Project Report: Jackfruit - A Web-Based Encryption and Password Security Tool

SRN	Name	Section
PES2UG25EC008	Adithya N	A
PES2UG25EC006	Adarsh A Kulkarni	A
PES2UG25CS036	Aditya Kiran	A
PES2UG25CS028	Adithya Kadavath Umesh	A

Problem Statement

In the modern digital landscape, the secure handling of sensitive text and passwords is paramount. The problem addressed by this project is the need for a simple, accessible, and extensible web application that provides two core functionalities:

1. **Password Strength Assessment:** Allow users to check the security level of their passwords against common criteria.
2. **Text Encryption/Decryption:** Offer a secure way for users to encrypt and decrypt sensitive messages using multiple, user-definable ciphers,

including a basic one like ROT13 and a custom Vigenère-like shift cipher.

The solution must be built using Python and the Flask framework, ensuring user authentication and persistence of user-created ciphers.

Approach/Methodology/Data Structure Used

The project is structured as a Flask web application and relies on several key modules and data structures:

1. **Core Application Structure:**
    - **Flask Framework:** Used to handle routing, session management, and serving static files.
    - **Flask CORS:** Enabled for cross-origin resource sharing, typically for frontend interaction.
    - **Session Management:** `session` is used to maintain the logged-in user's state, securing user-specific key management.

2. **Password Security:**
    - **Password Strength Check (`strength_level` function):**
        - Uses **Regular Expressions (`re` module)** to efficiently check for the presence of uppercase, lowercase, digits, and special characters.
        - A simple scoring system is implemented: 1 point for each satisfied criterion (length ≥8, uppercase, lowercase, digit, special char). The score maps directly to the strength level (e.g., score 5 = "Very Strong").
    - **User Password Storage (`encrypt_password` function):**
        - A custom, illustrative encryption method is used for storing user passwords in `users.txt`.
        - **Data Structures:** A global list `charsList` (for the printable characters) and a dictionary `charsDict` (mapping characters to their index) are used to map characters to numerical values for encryption.

- **Algorithm:** It employs a basic substitution based on a list of **primes** (pre-calculated up to 100). Each character's index is shifted by a corresponding prime number from the list, modulo 100 (due to 100 printable characters).
  - $E_i = (\text{Index}(C_i) + \text{Prime}_i) \pmod{100}$

> 💡 **Note:**
> This custom encryption is used for storing user login passwords in this sample project. In a production environment, industry-standard hashing functions (like Argon2 or bcrypt) would be mandatory.

3. **User Management and Persistence:**
   - **User Data (`read_users`, `write_users`):** User accounts (username and encrypted password) are persisted in a local file, `users.txt`, using the **JSON** format for structured storage.
   - **Cipher Key Management** (`read_all_keys`, `write_all_keys`, `get_user_keys`, `save_user_keys`**):**
     - User-defined cipher keys are stored globally in `keys.txt`, also in **JSON** format, keyed by username.
     - **Data Structure:** Each user's keys are stored as a **List of Dictionaries**, where each dictionary represents a cipher with a `"name"` (e.g., "ROT13") and a `"pattern"` (e.g., "rot13" or "123").

4. **Encryption Ciphers:**
   - **ROT13 (`rot13` function):** Implements the classic ROT13 Caesar cipher.
     - **Algorithm:** Shifts alphabetic characters by 13 positions within their case (A-Z or a-z), using the modulo operator for wrap-around.
       - **$E = (\text{ord}(C) - \text{Base} + 13) \pmod{26} + \text{Base}$**
   - **Custom Shift Cipher (`shift_encrypt`, `shift_decrypt`):** A simplified Vigenère-like cipher where the shift values are taken sequentially from a user-defined pattern of digits (e.g., "123").

- **Algorithm**: Iterates through the message, shifting each alphabet character by a digit from the `pattern`. The pattern index wraps around using the modulo operator (`pi = (pi + 1) % len(digits)`).
- Encryption Shift: E = (ord(C) - Base + Shift) (mod26) + Base
- Decryption Shift: D = (ord(C) - Base - Shift) (mod26) + Base

---

Sample Input/Output

1. **Password Strength Check:**

Input Password	Output Strength
`password`	Weak
`Pass1234`	Strong
`P@sswOrd!23`	Very Strong

2. **Encryption/Decryption using ROT13**

Operation	Input Message	Key	Output
Encrypt	`Hello`	ROT13	`Uryyb`
Decrypt	`Uryyb`	ROT13	`Hello`

3. **Encryption/Decryption using Custom Shift Cipher**

Operation	Input Message	Key	Output
Encrypt	`ABCDEFG`	MyKey (123)	`BDFFHJI`
Decrypt	`BDFFHJI`	MyKey (123)	`ABCDEFG`

Explanation of Custom Shifts:

- A shifts by 1 → B
- B shifts by 2 → D
- C shifts by 3 → F

- D shifts by 1 (wraps around) → E

Challenges Faced

1. **Cipher Implementation Wrap-around:** A primary challenge was correctly implementing the wrap-around logic for the Caesar (ROT13) and Custom Shift ciphers. This required careful use of the modulo operator ((mod26)) to ensure that shifting 'Z' by 1 results in 'A', and 'A' by -1 (for decryption) results in 'Z'. This had to be applied separately for uppercase and lowercase letters.
2. **State Management:** Maintaining user-specific data (especially the custom keys) across different sessions and application restarts required robust implementation of file I/O operations (`read_all_keys` , `write_all_keys`) and reliable user session management using Flask's `session` object. Handling file corruption and ensuring default keys (like ROT13) always exist for a new user also added complexity.
3. **Password Storage (for the Project):** Devising an illustrative, yet not production-ready, password storage mechanism that was more complex than plain text but did not require external libraries for strong hashing (like bcrypt) was a design constraint. The custom character/prime-based shift was chosen as a teaching example.

Scope for Improvement

1. **Strong Password Hashing:** Replace the current custom `encrypt_password` function with an industry-standard hashing library (e.g., `passlib` using **Argon2** or **bcrypt**) for secure storage of user login credentials.
2. **More Robust Ciphers:** Implement more cryptographically secure and well-known ciphers, such as **AES** or **RSA**, which would require using the Python `cryptography` library.
3. **Key Management Security:** Instead of storing keys in plain text in `keys.txt` , encrypt the user's custom keys using a master key derived from the user's login password (Key Derivation Function - KDF), only decrypting them into memory upon successful login.
4. **Frontend/ UI Improvement:** Develop a dedicated frontend (e.g., using React or Vue) to replace the simple static file serving, providing a more interactive

and user-friendly experience, including client-side validation and visual feedback.

5. **Handling Non-Alphanumeric Characters:** The current ciphers only encrypt/decrypt alphabetic characters. They could be extended to handle numbers and special characters based on a user's chosen pattern or a larger character set map.