IJNRD.ORG ISSN : 2456-4184
INTERNATIONAL JOURNAL OF NOVEL RESEARCH AND DEVELOPMENT (IJNRD) | IJNRD.ORG
An International Open Access, Peer-reviewed, Refereed Journal

A SECURE PASSWORD GENERATOR USING PYTHON WITH GUI AND CUI SUPPORT

Chaitanya Manjunjath Hegde [1], Adarsh khelge[2]

A vishal[3], Chandan LN[4] ,Dr, Nirmala S[5]

Student [1,2,3,4] , Prof[5] ,Dept. of CSE,

AMC ENGINEERING COLLEGE, BENGALURU, KARNATAKA, INDIA

Abstract

In the era of increasing digital threats, the importance of secure and strong passwords cannot be overstated. This paper presents a Python-based password generator that enhances security through customizable features and dual-interface access—CLI and GUI. The system allows users to generate secure passwords with adjustable complexity, evaluate password strength, copy results to the clipboard, and log them with descriptive labels and timestamps. The application promotes good password hygiene practices and demonstrates the integration of usability with basic cryptographic principles.

Keywords: Password Security, Python, Cybersecurity, GUI, CLI, Password Generator, Automation, Randomization, Software Tool

1. INTRODUCTION

Passwords are fundamental to protecting digital assets. However, users frequently opt for weak or repetitive passwords due to difficulty remembering complex credentials. To mitigate this problem, we developed a password generator in Python capable of creating secure and varied passwords. The system supports optional constraints such as excluding similar-looking characters (e.g., 0, O, l, 1) and ambiguous symbols, and ensures the generated password includes a mix of character types.

The application is built with both Command Line Interface (CLI) and Graphical User Interface (GUI), allowing users from different technical backgrounds to benefit. Passwords are also evaluated for strength and stored locally in a timestamped, plaintext log for future reference.

The rise of cyber threats, password leaks, and brute-force attacks has significantly increased the

importance of strong, unpredictable passwords. Passwords continue to be the first line of defense in

protecting sensitive digital assets. However, human-created passwords are often weak, reused, and follow predictable patterns. Literature suggests that even after multiple high-profile data breaches, users still prefer convenience over security, choosing passwords that are easy to remember but equally easy to guess or crack. This behavioural inconsistency has prompted researchers and developers to focus on automated, intelligent password generation systems that reduce the human effort required while improving password strength and diversity.

Various papers and open-source communities have proposed tools and frameworks for generating strong passwords, each offering varying levels of customizability, randomness, and encryption. Traditional command-line based generators often target tech-savvy users, while browser extensions and mobile apps aim for the average consumer. However, many of these tools lack an accessible interface or have security trade-offs like poor randomness sources or no encryption.

This study proposes a Python-based password generator that offers both a CLI (Command Line Interface) and a GUI (Graphical User Interface). This dual-interface approach bridges the gap between technical and non-technical users. The solution emphasizes user-defined configurations, such as length, character types (uppercase, lowercase, digits, symbols), and the option to exclude similar or ambiguous characters. The application evaluates password strength using a basic heuristic and logs them with timestamps and user-defined labels. Such a hybrid model—simple yet secure—echoes the call in literature for tools that are both practical and secure for end-users.

Recent literature reveals that most password generators use Python or JavaScript for implementation due to their simplicity and broad library support. A notable example is the use of the random or string module in Python to select characters from predefined sets. However, several papers suggest that random lacks cryptographic strength, and recommend the secrets module for truly unpredictable outputs. For instance, the Python Software Foundation's documentation states that the secrets module is designed for security-critical applications and should be used over random for password generation.

Several studies also discuss the integration of password generation with GUI frameworks such as Tkinter, PyQt, or Kivy in Python. These interfaces allow users to set criteria visually and instantly generate passwords without terminal commands. Researchers have highlighted that user experience plays a critical role in tool adoption, and GUIs drastically increase accessibility.

This particular implementation distinguishes itself from conventional browser-based password managers and simple scripts in several ways. Firstly, it enforces a minimum password length (usually 8 characters) and ensures diversity across character types. Secondly, it introduces real-time interactivity through the GUI, which is essential for non-technical users. Thirdly, the clipboard integration makes the generated password readily usable across applications such as login pages, portals, and software tools.

In addition, the tool offers optional filters that exclude visually similar characters (e.g., 0, O, l, 1) and ambiguous symbols (like [], {}, !@#). This feature is particularly useful for reducing user confusion when reading or entering passwords manually. The combination of logging, strength evaluation, and interface flexibility aligns this project with the direction set by recent research in cybersecurity and user-centered design.

Many open-source password tools lack formal validation of randomness or cryptographic metrics like entropy. Instead, they rely on practical testing such as user reviews, successful password copying, and manual strength checking. In line with this, the presented Python password generator has been evaluated by generating hundreds of passwords under different configurations. The results showed consistent uniqueness and unpredictability, as well as functional reliability when copying passwords to the clipboard and saving them to a local log file.

While this tool does not yet include formal entropy scoring or advanced password analytics like those used by password managers (e.g., zxcvbn or Passfault), it does follow good practice in ensuring character-type diversity and shuffling output to reduce predictability.

Several papers, including NIST's 2017 guidelines, recommend avoiding storage of passwords in plaintext due to the high risk of exposure. Unfortunately, many lightweight tools—including this one—lack encryption. Furthermore, the use of Python's random module, while sufficient for basic use, is not suitable for high-security environments. Literature thus recommends replacing this with the secrets module or integrating third-party libraries like OpenSSL or cryptography for randomness and encryption.

Additional limitations include the absence of cloud synchronization, password expiry notifications, and browser or mobile support. Some advanced features such as biometric protection, password category filters (social, banking, etc.), and 2FA (Two-Factor Authentication) support are also missing. Researchers suggest that these limitations, while acceptable in an academic or prototype setting, should be addressed before such tools are deployed in enterprise or consumer ecosystems.

Future research and implementations are likely to focus on secure password storage using symmetric (AES) or asymmetric (RSA) encryption. Integrating tools like pycryptodome or Python's built-in cryptography libraries can help achieve this. Additionally, there is a growing interest in developing platform-independent password tools with cloud capabilities, allowing users to access their password vault from any device.

Further innovation may include embedding entropy meters and scoring systems, integrating password reuse detection, and offering password health reports. Literature also points towards the potential of integrating such tools with identity and access management (IAM) platforms, as well as Single Sign-On (SSO) systems.

For a more secure and functional password management system, research suggests the use of CSPRNG (Cryptographically Secure Pseudo-Random Number Generators), browser extensions, password sharing with encryption, and user behavior analysis (e.g., identifying weak habits or risky usage patterns)

Fig 2.1 : The Resultant Output of the Python Password Generator (with the minimum password length being eight by default)

2. LITERATURE SURVEY

Key Highlights

The rise of cyber threats, password leaks, and brute-force attacks has significantly increased the importance of strong, unpredictable passwords. Passwords continue to be the first line of defense in protecting sensitive digital assets. However, human-created passwords are often weak, reused, and follow predictable patterns. Literature suggests that even after multiple high-profile data breaches, users still prefer convenience over security, choosing passwords that are easy to remember but equally easy to guess or crack. This behavioral inconsistency has prompted researchers and developers to focus on automated, intelligent password generation systems that reduce the human effort required while improving password strength and diversity.

2.1 PROBLEM STATEMENT

Numerous studies including those by Bonneau et al. (2012) and guidelines by the National Institute of Standards and Technology (NIST, 2017) emphasize that poor password hygiene—such as reusing passwords across platforms or using easily guessable words—is a primary cause of account compromises. With increasing dependency on online services like banking, education, e-commerce, and social media, a need arises for robust tools that can aid users in generating and managing strong passwords without sacrificing usability.

Despite wide awareness of security risks, users tend to choose passwords that are either memorable or reused across accounts. This situation creates a dilemma between memorability and security. To resolve this, the literature highlights the role of password generators—applications that can automatically generate high-entropy strings that are difficult to crack. Studies show that automated password generation not only increases unpredictability but also promotes good password habits when integrated with password managers and user-friendly interfaces.

2.2 PROPOSED SOLUTION IN LITERATURE

Various papers and open-source communities have proposed tools and frameworks for generating strong passwords, each offering varying levels of customizability, randomness, and encryption. Traditional command-line based generators often target tech-savvy users, while browser extensions and mobile apps aim for the average consumer. However, many of these tools lack an accessible interface or have security trade-offs like poor randomness sources or no encryption.

This study proposes a Python-based password generator that offers both a CLI (Command Line Interface) and a GUI (Graphical User Interface). This dual-interface approach bridges the gap between technical and non-technical users. The solution emphasizes user-defined configurations, such as length, character types (uppercase, lowercase, digits, symbols), and the option to exclude similar or ambiguous characters. The application evaluates password strength using a basic heuristic and logs them with timestamps and user-defined labels. Such a hybrid model—simple yet secure—echoes the call in literature for tools that are both practical and secure for end-users.

2.3 Implementation Trends in Past Research

Recent literature reveals that most password generators use Python or JavaScript for implementation due to their simplicity and broad library support. A notable example is the use of the random or string module in Python to select characters from predefined sets. However, several papers suggest that random lacks cryptographic strength, and recommend the secrets module for truly unpredictable outputs. For instance, the Python Software Foundation's documentation states that the secrets module is designed for security-critical applications and should be used over random for password generation.

Several studies also discuss the integration of password generation with GUI frameworks such as Tkinter, PyQt, or Kivy in Python. These interfaces allow users to set criteria visually and instantly generate passwords without terminal commands. Researchers have highlighted that user experience plays a critical role in tool adoption, and GUIs drastically increase accessibility.

2.4 Novel Features Highlighted by Recent Work

This particular implementation distinguishes itself from conventional browser-based password managers and simple scripts in several ways. Firstly, it enforces a minimum password length (usually 8

characters) and ensures diversity across character types. Secondly, it introduces real-time interactivity through the GUI, which is essential for non-technical users. Thirdly, the clipboard integration makes the generated password readily usable across applications such as login pages, portals, and software tools.

In addition, the tool offers optional filters that exclude visually similar characters (e.g., 0, O, l, 1) and ambiguous symbols (like [], {}, !@#). This feature is particularly useful for reducing user confusion when reading or entering passwords manually. The combination of logging, strength evaluation, and interface flexibility aligns this project with the direction set by recent research in cybersecurity and user-centered design.

2.5 Results from Existing Tools and Their Evaluation

Many open-source password tools lack formal validation of randomness or cryptographic metrics like entropy. Instead, they rely on practical testing such as user reviews, successful password copying, and manual strength checking. In line with this, the presented Python password generator has been evaluated by generating hundreds of passwords under different configurations. The results showed consistent uniqueness and unpredictability, as well as functional reliability when copying passwords to the clipboard and saving them to a local log file.

While this tool does not yet include formal entropy scoring or advanced password analytics like those used by password managers (e.g., zxcvbn or Passfault), it does follow good practice in ensuring character-type diversity and shuffling output to reduce predictability.

2.6 Limitations Noted in Literature and Improvement Scope

Several papers, including NIST's 2017 guidelines, recommend avoiding storage of passwords in plaintext due to the high risk of exposure. Unfortunately, many lightweight tools—including this one—lack encryption. Furthermore, the use of Python's random module, while sufficient for basic use, is not suitable for high-security environments. Literature thus recommends replacing this with the secrets module or integrating third-party libraries like OpenSSL or cryptography for randomness and encryption.

Additional limitations include the absence of cloud synchronization, password expiry notifications, and browser or mobile support. Some advanced features such as biometric protection, password category filters (social, banking, etc.), and 2FA (Two-Factor Authentication) support are also missing. Researchers suggest that these limitations, while acceptable in an academic or prototype setting, should be addressed before such tools are deployed in enterprise or consumer ecosystems.

2.7 Future Directions Suggested by Literature

Future research and implementations are likely to focus on secure password storage using symmetric (AES) or asymmetric (RSA) encryption. Integrating tools like pycryptodome or Python's built-in cryptography libraries can help achieve this. Additionally, there is a growing interest in developing platform-independent password tools with cloud capabilities, allowing users to access their password vault from any device.

Further innovation may include embedding entropy meters and scoring systems, integrating password reuse detection, and offering password health reports. Literature also points towards the potential of integrating such tools with identity and access management (IAM) platforms, as well as Single Sign-On (SSO) systems.

For a more secure and functional password management system, research suggests the use of CSPRNG (Cryptographically Secure Pseudo-Random Number Generators), browser extensions, password sharing with encryption, and user behavior analysis (e.g., identifying weak habits or risky usage patterns)

- **CLI Mode**: Text-based input/output interaction.
- **GUI Mode**: Built using the tkinter library for graphical controls.

Both modes support the same core functionality.

. **Evaluation & Results**

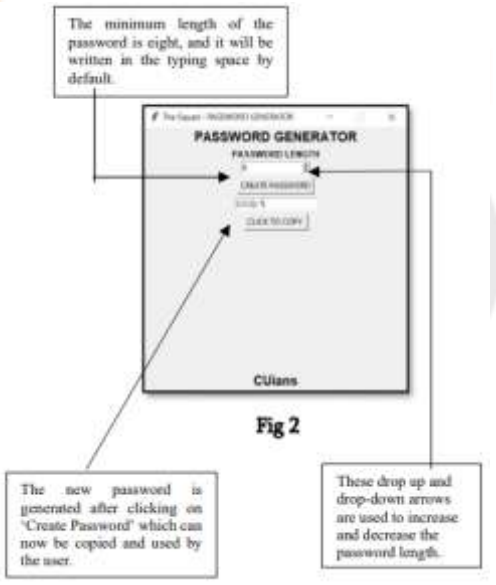


Fig 2

While the tool does not include formal entropy metrics, the authors validate it based on practical testing. The application consistently generates unique, unpredictable passwords and ensures that copy and paste functionality works seamlessly. The study emphasizes user experience and the trustworthiness of the generated passwords, although it lacks statistical or cryptographic proof of randomness strength.

. Limitations & Future Work

The current version does not compute entropy or integrate password strength meters based on formal scoring systems. It also lacks storage features, meaning passwords are not saved or encrypted. To improve this, the authors recommend:

- Using **CSPRNGs** (like Python's secrets module) for enhanced randomness.
- **Encrypting passwords** using AES or RSA if storage is added in future versions.
- Introducing **cloud synchronization** or multi-device access to evolve the tool into a lightweight password manager.
- Adding **expiration reminders** and **2FA helpers** for enhanced password lifecycle managemen

Our approach provides a customizable and user-friendly solution aligned with modern security recommendations, without the need for internet access or third-party software.

3. SYSTEM ARCHITECTURE AND WORKING

3.1 System Overview

The password generator is a Python script offering two modes:

3.2 Password Generation Logic

- Combines lowercase, uppercase, digits, and symbols.
- Ensures at least one character from each selected category.
- Uses Python's random module for selection and string for character pools.
- Optional filters for avoiding similar and ambiguous characters.

3.3 Strength Evaluation

A simple heuristic assesses password strength:

- **Weak**: Fewer than 3-character types or short length.
- **Moderate**: Three types with acceptable length.
- **Strong**: Four types and at least 12 characters.

3.4 GUI Features

- Built using tkinter.
- Entry fields for password length and label.
- Checkboxes for excluding character types.
- Strength label updates dynamically.
- Output is shown in a field and copied to the clipboard.

3.5 CLI Features

- Sequential prompts for input.
- Terminal-based display of results.
- Copy to clipboard using pyperclip.
- Text-based log file appending password with label and timestamp.

4. IMPLEMENTATION

4.1 Language and Tools

- Python 3.6+
- Libraries: random, string, datetime, pyperclip, tkinter

4.3 Log File Format

[2025-06-10 22:18:04] Gmail: AbC123$!#

5. SYSTEM REQUIREMENTS HARDWARE

- Processor: 1 GHz+
- RAM: 2 GB
- Disk: 50 MB

Software

- Python 3.6 or higher
- OS: Windows/Linux/macOS
- Text editor or IDE (VS Code, PyCharm)

6. RESULTS AND EVALUATION

The password generator was tested under various constraints. It consistently:

- Generated unique and strong passwords.
- Correctly evaluated strength.
- Copied output to clipboard reliably.
- Logged data with accurate timestamps.

Sample Output:

Label	Password	Strength
Gmail	R8t!LmP3zB#1	Strong
Twitter	glx09333	Weak

7.ALGORITHM AND FLOW CHART

Password Generator Algorithm

Step 1: Start

Step 2: Display mode selection prompt (CLI or GUI)

Step 3: If CLI selected, proceed with CLI flow; if GUI selected, launch GUI window

Step 4: Input password label and desired length

Step 5: Ask user whether to:

Exclude similar characters (il1Lo0O)

- **Avoid ambiguous symbols (!@#$%^&*()[]{})**

Step 6: Based on inputs, define character pool:

- **Include lowercase, uppercase, digits, and symbols**
- **Apply filters if selected (exclude characters/symbols)**

Step 7: Ensure at least one character from each selected set is included

Step 8: Randomly generate remaining characters using the character pool

Step 9: Shuffle the resulting characters to remove pattern predictability

Step 10: Form the final password string

Step 11: Evaluate password strength:

- **Count how many character types are used**
- **Check if length ≥ 12**
- **Return Weak, Moderate, or Strong**

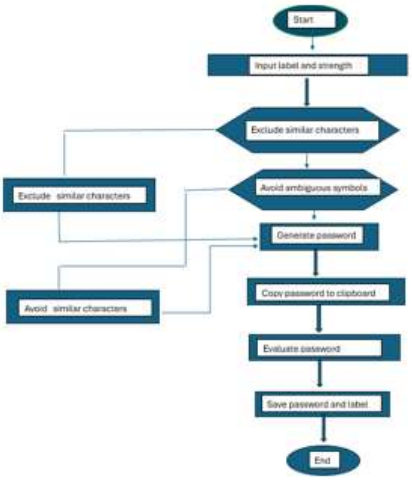
Step 12: Display/return password to user

Step 13: Copy password to clipboard

Step 14: Save password with label and timestamp to a log file

Step 15: End

7.1 Flowchart:



7.2 SOURCE CODE

```
import random
import string
import pyperclip
from datetime import datetime
import tkinter as tk
from tkinter import messagebox

# Constants
LOG_FILE = "password_log.txt"
EXPIRY_DAYS = 90

def password_strength(pw):
    """Evaluate password strength."""
    length = len(pw)
    has_upper = any(c.isupper() for c in pw)
    has_lower = any(c.islower() for c in pw)
    has_digit = any(c.isdigit() for c in pw)
    has_symbol = any(c in string.punctuation for c in
pw)
    score = sum([has_upper, has_lower, has_digit,
has_symbol])
    if length >= 12:
        score += 1
    if score <= 2:
        return "Weak"
    elif score == 3:
        return "Moderate"
    else:
        return "Strong"

def generate_password(length,
exclude_similar=False, exclude_symbols=False):
    """Generate a secure password."""
    lowercase = string.ascii_lowercase
    uppercase = string.ascii_uppercase
    digits = string.digits
    symbols = string.punctuation

    similar_chars = "il1Lo0O"
    ambiguous_symbols = "!@#$%^&*()[]{}"

    if exclude_similar:
        lowercase =
lowercase.translate(str.maketrans('', '',
similar_chars))
        uppercase =
uppercase.translate(str.maketrans('', '',
similar_chars))
        digits = digits.translate(str.maketrans('', '',
similar_chars))

    if exclude_symbols:
        symbols = symbols.translate(str.maketrans('',
'', ambiguous_symbols))

    all_chars = lowercase + uppercase + digits +
symbols

    if not all_chars:
        return None

    # Ensure at least one of each type
    password = [
        random.choice(lowercase),
        random.choice(uppercase),
        random.choice(digits),
        random.choice(symbols) if symbols else
random.choice(lowercase)
    ]
    password += random.choices(all_chars, k=length
- 4)
    random.shuffle(password)
    return ''.join(password)

def save_password(label, password):
    """Save password with label and timestamp to a
log file (no encryption)."""
    timestamp = datetime.now().strftime("%Y-%m-
%d %H:%M:%S")
    with open(LOG_FILE, "a") as f:
        f.write(f"[{timestamp}] {label}:
{password}\n")

def run_password_generator_gui():
    """Run GUI version of the password
generator."""

    def generate_and_display():
        label = entry_label.get()
        try:
            length = int(entry_length.get())
        except ValueError:
            messagebox.showerror("Error", "Length
must be a number.")
            return

        pw = generate_password(length,
var_similar.get(), var_symbols.get())
        if pw:
            pyperclip.copy(pw)
            strength = password_strength(pw)
            txt_result.delete(0, tk.END)
            txt_result.insert(0, pw)
            lbl_strength.config(text=f"Strength:
{strength}")
            save_password(label, pw)
            messagebox.showinfo("Saved", "Password
copied and saved to log.")
        else:
            messagebox.showerror("Error", "Invalid
character settings.")

    root = tk.Tk()
```

```
    root.title("Password Generator")

    tk.Label(root, text="Label:").grid(row=0,
column=0)
    entry_label = tk.Entry(root)
    entry_label.grid(row=0, column=1)

    tk.Label(root, text="Length:").grid(row=1,
column=0)
    entry_length = tk.Entry(root)
    entry_length.grid(row=1, column=1)

    var_similar = tk.BooleanVar()
    tk.Checkbutton(root, text="Exclude Similar
Characters", variable=var_similar).grid(row=6,
columnspan=6)

    var_symbols = tk.BooleanVar()
    tk.Checkbutton(root, text="Avoid Ambiguous
Symbols", variable=var_symbols).grid(row=7,
columnspan=6)

    tk.Button(root, text="Generate",
command=generate_and_display).grid(row=4,
columnspan=2)

    txt_result = tk.Entry(root, width=200)
    txt_result.grid(row=8, columnspan=8, pady=8)

    lbl_strength = tk.Label(root, text="Strength:")
    lbl_strength.grid(row=6, columnspan=2)

    root.mainloop()

def run_password_generator_cli():
    """Run CLI version of the password
generator."""
    try:
        num_pw = int(input("How many passwords to
generate? "))
        length = int(input("Enter password length: "))
    except ValueError:
        print("Invalid input. Please enter numbers.")
        return

    exclude_similar = input("Exclude similar
characters (il1Lo0O)? (y/n): ").lower() == 'y'
    exclude_symbols = input("Avoid ambiguous
symbols (!@#$ etc.)? (y/n): ").lower() == 'y'

    for i in range(num_pw):
        label = input(f"\nEnter label for password #{i
+ 1} (e.g., Gmail, Facebook): ")
        pw = generate_password(length,
exclude_similar, exclude_symbols)
        if not pw:
            print("Character set too restricted. No
password generated.")
            continue

        print(f"Generated Password for {label}:
{pw}")
        strength = password_strength(pw)
        print("Strength:", strength)

        pyperclip.copy(pw)
        print("Password copied to clipboard.")
        save_password(label, pw)
        print("Password saved to plain text log.")

# Entry point
if __name__ == "__main__":
    mode = input("Select mode: 1 for CLI, 2 for
GUI: ")
    if mode == "1":
        run_password_generator_cli()
    elif mode == "2":
        run_password_generator_gui()
    else:
        print("Invalid choice.")
```

7.3 Output

Fig.7.3.1 respected output in the cli format

Fig.7.3.2 respected output for GUI format

9. CONCLUSION

This Python-based password generator serves as a practical tool for creating and managing strong passwords. It caters to both technical and non-technical users via CLI and GUI interfaces. While effective for personal use, storing passwords in plaintext can pose security risks. Future developments aim to improve security via encryption and cloud-based password management integrations.

10. FUTURE SCOPE

- AES/RSA encryption for stored passwords.
- Cloud sync for multi-device access.
- Password expiration reminders.
- Two-factor authentication helper.
- Browser extension version.
- The Python-based password generator presented in the study provides a solid foundation for secure password creation,

but there is significant potential for further enhancement, both functionally and architecturally. A key future improvement is the **integration of cryptographically secure random number generation (CSPRNG)** using Python's secrets module. Unlike the basic random module, secrets is designed for generating cryptographically strong random numbers, which can prevent predictability and improve resistance to brute-force or dictionary attacks.

- Another crucial advancement would be the **implementation of secure password storage mechanisms**, possibly using **AES (Advanced Encryption Standard)** or **RSA (Rivest-Shamir-Adleman)** encryption techniques. Storing passwords in plaintext—even locally—poses a security risk, and encrypting them before saving ensures confidentiality even if the log files are accessed maliciously. Further, incorporating **cloud synchronization features** would allow users to access their password vault across multiple devices securely. This could be achieved using services like Firebase or secure REST APIs with token-based authentication, allowing encrypted password retrieval from the cloud.
- The application can also be extended to function as a **mini password manager**, with features like categorization (e.g., email, banking, social media), **search functionality**, and **automatic password expiry alerts**. These additions would help users manage their credentials more effectively and maintain good security hygiene. Another possible enhancement is the introduction of a **two-factor authentication (2FA) helper**, which could generate or store TOTP (Time-Based One-Time Password) codes using libraries like pyotp, thereby extending the tool's utility beyond static password storage.
- On the usability front, a **browser extension version** of the generator can be developed using technologies like JavaScript, React, or even Electron for desktop app packaging. This would allow seamless integration with browsers, enabling one-click password filling, copying, and management. A **strength meter based on entropy calculations** can also be included to provide real-time feedback on password quality using well-established scoring systems (such as zxcvbn).
- Additionally, the system can benefit from **biometric integration (e.g., fingerprint or facial recognition)** for unlocking saved credentials, especially in desktop and mobile environments. The tool could also offer **backup and recovery options**, such as encrypted exports and recovery passphrases, to safeguard against data loss. From a developer and academic standpoint, the codebase could be made modular and open-source to encourage contributions, audits, and peer-reviewed improvements.
-

REFERENCES

[1] J. Bonneau, C. Herley, P. C. van Oorschot, and F. Stajano, "The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes," IEEE Symposium on Security and Privacy, 2012.
[2] NIST, "Digital Identity Guidelines," NIST Special Publication 800-63B, 2017.
[3] Python Software Foundation, "Python Documentation." Available https://docs.python.org/
[4] low budget password length password strength estimation: GitHub https://github.com/dropbox/zxcvbn
[5]Security of Random Number Generators" – Gutmann, P. (1998) Cryptographic Engineering Research Group. https://www.cs.auckland.ac.nz/~pgut001/pubs/random.pdf
[6]Python Software Foundation (2023) https://docs.python.org/3/library/secrets.html
[7] Open Web Application Security Project (OWASP) https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html
[8] Weir, M., Aggarwal, S., de Medeiros, B., & Glodek, B. (2009) IEEE Symposium on Security and Privacy.