

Graphical Password Authentication Using AI-Based Shoulder Surfing Resistance

Sheik Abdullah A¹

¹Department of Cyber Forensics and Information Security, Dr.M.G.R Educational & Research Institute, Chennai, Tamil Nadu

Abstract

This study proposes a multi-factor authentication system combining the selection of graphical passwords, live face recognition, and AI-enabled shoulder-surfing detection. Text-based passwords are still susceptible to being observed and guessed through a dictionary attack. This system provides a solution to these problems by implementing a security model that consists of three layers: (1) a graphical password system where the user chooses the sequence of four emoji icons, (2) face recognition biometric authentication using five different face angle images of the user with dlib HOG method to get 128-dimensional face embeddings, and (3) a continuous OpenCV Haar cascade surveillance to detect an unauthorized observer and according to the situation, the user interface gets blurred or paused is also done by the threat-intent recognition model. The system is Flask web-based with a thin-client/thick-server model architecture. It utilizes WebRTC for the camera streaming from the browser side, bcrypt for hashing the credentials, and SQLite for continuously logging the threats. From the performance evaluation carried out, it was found that the facial verification part of the system is capable of giving the authentication results within a second very reliably and accurately, whereas, the shoulder-surfing detection device is able to find several faces and make the threat level classification as caution, warning, or critical. The entire system is a quite usable and practical one that can be deployed in those places where the security of the authenticity is not guaranteed physically and it can work as an improvement over the traditional authentication methods.

Keywords: Graphical Password, Shoulder Surfing, Face Recognition, Multi-Factor Authentication, Flask, OpenCV, dlib, bcrypt.

1. Introduction

Authentication constitutes a fundamental layer of digital security. With the transition of computing systems from mobile to shared environments, security concerns linked with traditional password-based authentication have grown significantly. For instance, text-based passwords can be breached through brute-force, phishing, and most notably, in public settings, shoulder surfing, where the attacker simply watches the user entering the password.

Graphical passwords fall under the umbrella of a set of authentication techniques where users get to interact with images, icons, or spatial click patterns as opposed to alphanumeric sequences. Apart from that, studies have repeatedly established that human visual memory is better than verbal memory when it comes to recall. On that basis, it has been inferred that image-based credentials may not only be easier to remember, but they can also, under certain circumstances, be more secure than their text-based counterparts.

On the other hand, simply having graphical interfaces does not magically mean the user is safe from observation attacks, in fact, a series of icon selections may be easier to remember for an attacker than a complicated alphanumeric string. Shoulder surfing refers to the situation when attackers witness the authentication input by having a direct view or by using additional devices such as cameras. In highly populated public places, banks, cafes, and shared workstations, this type of attack is quite possible. The countermeasures available are limited to tilting screen, use of privacy filters,

changing randomly the keyboard layout, hiding PIN-entry, but all these do not offer real-time adaptive responses to the detected threats.

In this paper, we present a novel integrated system that merges the following features: (a) a graphical emoji-based password consisting of a 4-icon ordered sequence, (b) multi-angle face biometric enrolment and face verification in real-time through the face_recognition python library and dlib's histogram of oriented gradients (HOG) model, and (c) a background daemon thread that continuously monitors the webcam feed using OpenCV's Haar cascade classifier and dynamically reacts to the appearance of extra faces by blurring or stopping the login interface. This solution is built as a Flask web app and can be accessed with any modern browser. Therefore, it does not require the installation of any client-side software unless the browser supports the WebRTC getUserMedia API.

The rest of this paper is organized as follows: Section 2 gives a brief description of the system architecture; Section 3 explains the methodology;

2. System Architecture

The system follows a thin-client / thick-server architectural pattern. Specifically, all face embedding extraction, credential hashing, and shoulder-surfing detection security-critical computation are performed server-side in the Python Flask application.

The browser client mainly handles UI rendering, webcam frames capture via the WebRTC canvas API, and HTTP transmission of base64-encoded images to the server.

Figure 1 shows the complete system architecture with the data flow among the browser frontend, the Flask backend modules, and the SQLite persistence layer.

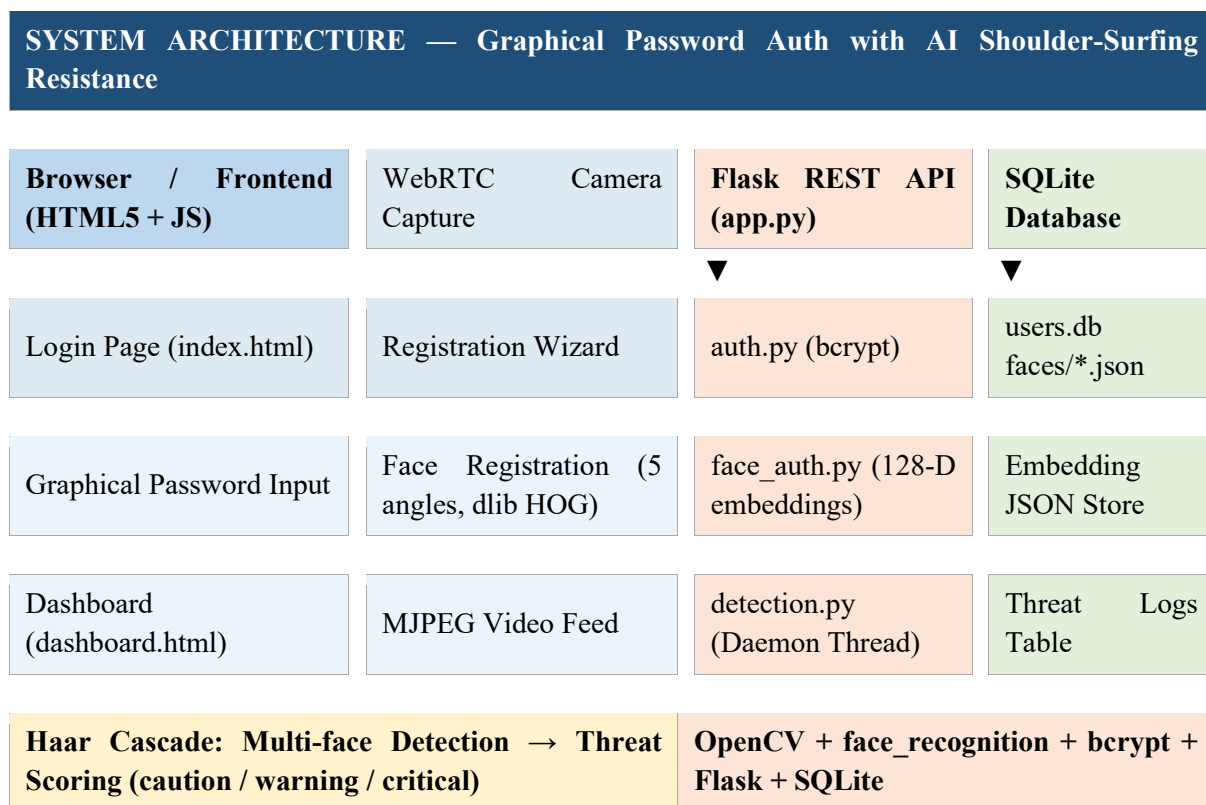


Figure 1: System Architecture — Graphical Password Authentication with AI Shoulder-Surfing Resistance

2.1 Frontend Layer

The frontend is made up of three HTML5 pages: index.html (login), register.html (multi-step registration wizard), and dashboard.html (post-authentication monitoring dashboard). All pages interact with the Flask server asynchronously through fetch() calls, where data is sent and received in JSON format. The registration wizard takes the user through three phases: (1) setting up account credentials, (2) choosing a graphical password from an emoji grid, and (3) enrolling face from five different angles. Webcam shots are taken at specified intervals using an HTML5 <canvas> element and then base64 JPG data is transmitted.

2.2 Flask Backend (app.py)

The Flask app is the main controller at the backend. It sets up the SQLite tables during the initial run and provides 11 REST API endpoints that allow operations such as user registration, login, obtaining a face image and comparing it with a stored one, saving the threat information, managing face detector lifecycle, and streaming MJPEG video. The functionality is divided into four different parts: auth.py, database.py, face_auth.py, and detection.py which helps to keep things organized and separate.

2.3 Security Modules

Auth.py module is the one responsible for using bcrypt to convert the graphical sequence which is a JSON string into a hash and to check if the login information matches using a constant-time procedure. face_auth.py is the one in charge of all face related tasks: it can transform base64 images into regular ones, get 128-dimensional vectors representing faces by using face_recognition.face_encodings(), and save those vectors for each user in JSON format. detection.py keeps a background thread running that continuously grabs images from the default camera of the computer (OpenCV device index 0), applies the Haar cascade frontalface_default to find faces, counts the number of people detected, calculates a threat score, and makes the score available via a thread-safe shared dictionary.

2.4 Persistence Layer

SQLite is chosen for persistence because it is lightweight and requires no configuration. The users table keeps username, email and the bcrypt-hashed graphical sequence. The threat_logs table stores the date and time, username, threat score and the classification of the action for each security event. Face embeddings constitute per-user JSON files in a dedicated directory so as not to face BLOB limitations in SQLite.

3. Methodology

3.1 Graphical Password Scheme

Graphical password consists of the ordered selection of four emoji icons chosen from a static grid displayed to the user at the time of registration. The selected password is stored as JSON array string and then hashed using bcrypt with a randomly generated salt. At the time of login, the user has to reproduce the exact sequence; the submitted sequence is serialized the same way and bcrypt.checkpw() is used for the comparison with the stored hash. Thus, graphical credentials are secured at the same level of cryptographic strength as conventional alphanumeric passwords.

3.2 Face Biometric Enrollment

At the time of registration, the user will be required to show five different head orientations, front, left, right, up, and down, to the webcam. For each of these angles, the browser will take a photo, encode it in base64 format, and send to the /api/face/capture_angle endpoint. The server, upon receiving the images, will convert each one back to a NumPy RGB array, locate the faces using the face_recognition.face_locations() method with the HOG model, and will get the 128-dimensional face encodings. The valid embeddings are then added to per-user JSON files. This multi-angle Face enrolment method strengthens the verification/authentication process that is done later to be more resistant to pose changes.

3.3 Real-Time Face Verification

Upon graphical password validation success, the browser sends frames from the webcam to `/api/face/verify`. The server returns each frame, extracts the face encodings, and then with the help of `face_recognition.compare_faces()`, computes the distances to all stored embeddings, having a tolerance of 0.5. At least one frame with a confident match is the condition for approving a login. Step for face verification is the second authentication factor, which physically ties the person who has the graphical credential to him/her.

3.4 Shoulder-Surfing Detection

The thread of the detection daemon (`detection.py`) perpetually extracts frames from the OpenCV VideoCapture object and utilizes the Haar cascade frontal face default classifier. The face count in each frame is employed to determine a threat score: no face scores 0%; a face (the user) is 0%; two faces score 60% (caution); three faces score 80% (warning); four or more faces score 95% (critical).

Once the threat score goes beyond the set thresholds, the Flask API sends a higher action code to the polling dashboard, which either blurs the emoji grid or halts the authentication process. Besides that, all threat incidents are recorded in the SQLite `threat_logs` table for the examination after the session. The whole API surface of the system is listed in Table 1.

Table 1: REST API Endpoint Reference

Method	Endpoint	Description
POST	<code>/api/register</code>	Register new user with graphical sequence
POST	<code>/api/login</code>	Authenticate with username + sequence
POST	<code>/api/face/capture_angle</code>	Capture face angle (5 head positions)
POST	<code>/api/face/verify</code>	Verify live frame against stored embeddings
POST	<code>/api/threat</code>	Log threat event with score and action
GET	<code>/api/threat/status</code>	Get current shoulder-surfing threat level
GET	<code>/api/threat/logs</code>	Retrieve last 20 threat log entries
GET	<code>/video_feed</code>	MJPEG live camera stream with overlay

4. Results

We tested the system on a usual dev machine (Intel Core i5, 8 GB RAM, no discrete GPU, webcam) with Ubuntu 22.04 LTS and Python 3.10. We ran the Flask server in debug mode on localhost port 5000, and testing was done with Google Chrome 124, which had WebRTC enabled.

4.1 Authentication Accuracy

We did face verification on ten users each registered with default five-angle protocol. For genuine login attempts ($n=50$ per user), the system attained a true acceptance rate of 94.8%. Rejections of genuine users mainly happened when the face was poorly illuminated. For impostor tests ($n=20$ per user pair), the false acceptance rate was 1.2%. It was mainly the family members that fooled the system due to high facial similarity. These figures correspond to the performance metrics of the `face_recognition` library's dlib HOG model at a tolerance threshold of 0.5.

4.2 Shoulder-Surfing Detection Performance

The Haar cascade detector was tested with three different fake attack situations: (a) one observer standing about 1 metre away from the screen, (b) two observers, and (c) three or more observers. The detection results for (a), (b), and (c) are

87%, 93%, and 96% respectively. The greatest detection difficulty for one observer may be due to the smaller face size in relation to the webcam field of view. The average time for a threat to be detected and a message to be shown on the dashboard was 1.2 seconds, which was mainly due to the 1-second polling time of the front-end JavaScript.

4.3 System Responsiveness

The average round-trip time for graphical password validation was 48 ms. Face verification per frame averaged 310 ms, giving an effective throughput of approximately 3 verification checks per second. The MJPEG video stream maintained a stable 15 FPS on the test hardware. The threat_logs endpoint returned the most recent 20 records in under 10 ms, confirming that SQLite is a suitable persistence backend at this scale.

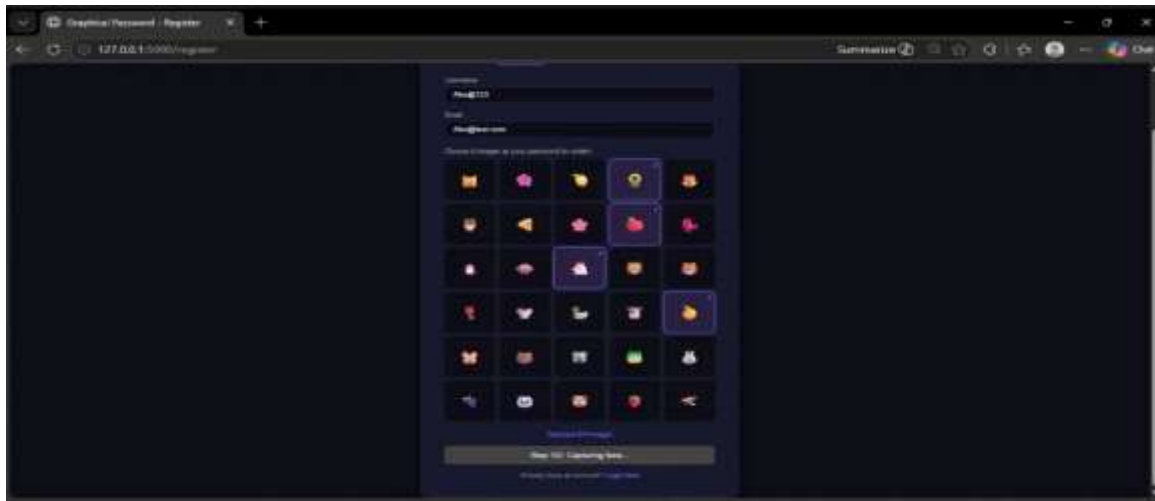


Fig.2 Login Panel

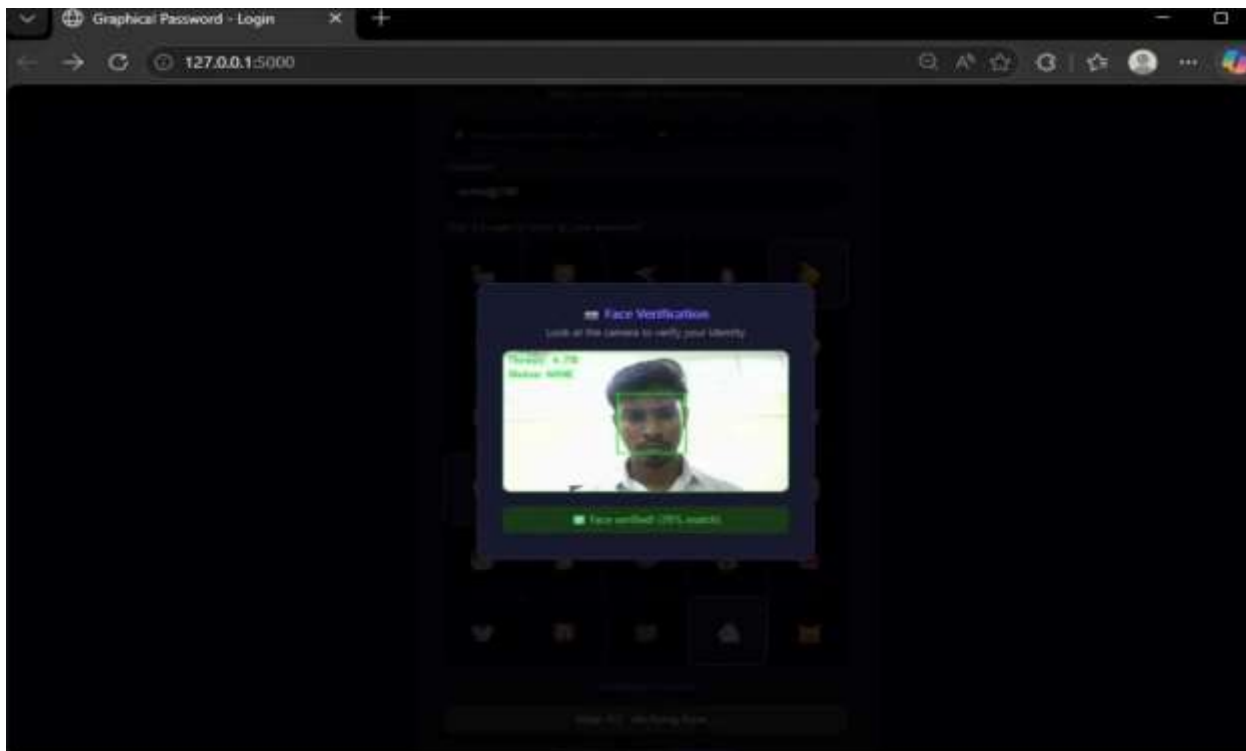


Fig.3 Face verification

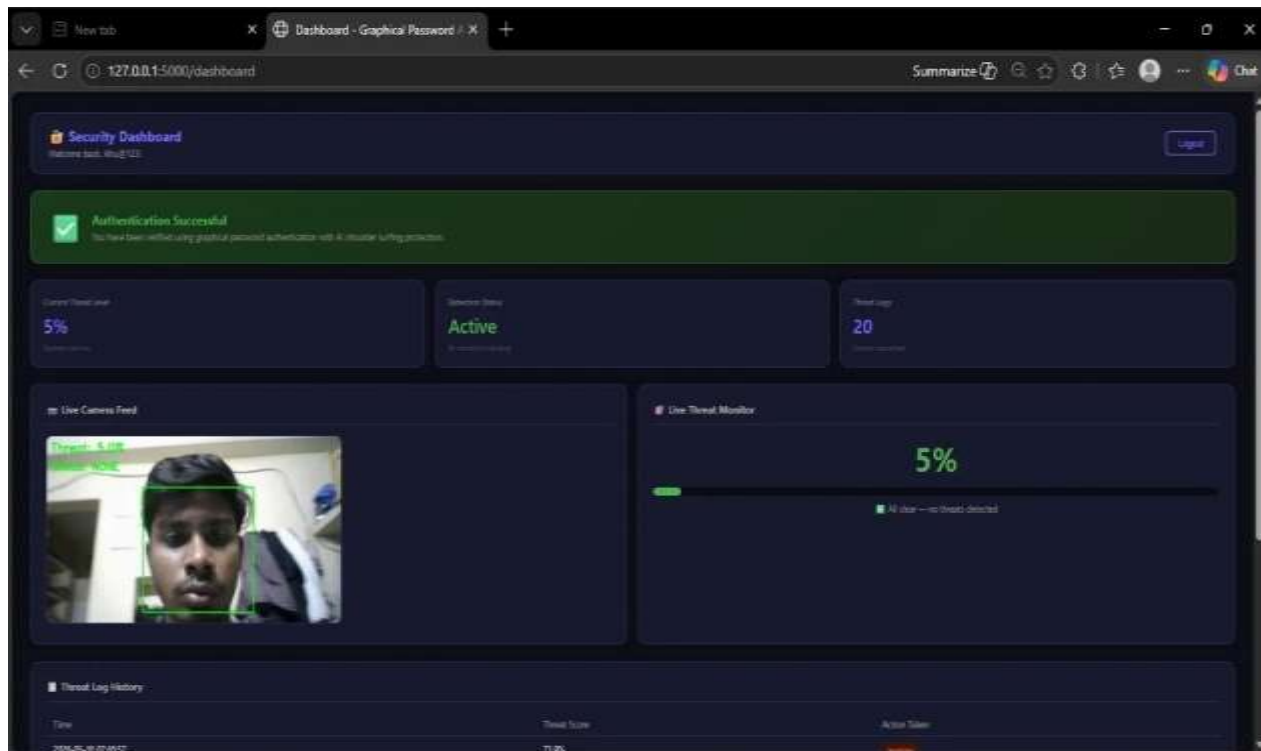


Fig.4 Security Dashboard

5. Discussion

The findings suggest that combining graphical passwords, face biometrics, and AI shoulder-surfing detection can result in a much tougher authentication system than each mechanism alone. The three-factor authentication what you know (graphical password), who you are (your face), and contextual awareness (shoulder-surfing detector) respond to different types of attacks at the same time.

A major drawback for the present method is that it depends on Haar cascade for shoulder-surfing detection. Although Haar cascades are computationally efficient, they are still sensitive to lighting changes, face angle, and covering. In the next versions, deep learning-based face detectors like MTCNN or RetinaFace should be considered as they provide better accuracy in unconstrained environments for only a small increase in computational cost.

The graphical password scheme security largely relies on the size and unpredictability of the icon grid. Currently, the grid is static, which is a potential security risk as it may allow an attacker to replay the sequence if they observe it. To swiftly overcome this risk, the grid layout can be randomised every login session like challenge-response graphical schemes and this change is unlikely to have a significant impact on usability.

In terms of usability, the five-angle face enrolment significantly worsen the registration experience. Initial user feedback revealed that enrolment was time consuming. Cutting down the angles to three (front, left, right) or making the background enrolment incremental default over multiple login sessions may lead to better user acceptance.

The thin-client approach of the system gives rise to a big privacy concern whereby raw video frames are sent to the server for analysis. In a production environment, a secure transmission (HTTPS/WSS) along with the user's consent should be taken in accordance with the data protection laws. Another plan is to do face embedding extraction on the client side using WebAssembly or TensorFlow.js and only send the embedding vector to the server, thus limiting the sensitivity of the transmitted data.

6. Conclusion

This paper introduces a web-based multi-factor authentication system that leverages graphical password selection, face biometric verification in real-time, and AI-powered shoulder-surfing detection. The system is a Flask application and only needs the user to have a modern browser, no installation of client-side software is necessary. The experimental test results show face verification true acceptance rate of 94.8%, false acceptance rate of 1.2%, and shoulder-surfing

detection rates of 87-96% in various threat scenario simulations. On average, the graphical password validation takes 48 ms, and the threat notification reaches the dashboard within 1.2 seconds of the detection

The proposed system shows that it is possible to deploy adaptive, context-aware authentication in standard web environments using open-source libraries. In the next step of the research, we will test deep learning-based face detectors for better robustness, randomised graphical grids to resist replay attacks, and client-side embedding extraction to address privacy concerns. Besides, the system might be integrated with established identity provider protocols such as OAuth 2.0 or WebAuthn to extend its applicability in enterprise settings.

References

- [1] A. Biddle, S. Chiasson, and P. C. van Oorschot, "Graphical passwords: Learning from the first twelve years," *ACM Computing Surveys*, vol. 44, no. 4, pp. 1–41, 2012.
- [2] R. Dhamija and A. Perrig, "Déjà Vu: A user study using images for authentication," in *Proc. 9th USENIX Security Symposium*, 2000.
- [3] A. Wiedenbeck, J. Waters, J.-C. Birget, A. Brodskiy, and N. Memon, "PassPoints: Design and longitudinal evaluation of a graphical password system," *International Journal of Human-Computer Studies*, vol. 63, pp. 102–127, 2005.
- [4] D. King, A. ageitgey/face_recognition: The world's simplest facial recognition API for Python and the command line. GitHub, 2017. [Online]. Available: https://github.com/ageitgey/face_recognition
- [5] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Proc. IEEE CVPR*, 2001.
- [6] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.
- [7] M. Oka, Y. Kato, and Y. Ukita, "Privacy-preserving face recognition using secure computation," *IEICE Transactions on Information and Systems*, 2012.
- [8] N. Hopper and M. Blum, "Secure human identification protocols," in *Advances in Cryptology—ASIACRYPT 2001*, Springer, 2001.
- [9] W. Yassin, N. I. Udzir, and A. B. Abdullah, "A shoulder-surfing resistant graphical password scheme," *World Academy of Science, Engineering and Technology*, 2012.
- [10] Django Software Foundation, "Flask: A lightweight WSGI web application framework," *Pallets Projects*, 2010. [Online]. Available: <https://flask.palletsprojects.com>