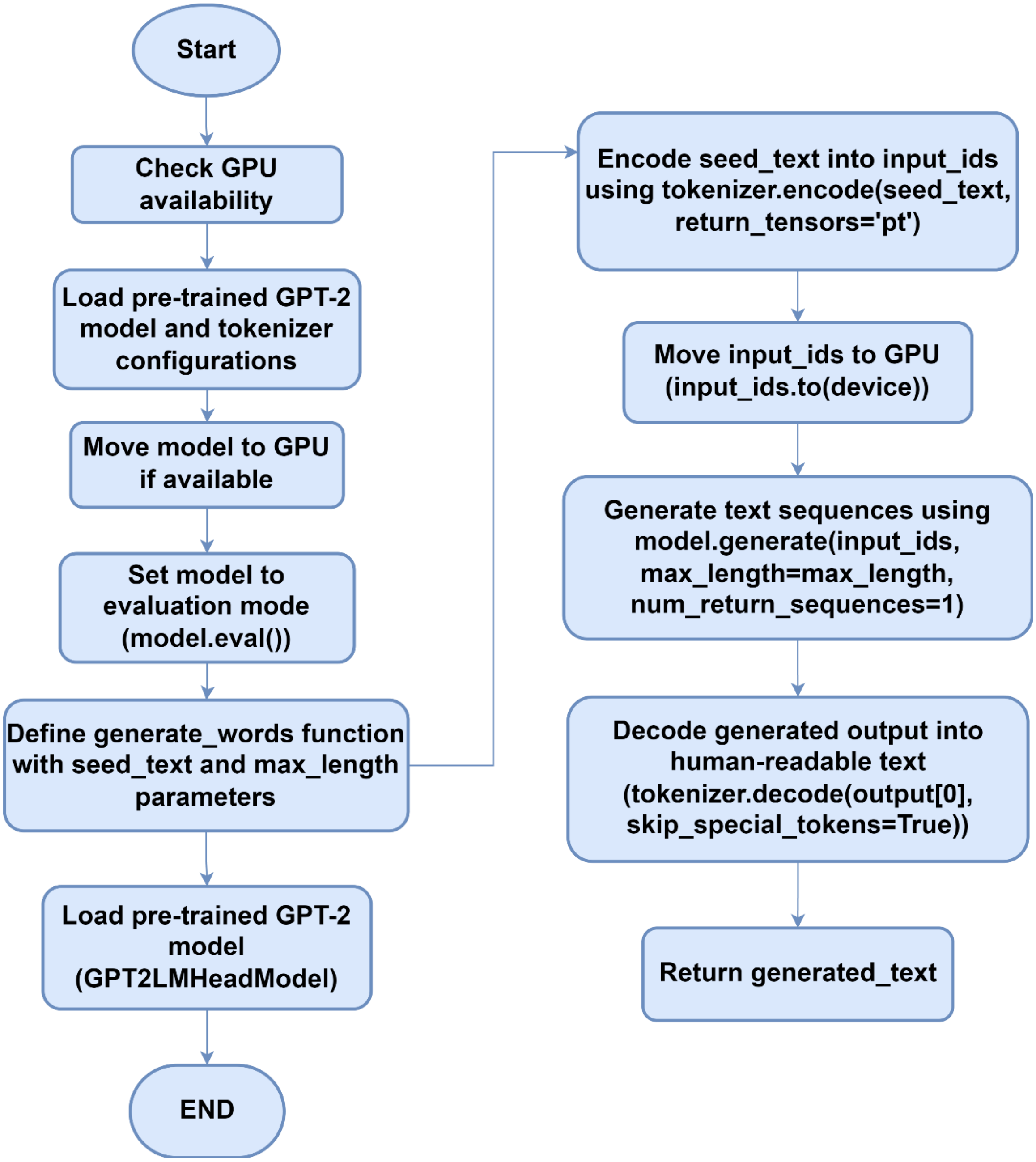

Figure 30: Flowchart of Transformers for Generating Passwords

5.2.1.5.3 Pseudocode (For Training)

Pseudocode Transformers Training > ...

```
import torch
from transformers import GPT2Tokenizer, GPT2LMHeadModel, AdamW
from torch.utils.data import Dataset, DataLoader
from tqdm import tqdm

# Check for GPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")

# Load and preprocess data
def load_words(file_path):
    with open(file_path, 'r', encoding='utf-8') as file:
        words = file.read().splitlines()
    return words

print("Loading words from file...")
words = load_words('path_to_your_dataset.txt')  # Update with your file path
print(f"Loaded {len(words)} words.")

print("Initializing tokenizer and tokenizing words...")
tokenizer = GPT2Tokenizer.from_pretrained('gpt2')

# Add padding token if not already added
if tokenizer.pad_token is None:
    tokenizer.add_special_tokens({'pad_token': '[PAD]'})

# Tokenize words and create dataset
tokens = tokenizer(words, return_tensors='pt', padding=True, truncation=True)
dataset = torch.utils.data.TensorDataset(tokens['input_ids'], tokens['attention_mask'])

# Define data loader
dataloader = DataLoader(dataset, batch_size=2, shuffle=True)
print("Data loader created.")

# Define pre-trained GPT-2 model
print("Loading pre-trained GPT-2 model...")
model = GPT2LMHeadModel.from_pretrained('gpt2')

# Resize model embeddings to account for the new special tokens
model.resize_token_embeddings(len(tokenizer))
```

Figure 31: Pseudocode of Transformers for Training

```python

# Move model to GPU if available
model.to(device)

# Training setup
optimizer = AdamW(model.parameters(), lr=5e-5)
model.train()
print("Training setup complete.")

# Determine the number of steps per epoch
steps_per_epoch = len(dataloader)
print(f"Steps per epoch: {steps_per_epoch}")

# Training loop
num_epochs = 3
print("Starting training...")
for epoch in range(num_epochs):
    print(f"Epoch {epoch+1}/{num_epochs}")
    epoch_iterator = tqdm(dataloader, desc="Training")
    for step, batch in enumerate(epoch_iterator):
        optimizer.zero_grad()
        input_ids, attention_mask = batch
        input_ids, attention_mask = input_ids.to(device), attention_mask.to(device)

        # Forward pass
        outputs = model(input_ids, attention_mask=attention_mask, labels=input_ids)
        loss = outputs.loss

        # Backward pass
        loss.backward()
        optimizer.step()

        # Update progress bar
        epoch_iterator.set_postfix({'loss': loss.item(), 'step': step+1})

    print(f"Epoch {epoch+1} completed.")

# Save model and tokenizer
print("Saving model...")
model.save_pretrained('path_to_save_model_directory')  # Update with your desired save path
tokenizer.save_pretrained('path_to_save_model_directory')  # Update with the same path as above
```

Figure 32: Pseudocode of Transformers for Training-2

```python
print("Model saved.")

```

Figure 33: Pseudocode of Transformers for Training-3

5.2.1.5.4 Flowchart

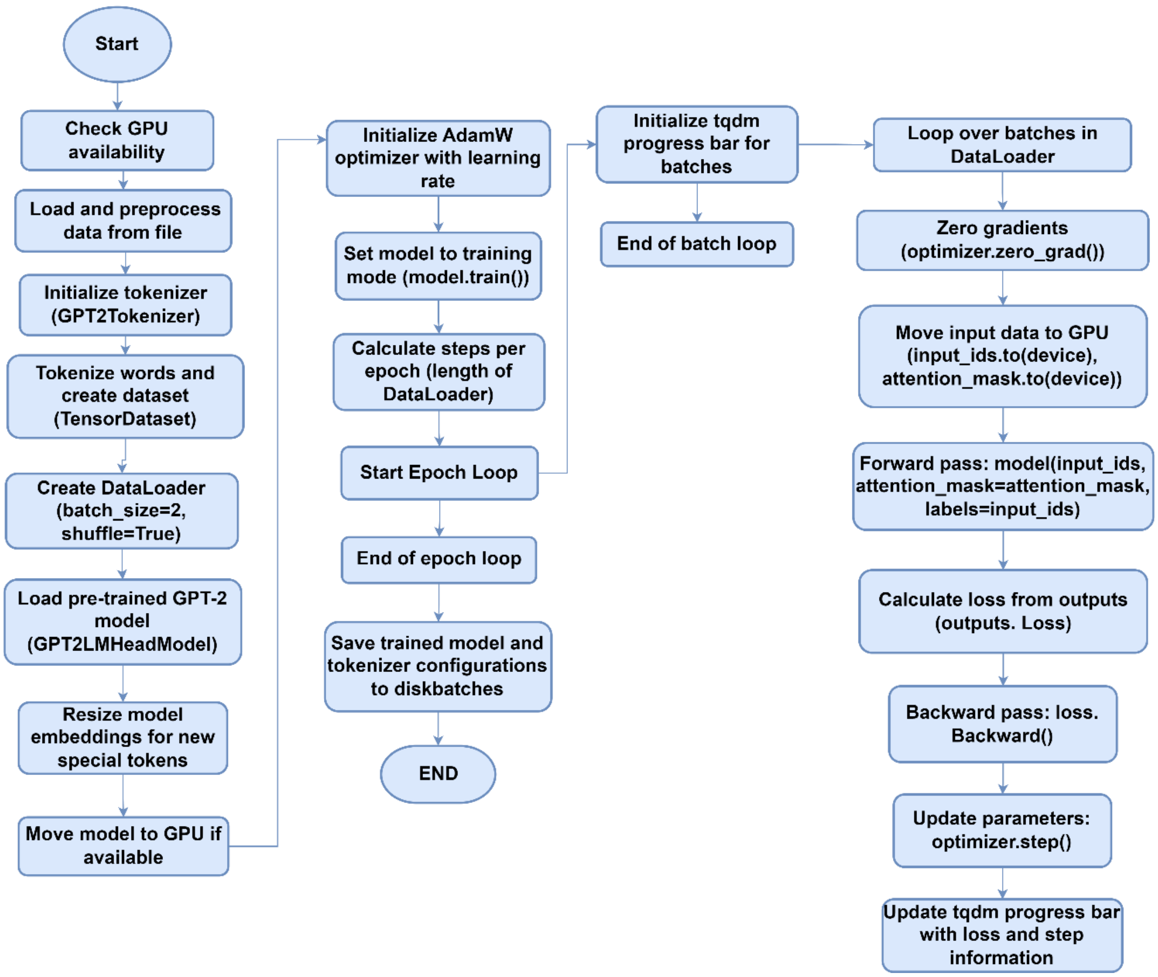

Figure 34: Flowchart of Transformers for Training

5.2.1.6 Probabilistic Context-Free Grammars (PCFGs)

Probabilistic Context-Free Grammars (PCFGs) are formal grammars used to model the hierarchical structure of sequences, where each production rule is associated with a probability indicating its likelihood of being applied. PCFGs are commonly used in natural language processing and computational linguistics for tasks such as syntax parsing and language generation. In the context of password generation, PCFGs can be used to model the structural patterns and dependencies present in passwords and generate new passwords that adhere to these patterns.

Training Phase

a. **Input**

A dataset of passwords represented as sequences of characters.

b. Grammar Construction

From the dataset, a PCFG is constructed by identifying common structural patterns and dependencies present in the passwords. Each production rule in the grammar represents a possible transformation or combination of characters.

c. Rule Probabilities

Each production rule in the PCFG is associated with a probability indicating its likelihood of being applied. These probabilities are estimated based on their frequencies in the training dataset.

d. Smoothing

To handle unseen or rare patterns, smoothing techniques may be applied to adjust the probabilities of production rules.

Generation Phase

a. Seed

During the generation phase, a seed symbol or starting point is provided as input to the PCFG to initiate the generation process. This seed symbol could represent a starting character or character sequence.

b. Probabilistic Expansion

The PCFG probabilistically expands the seed symbol into a sequence of characters by recursively applying production rules according to their probabilities. At each step, a production rule is chosen based on its probability, and the corresponding symbols are generated.

c. Output

The generated characters are concatenated to form a new password sequence.

Evaluation

The quality of the generated passwords can be evaluated using metrics such as similarity to real passwords, diversity, and entropy. Human evaluators can also assess the usability and security of the generated passwords.

5.3 Train and optimize the AI model with the selected datasets.

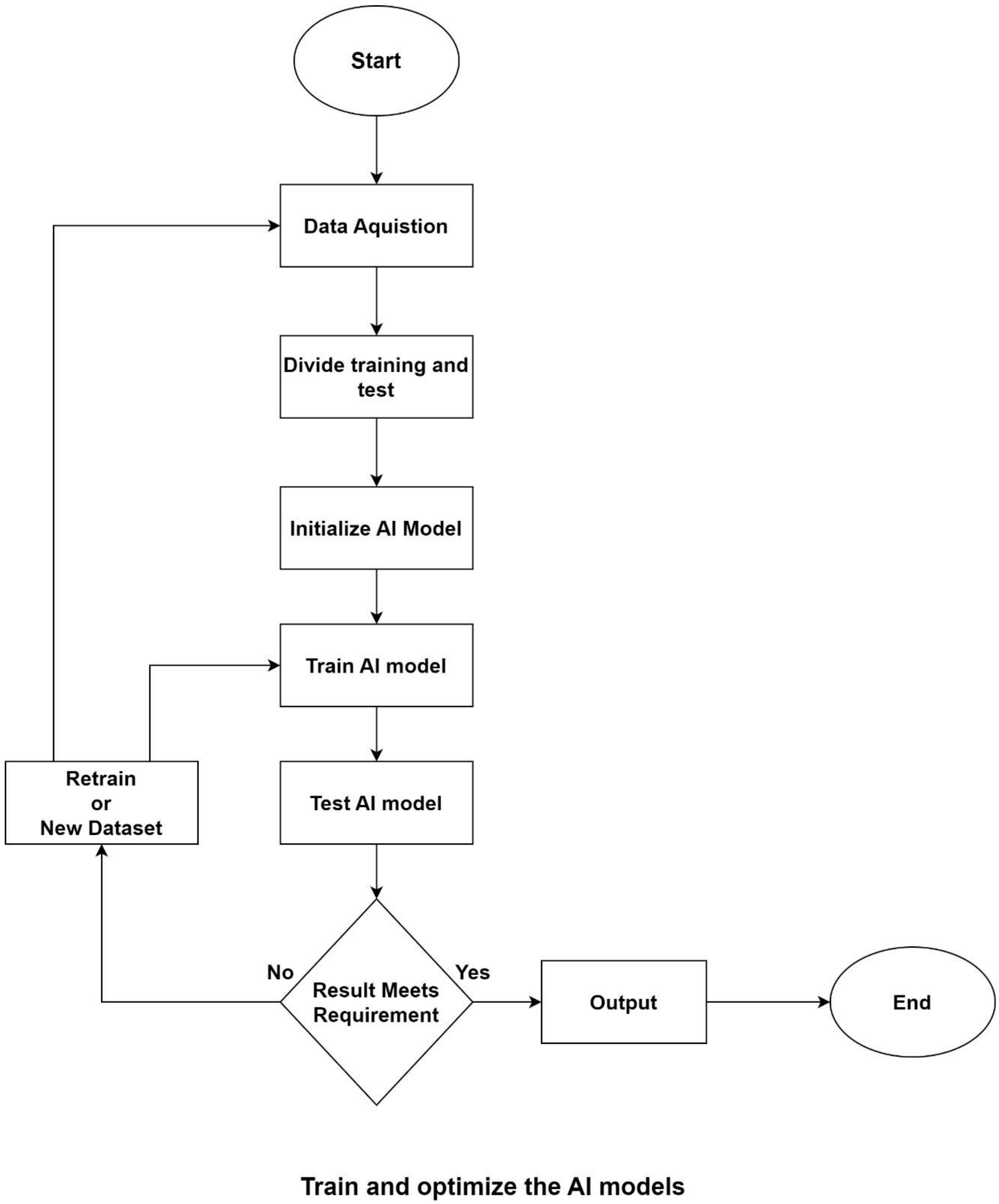

Figure 35: Training AI Model Flow Chart

References

- Hitaj, Briland et al. “PassGAN: A Deep Learning Approach for Password Guessing.” *International Conference on Applied Cryptography and Network Security* (2017).
- Hitaj, Briland, et al. "Passgan: A deep learning approach for password guessing." *Applied Cryptography and Network Security: 17th International Conference, ACNS 2019, Bogota, Colombia, June 5–7, 2019, Proceedings 17*. Springer International Publishing, 2019.

Appendix

Appendix A: Data Preparation Details

Table A.1: Collected Datasets

No	Name	Size
1	rockyou2021.txt dictionary from kys234 on RaidForums	12.7 GB
2	36.4GB-18_in_1.lst_2.7z	4.50 GB
3	ASLM.txt.7z	127 MB
4	b0n3z_dictionary-SPLIT-BY-LENGTH-34.6GB_2.7z	3.29 GB
5	b0n3z-wordlist-sorted_REPACK-69.3GB_3.7z	9.07 GB
6	bad-passwords-master.zip	1.34 MB
7	crackstation.txt.gz	4.19 GB
8	dictionaries-master.zip	19.3 MB
9	Password lists.zip	336 MB
10	password-list-main.zip	291 MB
11	password-lists-master.zip	8.86 MB
12	pastePasswordLists-main_2.zip	54.6 MB
13	PowerSniper-master.zip	0.3 MB
14	pwlist-master.zip	8.02 MB
15	rockyou.zip	41.7 MB
16	SecLists-master.zip	554 MB
17	statistically-likely-usernames-master.7z	9.07 MB
18	vietnam-password-lists-master.zip	5.14 MB
19	wpa-passwords-master.zip	5.79 MB
20	WPA-PSK WORDLIST 3 Final (13 GB).rar	4.49 GB
21	cyclone.hashesorg.hashkiller.combined.7z	6.59 GB

Figure A.1: Dataset Downloaded Screenshot

Appendix B: Data Filtration

Table A.2: Filtered Passwords Count

Total number of Input passwords	8459063135
Length	**Total Count of Filtered Passwords**
5	10,751,871
6	563,608,354
7	465,597,114
8	1,357,729,013

Appendix C: Data Splitting

Pseudocode: Data Splitting

DataSpliter.py | Psuedocode Data Splitter

Psuedocode Data Splitter

```
START Main Script
    Set input directory
    Set output directory
    Set log file path
    Set max chunk size (default: 2GB)

    CALL process_files with input directory, output directory, log file, and max chunk size
END Main Script

FUNCTION process_files(input_dir, output_dir, log_file, max_chunk_size)
    IF output directory does not exist
        Create output directory

    Initialize file_counter to 1
    Initialize buffer as empty list
    Initialize current_size to 0

    FOR each file in input directory and its subdirectories
        Get full input file path
        IF file has already been processed (check log file)
            Continue to next file

        Get input file size
        Open input file and initialize progress bar

        WHILE reading chunks from input file
            Read chunk up to max_chunk_size
            IF no more data in chunk
                Break loop

            FOR each line in chunk
                TRY
                    Add line to buffer
                    Increment current_size by line length + 1 (for newline character)
                    IF current_size exceeds max_chunk_size
                        Write buffer to new output file
                        Increment file_counter
                        Clear buffer
                        Reset current_size to 0
                EXCEPT error
                    Print error message
                    Continue to next line

            Update progress bar with chunk size
            Free memory after processing chunk

        Log processed file in log file

    IF buffer is not empty
        Write remaining buffer to new output file
END FUNCTION
```

```

FUNCTION create_new_output_file(output_dir, file_counter)
    Create output file path using file_counter with zero padding
    Open output file for writing in UTF-8 encoding
    RETURN output file and its path
END FUNCTION

FUNCTION log_processed_file(log_file, input_file_path)
    Open log file for appending in UTF-8 encoding
    Write input file path to log file
    Close log file
END FUNCTION

FUNCTION is_file_processed(log_file, input_file_path)
    IF log file does not exist
        RETURN False

    Open log file for reading in UTF-8 encoding
    Read all processed file paths
    Close log file

    IF input file path is in processed file paths
        RETURN True

    RETURN False
END FUNCTION
```

Flowchart: Data Splitting