# SE project notes

**Core Use-Cases (What your software *does*)**

1. **Secure Login System**
   * User opens app → Face is scanned (via webcam) → Voice is recorded (say a passphrase).
   * Both biometrics are verified → If matched, app decrypts user’s stored profile & grants access.
2. **Biometric Enrollment (Registration)**
   * New user registers → Capture multiple face images + voice samples.
   * Extract features → Encrypt & store securely in database.
3. **Multi-Factor Authentication**
   * Combine **face + voice** before granting access.
   * Even if one factor fails (e.g., poor voice recording), cryptography ensures added security.
4. **Data Security Module**
   * Store biometric data in encrypted form using AES/RSA.
   * Allow optional file/folder encryption for user’s sensitive data.

**🔹 2. MVP Features (Minimal but Functional)**

✅ User registration (face + voice samples).  
✅ Secure login using biometrics.  
✅ Encryption of biometric templates.  
✅ Simple database (SQLite/MySQL).  
✅ Admin dashboard to manage users.

**🔹 3. Software Flow (User Journey)**

1. **Home Screen**
   * Options: "Register" | "Login" | "Admin"
2. **Registration Page**
   * Capture live face (camera preview).
   * Record voice sample (with playback option).
   * Store encrypted features in DB.
3. **Login Page**
   * Show webcam feed (detect & verify face).
   * Ask user to speak passphrase (verify voice).
   * If both pass → Decrypt & unlock access.
4. **Dashboard (Post-Login)**
   * Show user profile.
   * Optionally allow file encryption/decryption using biometrics.

**🔹 4. How the UI Should Look (Simple & Modern)**

* **Minimalist & clean interface** (like authentication apps).
* Use **tabs or side menu**:
  + Register
  + Login
  + Settings
* **Visual Elements:**
  + Webcam preview window (for face capture).
  + Mic recording button (for voice).
  + Status messages: “Face detected ✅”, “Voice matched ✅”, “Access Granted 🔓”.
* **Optional:** Dark mode for a security-themed look.

**🔹 5. Expansion Ideas (Future Phases)**

* Mobile app version (Android/iOS).
* Liveness detection (prevent spoofing with photos/recorded voices).
* Cloud storage of encrypted biometric data.
* Integration with smart cards or USB keys.

**Biometric Template**

When your software captures a **face** or a **voice**, it doesn’t usually store the raw photo or raw audio (that would be insecure and heavy).  
Instead, it **extracts key features** (like unique face points, or voice frequency patterns) and represents them in a compact digital form (like a vector of numbers).

👉 That compact digital representation is called a **biometric template**.

* For face → it could be a **128-dimensional feature vector**.
* For voice → it could be **MFCC features** (Mel Frequency Cepstral Coefficients).

So instead of storing someone’s actual face/voice, you store this "mathematical fingerprint."

**🔹 Encryption of Biometric Templates**

Now, if you just store those templates in a database, a hacker could steal them and misuse them.  
That’s where **encryption** comes in:

* Before saving the template, you **encrypt it with a cryptographic algorithm** (like AES).
* When verifying identity, the software **decrypts the template**, compares it with the new captured template, then discards it.

This way:  
✅ Even if the database is hacked, the attacker only gets **encrypted gibberish**, not real biometric data.  
✅ It ensures **privacy + security**, since biometric data cannot be changed like a password.

**🔹 Simple Example**

1. Extract face template:
2. [0.123, 0.456, 0.789, ...]
3. Convert into a string/bytes.
4. Encrypt using AES:
5. u89h2@#jda92!lkd... (ciphertext)
6. Store in DB.
7. During login, capture face again → extract template → decrypt stored one → compare.

👉 In short:  
**Encryption of biometric templates = locking the "digital fingerprint" of your face/voice with cryptography before saving it, so only authorized software can unlock and compare it.**

**Phase 1 Implementation Roadmap**

**1. Finalize Requirements (Write Them Clearly)**

👉 Your first deliverable is a **requirements document**. Write down:

* **Functional requirements**
  + Face recognition (capture, feature extraction, matching).
  + Voice recognition (capture, feature extraction, matching).
  + Cryptographic protection (AES/RSA encryption of biometric templates).
  + User registration (store encrypted data).
  + User login (verify face + voice).
* **Non-functional requirements**
  + Accuracy ≥ 90%.
  + Real-time authentication (under 3 seconds).
  + Data security (encrypted storage).
  + User-friendly UI (basic, minimal).

✅ By the end of this step: You’ll have a **written requirements analysis**.

**2. Choose Tools & Tech Stack**

* **Programming:** Python 3.9+
* **Libraries:**
  + opencv-python (face capture + detection)
  + face-recognition (feature extraction & matching)
  + speechrecognition + librosa (voice feature extraction)
  + cryptography (AES/RSA encryption)
* **Database:** SQLite (start simple)
* **UI:** Tkinter or PyQt5 (optional for Phase 1, but good for MVP)

✅ By the end of this step: You’ll have installed all required libraries.

**3. Setup Development Environment**

Run these commands:

python -m venv venv

source venv/bin/activate # Linux/Mac

venv\Scripts\activate # Windows

pip install opencv-python face-recognition speechrecognition librosa cryptography

✅ By the end of this step: You’ll have a working Python environment.

**4. Prototype First Modules (Minimal Tests)**

* **Face Capture & Encoding Test:** Use webcam to capture your face and generate a **face encoding vector**.
* **Voice Capture & Feature Test:** Record voice and extract MFCC features.
* **Encryption Test:** Encrypt and decrypt a random string using AES.

👉 These are **mini test scripts** to confirm each module works before combining.

**5. Document Phase 1**

Your **Phase 1 report** should include:

1. **Introduction & Objective** (what your project aims to solve).
2. **Functional & Non-Functional Requirements** (clear tables).
3. **Tools & Libraries** (with justification).
4. **System Architecture (High-level diagram):**
   * Input (Camera + Mic) → Feature Extraction → Encryption → Storage/Verification.

**Libraries**

1. . **OpenCV (Face Capture & Detection)**
2. 2. **face-recognition (Face Feature Extraction & Matching)**
3. **speechrecognition (Voice Input)**
4. librosa (Voice Feature Extraction – MFCCs)
5. cryptography (AES/RSA Encryption)
6. pyaudio(Extra)
7. soundfile(Extra)

**Face Recognition Concepts**

1. **Computer Vision**:
   * It’s the field of making computers “see” and understand images or video.
   * In Python, OpenCV is the most common library. It allows you to detect faces, read images, process video, etc.
2. **Face Detection vs Face Recognition**:
   * **Detection**: Finding a face in an image. Think of it as “drawing a box around any face.”
   * **Recognition**: Identifying whose face it is. For this, we compare a detected face with stored faces.
3. **Face Encoding**:
   * A face is converted into a numerical representation (vector) using a **deep learning model**.
   * These vectors are stored in a database. Later, when a user tries to authenticate, we encode their live face and compare it with stored vectors.
4. **Python Libraries for Face Recognition**:
   * **OpenCV**: For image/video processing.
   * **face\_recognition**: Built on top of dlib, simpler to use for face detection + recognition.
   * **Mediapipe**: Alternative to dlib for face landmarks and detection.

**Step 1: Face Recognition Setup**

**Concepts First**

1. **Face Detection vs Recognition**
   * **Detection**: Find faces in an image/video. Draw boxes around them.
   * **Recognition**: Identify *whose* face it is by comparing with stored face data.
2. **Face Encoding**
   * Each face is converted to a **vector of numbers** representing facial features.
   * This vector is stored securely. Later, a live face is encoded and compared to stored vectors.
3. **Libraries**
   * opencv-python: Handles image/video processing.
   * face\_recognition: Simplifies face detection and encoding.

It’s easier than dlib directly and avoids your previous installation issues.

VoiceFaceCryptoProject/ <-- Root project folder

│

├── venv/ <-- Virtual environment (ignored in Git)

│

├── data/ <-- Folder to store face/voice data

│ ├── face\_encodings/ <-- Encoded face vectors (encrypted files)

│ └── voice\_samples/ <-- Optional: stored voice samples

│

├── scripts/ <-- All Python scripts

│ ├── face\_recognition\_module.py

│ ├── voice\_recognition\_module.py

│ ├── encryption\_module.py

│ └── main.py <-- Main program integrating modules

│

├── requirements.txt <-- List of Python dependencies

├── .gitignore <-- Files/folders Git should ignore

└── README.md <-- Project overview & instructions

1. **face\_recognition-module.py**
2. We have used **while True** because the frames needs to be captured continuously , so that the video can be streamed , frame by frame .

* ret, frame = video\_capture.read()
* video\_capture.read() is a method from OpenCV’s cv2.VideoCapture class.
* It tries to grab the next frame from the webcam (or video file).
* It returns two values:
* ret (short for return value):
* A boolean (True or False).
* True means a frame was successfully captured.
* False means the capture failed (for example, no camera connected, end of video file, or permission issues).
* frame:
* The actual image data (a NumPy array).
* Shape is typically (height, width, 3) → because it’s a color image with 3 channels (Blue, Green, Red).

1. **# Convert BGR (OpenCV default) to RGB (face\_recognition expects RGB)**

**rgb\_frame = cv2.cvtColor(frame, cv2.COLOR\_BGR2RGB)** # Convert BGR (OpenCV) to RGB

1. **face\_locations = face\_recognition.face\_locations(rgb\_frame)**

* Detects all faces in the rgb\_frame.
* Returns a list of coordinates for each face: (top, right, bottom, left).
* These coordinates tell you where the face is located in the image.

1. for (top, right, bottom, left) in face\_locations:

* cv2.rectangle(frame, (left, top), (right, bottom), (0, 255, 0), 2) # Draw rectangle
* Loops through all detected faces.
* cv2.rectangle draws a rectangle around each face:
* (left, top) → top-left corner of rectangle.
* (right, bottom) → bottom-right corner.
* (0, 255, 0) → color of rectangle (Green in BGR format).
* 2 → thickness of rectangle lines.

1. key = cv2.waitKey(1) & 0xFF

* cv2.waitKey(1) waits for 1 millisecond for a key press.
* & 0xFF ensures compatibility across different systems (like Windows, Linux).
* key stores the ASCII code of the key pressed.

1. face\_encoding = face\_recognition.face\_encodings(rgb\_frame, face\_locations)[0]

* Converts the first detected face into a 128-dimensional numerical vector.
* This vector represents unique facial features and is used for recognition.

1. if key == ord('s') and face\_locations: # Press 's' to save encoding

* Checks if the user pressed 's' and a face was detected.
* ord('s') converts the character 's' to its ASCII code.

1. filename = os.path.join(ENCODINGS\_DIR, "user\_face.npy")

* Creates the path where the face encoding will be saved.
* os.path.join ensures it works on all operating systems.

1. np.save(filename, face\_encoding) # Save encoding as .npy file

* Saves the face encoding as a .npy file using NumPy.
* .npy files store arrays efficiently and can be loaded later with np.load.

1. video\_capture.release()

* cv2.destroyAllWindows()
* release() → Frees the webcam so other programs can use it.
* destroyAllWindows() → Closes all OpenCV windows safely.

For encryption

**Quick concept primer (simple words)**

* **Symmetric encryption**: the same secret (a key) is used to encrypt and decrypt. Keep the key secret.
* **Fernet**: provides *authenticated* symmetric encryption (encryption + integrity check). Good for our use: encrypting face encodings.
* **Key storage**: do **NOT** commit the key file to Git. Put it in .gitignore or use environment variables / OS key vault for production.
* **Why encrypt encodings?** Face encodings are sensitive biometric data — encrypting them protects user privacy and prevents misuse if files are leaked.

**Security notes & best practices (must-read)**

* **Never** commit secret.key to version control. Add it to .gitignore.
* For a real product use secure key storage (KMS, OS keyring, or environment secrets). Don’t store keys in project root.
* Consider rotating keys periodically and re-encrypting stored data.
* Fernet includes integrity checking. If encrypted bytes are tampered with, fernet.decrypt() will raise an exception.

**What is Fernet?**

* **Fernet** is a system for **symmetric encryption** provided by the Python cryptography library.
* **Symmetric** means: the *same key* is used to both encrypt and decrypt.
* It is designed to be **safe by default**, so you don’t need to worry about low-level crypto mistakes.

**2. What does “authenticated encryption” mean?**

Normally, when people think of encryption, they think only of *hiding the data*.  
But encryption **alone** is not enough. Why?

* Imagine someone intercepts your encrypted file and flips a few random bits.
* When you later decrypt it, you’d get garbage, and you might not even realize the file was tampered with.
* This is where **authentication** comes in.

🔹 **Authenticated encryption** ensures two things:

1. **Confidentiality** → nobody without the key can read the message.
2. **Integrity** → if anyone changes even *one bit* of the ciphertext, decryption fails with an error.

So Fernet doesn’t just hide your data — it also protects it from tampering.

**3. What does Fernet actually do under the hood?**

You don’t need to memorize this, but it’s good to know:

* Fernet uses **AES** (Advanced Encryption Standard) in **CBC mode** for encryption.
* It uses **HMAC (SHA256)** for authentication (integrity check).
* It automatically handles things like:
  + generating random initialization vectors (IVs)
  + timestamping the message (so you can set expiry if you want)
  + combining everything into a single token

**4. What does a Fernet “token” look like?**

When you encrypt something with Fernet, the output is **Base64 text** that includes:

* Version
* Timestamp
* IV (random bits for randomness)
* Ciphertext (encrypted data)
* HMAC (integrity check)

Example:

gAAAAABlZ7Bz5Jm9W7LW... (a long string)

This is the **encrypted + signed package**. Only someone with the secret key can decrypt it.

**5. Why Fernet is good for our project**

* We’re working with **sensitive biometrics** (face & voice encodings).
* We want to ensure:
  + Nobody can read them without the key (**confidentiality**).
  + Nobody can modify them (integrity).
* Fernet is easy to use (just .encrypt() and .decrypt()) but also secure enough for real projects.

✅ In short:

**Fernet = AES encryption (keeps it secret) + HMAC authentication (keeps it safe from tampering) all wrapped up in a simple interface.**

**1. What is Fernet?**

* **Fernet** is a system for **symmetric encryption** provided by the Python cryptography library.
* **Symmetric** means: the *same key* is used to both encrypt and decrypt.
* It is designed to be **safe by default**, so you don’t need to worry about low-level crypto mistakes.

**2. What does “authenticated encryption” mean?**

Normally, when people think of encryption, they think only of *hiding the data*.  
But encryption **alone** is not enough. Why?

* Imagine someone intercepts your encrypted file and flips a few random bits.
* When you later decrypt it, you’d get garbage, and you might not even realize the file was tampered with.
* This is where **authentication** comes in.

🔹 **Authenticated encryption** ensures two things:

1. **Confidentiality** → nobody without the key can read the message.
2. **Integrity** → if anyone changes even *one bit* of the ciphertext, decryption fails with an error.

So Fernet doesn’t just hide your data — it also protects it from tampering.

**3. What does Fernet actually do under the hood?**

You don’t need to memorize this, but it’s good to know:

* Fernet uses **AES** (Advanced Encryption Standard) in **CBC mode** for encryption.
* It uses **HMAC (SHA256)** for authentication (integrity check).
* It automatically handles things like:
  + generating random initialization vectors (IVs)
  + timestamping the message (so you can set expiry if you want)
  + combining everything into a single token

**4. What does a Fernet “token” look like?**

When you encrypt something with Fernet, the output is **Base64 text** that includes:

* Version
* Timestamp
* IV (random bits for randomness)
* Ciphertext (encrypted data)
* HMAC (integrity check)

Example:

gAAAAABlZ7Bz5Jm9W7LW... (a long string)

This is the **encrypted + signed package**. Only someone with the secret key can decrypt it.

**5. Why Fernet is good for our project**

* We’re working with **sensitive biometrics** (face & voice encodings).
* We want to ensure:
  + Nobody can read them without the key (**confidentiality**).
  + Nobody can modify them (integrity).
* Fernet is easy to use (just .encrypt() and .decrypt()) but also secure enough for real projects.

✅ In short:

**Fernet = AES encryption (keeps it secret) + HMAC authentication (keeps it safe from tampering) all wrapped up in a simple interface.**

Would you like me to **show you a very tiny demo script** (independent of face recognition) so you can actually see:

* the plaintext,
* the encrypted token, and
* what happens if someone tampers with it?

**You said:**

For a real product use secure key storage (KMS, OS keyring, or environment secrets). Don’t store keys in project root.

what are KMS,OS Keyring etc ?

**ChatGPT said:**

Great question 👏 — because this goes into **real-world key management**, which is one of the most important parts of cryptography.

When I said *“don’t just keep secret.key in your project folder”*, I was pointing toward more secure options like **KMS, OS keyring, or environment secrets**. Let’s break them down in simple words:

**1. KMS (Key Management Service)**

* A **cloud service** that manages encryption keys for you.
* Examples:
  + AWS KMS (Amazon Web Services)
  + Azure Key Vault (Microsoft)
  + Google Cloud KMS
* You never directly see the secret key. Instead:
  + You ask KMS: “Please encrypt this data.”
  + It returns the encrypted data.
  + You ask KMS again: “Please decrypt this (if I’m allowed).”
* Access is controlled by permissions (IAM roles, policies).
* Benefit: You don’t worry about generating, storing, rotating, or protecting keys — KMS handles that securely.

**2. OS Keyring (Operating System Key Vault)**

* Your operating system has a built-in **password manager / secret vault**.
* Examples:
  + **Windows Credential Manager**
  + **macOS Keychain**
  + **Linux Secret Service** (GNOME Keyring, KWallet, etc.)
* Instead of saving a key in secret.key file, you ask the OS:
  + “Store this secret for me.”
  + Later: “Give me back the secret.”
* Only your user account (or your app) can access it.
* In Python, there’s even a library called [keyring](https://pypi.org/project/keyring/) that makes this easy.

**3. Environment Secrets**

* Instead of keeping the key in a file, you load it from an **environment variable**.
* Example on Linux/Mac:
* export FACE\_KEY="my\_secret\_key\_here"
* In Python:
* import os
* key = os.environ["FACE\_KEY"]
* This way, the secret isn’t hardcoded in your code or saved in Git.
* In production (like Docker, Kubernetes, or CI/CD pipelines), you can inject secrets securely at runtime.

**📌 Summary**

* **KMS** → cloud-managed, enterprise-grade, best for serious apps.
* **OS Keyring** → secure, local, good for desktop apps or developer machines.
* **Environment secrets** → simple, flexible, widely used in servers, CI/CD pipelines, Docker.

For **your project right now**, storing secret.key in a file (but excluded via .gitignore) is **okay for learning and prototyping**.  
But if you later want to **deploy this system** (e.g., hackathon or real product), you’d pick one of the above secure storage methods.

**Encryption\_module.py**

* 1. from cryptography.fernet import Fernet: imports Fernet, which helps us encrypt and decrypt data using a secret key.
  2. import os: for working with file paths (not used much here, but useful).
  3. import numpy as np: NumPy library, for handling .npy files (face encodings are stored as NumPy arrays).
  4. from io import BytesIO: allows treating bytes like a file, useful when we don’t want to write decrypted data to disk.
  5. def generate\_key(key\_path: str = DEFAULT\_KEY\_PATH) -> bytes:
  + Generate a new Fernet key and save it to key\_path.
  + Returns the key bytes.
  + key = Fernet.generate\_key()
  + with open(key\_path, "wb") as f:
  + f.write(key)
  + return key
*  Fernet.generate\_key() → produces a secure random key (bytes).
*  with open(key\_path, "wb") as f: f.write(key) → write the key bytes to a file (binary mode).
*  Return the key so the caller can use it if needed.
  1. def load\_key(key\_path: str = DEFAULT\_KEY\_PATH) -> bytes:
  + Load a Fernet key from key\_path.
  + Raises FileNotFoundError if the key file doesn't exist.
  + with open(key\_path, "rb") as f:
  + key = f.read()
  + return key
* Reads the secret key back from the file.
* rb = read in **binary mode**.
* Returns the key so Fernet can use it.
  1. def encrypt\_npy\_file(npy\_path: str, key\_path: str = DEFAULT\_KEY\_PATH, out\_path: str = None) -> str:
* Function to encrypt a NumPy .npy file (like stored face encodings).
* npy\_path: path of the original .npy file.
* out\_path: where to save encrypted version (if not given, we add .enc at the end).
  1. if out\_path is None:
* out\_path = npy\_path + ".enc"
* If user didn’t say where to save, it will make a new file with .enc.
* Example: faces.npy → faces.npy.enc
  1. key = load\_key(key\_path)

fernet = Fernet(key)

* Loads the key we saved earlier.
* Makes a Fernet object so we can encrypt.

10. with open(npy\_path, "rb") as f:

plain\_bytes = f.read()

* Opens the original .npy file.
* Reads the raw bytes of that file (not the array yet, just the bytes).

11. encrypted\_bytes = fernet.encrypt(plain\_bytes)

* Encrypts the file bytes into gibberish (cipher text).

12. with open(out\_path, "wb") as f:

f.write(encrypted\_bytes)

* Saves the encrypted bytes to a new file.

👉 Example:

faces.npy.enc → b'gAAAAABkJ...'

13. return out\_path

* Returns the path of the encrypted file.

14. def decrypt\_npy\_file\_to\_npy(enc\_path: str, key\_path: str = DEFAULT\_KEY\_PATH, out\_npy\_path: str = None) -> str:

* Function to decrypt an encrypted .npy.enc file back to .npy.

15. if out\_npy\_path is None:

if enc\_path.endswith(".enc"):

out\_npy\_path = enc\_path[:-4] # remove .enc

else:

out\_npy\_path = enc\_path + ".dec.npy"

* If output path not given, it will:
* remove .enc and return to original file name, OR
* make a new file with .dec.npy.

16. key = load\_key(key\_path)

fernet = Fernet(key)

* Loads the key, prepares Fernet.

17. with open(enc\_path, "rb") as f:

encrypted\_bytes = f.read()

* Reads the encrypted file.

18. plain\_bytes = fernet.decrypt(encrypted\_bytes)

* Decrypts the encrypted bytes → original .npy file bytes.

19. with open(out\_npy\_path, "wb") as f:

f.write(plain\_bytes)

* Saves decrypted bytes back to disk.

20. return out\_npy\_path

* Returns the path of decrypted .npy file.

21. def decrypt\_npy\_file\_to\_array(enc\_path: str, key\_path: str = DEFAULT\_KEY\_PATH) -> np.ndarray:

* Similar to above, but instead of writing to disk, it directly gives back a NumPy array in memory.

22. key = load\_key(key\_path)

fernet = Fernet(key)

* Load key + Fernet object.

23. with open(enc\_path, "rb") as f:

encrypted\_bytes = f.read()

* Read encrypted file bytes.

24. plain\_bytes = fernet.decrypt(encrypted\_bytes)

* Decrypt to original .npy bytes.

25. bio = BytesIO(plain\_bytes)

arr = np.load(bio, allow\_pickle=False)

return arr

* BytesIO(plain\_bytes): treats decrypted bytes like a file.
* np.load(bio): loads the NumPy array directly from memory, no need to save.
* Returns the array to your code.
* 👉 Example:

arr = decrypt\_npy\_file\_to\_array("faces.npy.enc")

print(arr.shape) # (128,) face encoding vector

⚡ In short:

generate\_key → makes a key once.

load\_key → gets that key.

encrypt\_npy\_file → turns .npy into .npy.enc.

decrypt\_npy\_file\_to\_npy → brings .npy.enc back to .npy.

decrypt\_npy\_file\_to\_array → directly loads array in memory from .npy.enc.

1. **voice\_encrypt.py**
   1. import io

* Imports Python's io module (Input/Output helpers).
* We use it to create an in-memory bytes buffer (BytesIO) to convert numpy arrays to bytes and back.
  1. import os
* Imports the os module for operating-system-related utilities.
* In this file os isn't strictly used later, but commonly kept for file/path utilities (safe to keep or remove).
  1. from typing import Tuple
* Imports Tuple from typing so we can annotate function return types (helps with readability and type checking).
* Example: a function returning two strings can be typed as -> Tuple[str, str].
  1. import numpy as np
* Imports NumPy (numerical arrays) with alias np.
* Used for audio array handling and saving/loading arrays.
  1. import sounddevice as sd
* Imports sounddevice library as sd.
* sounddevice handles recording audio from the microphone (and playback).
  1. import soundfile as sf
* Imports soundfile as sf.
* soundfile can write and read audio files like WAV.
  1. import librosa
* Imports librosa, a popular audio processing library.
* We use it to extract MFCC features from raw audio.
  1. from cryptography.fernet import Fernet
* Imports Fernet class from cryptography package.
* Fernet gives us simple symmetric encryption (generate key, encrypt bytes, decrypt bytes).

9.

def record\_audio(duration: float = 3.0, sr: int = 16000) -> np.ndarray:

"""Record `duration` seconds from default microphone and return numpy array (mono)."""

print(f"Recording {duration} s at {sr} Hz ...")

audio = sd.rec(int(duration \* sr), samplerate=sr, channels=1, dtype='float32')

sd.wait()

audio = audio.flatten()

print("Recording finished.")

return audio

Line-by-line:

* def record\_audio(duration: float = 3.0, sr: int = 16000) -> np.ndarray:
* Defines a function named record\_audio.
* duration default is 3.0 seconds; sr (sampling rate) default 16000 Hz.
* Return type is annotated as np.ndarray (a NumPy array).
* """Record duration seconds ..."""
* Docstring: short description of the function.
* print(f"Recording {duration} s at {sr} Hz ...")
* Prints a friendly message to the console, e.g., "Recording 3.0 s at 16000 Hz ...".
* audio = sd.rec(int(duration \* sr), samplerate=sr, channels=1, dtype='float32')
* Calls sounddevice.rec() to start recording.
* int(duration \* sr) is number of samples (seconds × samples per second).
* channels=1 means record mono (one channel).
* dtype='float32' gives samples as floats between -1.0 and 1.0.
* sd.wait()
* Blocks execution until recording finishes (waits for the device to finish filling the buffer).
* audio = audio.flatten()
* sounddevice returns an array of shape (samples, channels); flatten() converts it to 1-D array (samples only).
* Example: shape (48000, 1) -> (48000,).
* print("Recording finished.")
* Prints that recording is done.
* return audio
* Returns the 1-D NumPy array containing the audio samples.

10. def save\_wav(path: str, audio: np.ndarray, sr: int = 16000):

"""Save numpy audio to wav file."""

sf.write(path, audio, sr)

print(f"WAV saved to: {path}")

 def save\_wav(path: str, audio: np.ndarray, sr: int = 16000):

* Function to write a NumPy audio array to a WAV file.
* path is the filename (e.g., 'out.wav').

 sf.write(path, audio, sr)

* Uses soundfile.write() to write the array to disk with the given sampling rate.

 print(f"WAV saved to: {path}")

* Confirmation message.

11. def extract\_mfcc(audio: np.ndarray, sr: int = 16000, n\_mfcc: int = 13) -> np.ndarray:

"""

Extract MFCC features from a 1-D float audio array.

Returns shape (n\_mfcc, frames).

"""

# librosa expects float32 mono

mfcc = librosa.feature.mfcc(y=audio.astype(np.float32), sr=sr, n\_mfcc=n\_mfcc)

return mfcc

* def extract\_mfcc(...):
* Function to compute MFCC (Mel-Frequency Cepstral Coefficients) from audio.
* n\_mfcc=13 is a common small choice.
* audio.astype(np.float32)
* Ensures audio samples are float32 (librosa prefers floats).
* mfcc = librosa.feature.mfcc(y=..., sr=sr, n\_mfcc=n\_mfcc)
* Calls librosa to compute MFCCs.
* mfcc shape is (n\_mfcc, frames) where frames depends on audio length and analysis hop size.
* return mfcc
* Returns the MFCC matrix.
* Example usage:
* If audio has 3 seconds at 16kHz, mfcc = extract\_mfcc(audio) → returns an array like (13, ~94) (frames vary).

12.

def numpy\_to\_bytes(arr: np.ndarray) -> bytes:

"""Serialize numpy array to bytes using np.save into a BytesIO."""

bio = io.BytesIO()

np.save(bio, arr, allow\_pickle=False)

bio.seek(0)

return bio.read()

* Purpose: convert a NumPy array to raw bytes so encryption can be done.
* bio = io.BytesIO()
* Creates an in-memory bytes buffer (like a file but in memory).
* np.save(bio, arr, allow\_pickle=False)
* Writes the NumPy array in .npy format into that buffer.
* allow\_pickle=False avoids risky pickle serialization.
* bio.seek(0)
* Move pointer to start before reading.
* return bio.read()
* Return the bytes content.

13. def bytes\_to\_numpy(b: bytes) -> np.ndarray:

"""Deserialize bytes back to numpy array."""

bio = io.BytesIO(b)

bio.seek(0)

arr = np.load(bio, allow\_pickle=False)

return arr

* Reverse of numpy\_to\_bytes.
* bio = io.BytesIO(b) wraps the bytes into a file-like buffer.
* arr = np.load(bio, allow\_pickle=False) reads .npy bytes back to NumPy array.
* Returns the array.
* Example usage:
* b = numpy\_to\_bytes(mfcc) then mfcc2 = bytes\_to\_numpy(b) → mfcc2 equals mfcc.

14. # ---------- encryption ----------

def generate\_key() -> bytes:

"""Generate a new Fernet key (save this securely!)."""

return Fernet.generate\_key()

* Returns a new random Fernet key (URL-safe base64-encoded bytes).
* You must save this key if you want to decrypt later.

15. def save\_key(path: str, key: bytes):

with open(path, 'wb') as f:

f.write(key)

print(f"Key saved to: {path}")

* Writes the key bytes to path in binary mode.
* Good practice: protect this file (permissions, server-side storage, etc.).

16. def load\_key(path: str) -> bytes:

with open(path, 'rb') as f:

return f.read()

* Reads the key bytes back from file and returns them.

17. def encrypt\_bytes(data: bytes, key: bytes) -> bytes:

f = Fernet(key)

return f.encrypt(data)

* Fernet(key) creates a Fernet object that knows the key.
* f.encrypt(data) encrypts the bytes data and returns an encrypted token (bytes).

18. def decrypt\_bytes(token: bytes, key: bytes) -> bytes:

f = Fernet(key)

return f.decrypt(token)

* Decrypts and returns the original plaintext bytes. If token or key is wrong, it raises an error.
* Example :
* k = generate\_key()
* cipher = encrypt\_bytes(b'hello', k)
* plain = decrypt\_bytes(cipher, k)

19. def load\_encrypted(path: str) -> bytes:

with open(path, 'rb') as f:

return f.read()

* Reads encrypted bytes back from disk.
* Example usage:
* save\_encrypted('enc.mfcf', token) then later t = load\_encrypted('enc.mfcf').

20. def record\_extract\_encrypt(

out\_enc\_path: str,

key\_path: str,

duration: float = 3.0,

sr: int = 16000,

n\_mfcc: int = 13,

save\_wav\_path: str = None

) -> Tuple[str, str]:

"""

High-level: record -> extract MFCC -> encrypt -> save encrypted bytes and key.

Returns (encrypted\_file\_path, key\_path)

"""

* Function that performs the whole pipeline end-to-end.
* It takes paths where to save encrypted output and key, and other options.
* Returns the two file paths (so caller knows where things were written).

# 1. Record

audio = record\_audio(duration=duration, sr=sr)

Calls record\_audio() to capture audio. audio is a NumPy array.

# optional: save wav (helpful for debugging)

if save\_wav\_path:

save\_wav(save\_wav\_path, audio, sr=sr)

If save\_wav\_path is provided (not None), saves the raw recording to a WAV — useful for debugging or inspection.

# 2. Extract MFCC

mfcc = extract\_mfcc(audio, sr=sr, n\_mfcc=n\_mfcc)

print("MFCC shape:", mfcc.shape)

Extract MFCC matrix from audio.

Print its shape to know how many frames you got (quick debug info).

# 3. Serialize

raw\_bytes = numpy\_to\_bytes(mfcc)

Convert the MFCC NumPy array to bytes (so it can be encrypted).

# 4. Key generation & encryption

key = generate\_key()

token = encrypt\_bytes(raw\_bytes, key)

Generate a new Fernet key and encrypt the serialized bytes to get token.

# 5. Save files

save\_encrypted(out\_enc\_path, token)

save\_key(key\_path, key)

Write encrypted data to out\_enc\_path and write the key to key\_path.

return out\_enc\_path, key\_path

Return the paths for caller convenience.

Example usage:

record\_extract\_encrypt('encrypted\_voice.mfcf', 'voice\_key.key', duration=3.0, save\_wav\_path='debug.wav')

This records, extracts MFCC, encrypts, saves encrypted\_voice.mfcf and voice\_key.key, and optionally debug.wav.

21. def decrypt\_load\_mfcc(enc\_path: str, key\_path: str) -> np.ndarray:

"""Load encrypted file, decrypt, and return MFCC numpy array."""

token = load\_encrypted(enc\_path)

key = load\_key(key\_path)

raw = decrypt\_bytes(token, key)

mfcc = bytes\_to\_numpy(raw)

return mfcc

* This function does the reverse: read encrypted file + key, decrypt, load NumPy array, and return the MFCC matrix.
* Typical use: to compare stored MFCC template with one extracted from a new recording.
* Example usage:
* mfcc = decrypt\_load\_mfcc('encrypted\_voice.mfcf', 'voice\_key.key') → mfcc is an array (13, frames).

22. # ---------- example usage ----------

if \_\_name\_\_ == "\_\_main\_\_":

# CHANGE these file paths as needed

encrypted\_file = "encrypted\_voice.mfcf" # encrypted MFCC file

key\_file = "voice\_key.key"

wav\_debug = "last\_record.wav"

# Record 3 seconds, extract 13 MFCCs, encrypt and save

record\_extract\_encrypt(

out\_enc\_path=encrypted\_file,

key\_path=key\_file,

duration=3.0,

sr=16000,

n\_mfcc=13,

save\_wav\_path=wav\_debug

)

# To test decrypt and load:

loaded\_mfcc = decrypt\_load\_mfcc(encrypted\_file, key\_file)

print("Loaded MFCC shape after decrypt:", loaded\_mfcc.shape)

* if \_\_name\_\_ == "\_\_main\_\_":
* This block runs only when the file is executed directly (python voice\_encrypt.py).
* It won't run if you import this file as a module.
* encrypted\_file = "encrypted\_voice.mfcf" etc.
* Default filenames used by the example.
* record\_extract\_encrypt(...)
* Calls the pipeline, saves encrypted file, key, and WAV for debug.
* loaded\_mfcc = decrypt\_load\_mfcc(...)
* Reads back and decrypts the saved MFCC to verify pipeline works.
* print("Loaded MFCC shape after decrypt:", loaded\_mfcc.shape)
* Shows shape so you can see it matches what was printed earlier.

**Extra notes & small tips (practical)**

* **Key management:** Right now the code creates and saves a fresh key each time. In production you will want:
  + One key per user (or per device).
  + Or use a server-managed key (do not store keys openly on user devices).
  + If the key is lost, data cannot be decrypted — that's expected with good encryption.
* **Switching to AES / existing face scheme:**
  + If your face encryption uses AES and a certain file format, keep using numpy\_to\_bytes and bytes\_to\_numpy exactly the same — only replace encrypt\_bytes / decrypt\_bytes with AES equivalents. That keeps data layout identical.
* **Using embeddings instead of MFCCs:**
  + If you later switch to speaker embeddings (from speechbrain, pyannote, or a tiny neural model), just replace extract\_mfcc with a function that returns the embedding (1-D array). The rest (serialize → encrypt) stays the same.
* **Security caution:**
  + allow\_pickle=False is good (prevents pickle attacks).
  + Protect the key file (voice\_key.key) with file permissions or store it on a secure server.

**Quick summary**

Record audio: record\_audio(duration, sr) → returns 1-D NumPy audio array.

* Save WAV: save\_wav(path, audio, sr) → writes WAV file.
* Extract features: extract\_mfcc(audio, sr, n\_mfcc) → returns (n\_mfcc, frames) matrix.
* Serialize: numpy\_to\_bytes(arr) → bytes; bytes\_to\_numpy(b) → array.
* Encrypt/decrypt: generate\_key(), encrypt\_bytes(data, key), decrypt\_bytes(token, key).
* Save/read encrypted: save\_encrypted(path, token), load\_encrypted(path); save/load key: save\_key, load\_key.
* One-step pipeline: record\_extract\_encrypt(out\_enc\_path, key\_path, ...).
* Load back: decrypt\_load\_mfcc(enc\_path, key\_path) → MFCC array.

**4. main.py**

**Voice Capture & Feature Extraction**

def capture\_voice(duration=VOICE\_DURATION, fs=FS):

print("Recording voice...")

recording = sd.rec(int(duration \* fs), samplerate=fs, channels=1)

sd.wait()

temp\_file = "temp\_voice.wav"

sf.write(temp\_file, recording, fs)

y, sr = librosa.load(temp\_file, sr=fs)

mfccs = librosa.feature.mfcc(y=y, sr=sr, n\_mfcc=13)

return np.mean(mfccs.T, axis=0)

**Explanation:**

* def capture\_voice(...) → Defines a function to record and process voice.
* sd.rec(...) → Records audio from the microphone for duration seconds.
* sd.wait() → Waits until recording finishes.
* sf.write(...) → Saves the recorded audio as a temporary .wav file.
* librosa.load(...) → Loads the audio for processing.
* librosa.feature.mfcc(...) → Extracts MFCC features (main characteristics of voice).
* np.mean(..., axis=0) → Converts MFCC matrix into a single vector for comparison.

**Face Capture**

def capture\_face():

print("Capturing face...")

cap = cv2.VideoCapture(0)

ret, frame = cap.read()

cap.release()

cv2.destroyAllWindows()

encodings = face\_recognition.face\_encodings(frame)

if len(encodings) == 0:

print("No face detected.")

return None

return encodings[0]

**Explanation:**

* Opens the **webcam** to capture an image.
* cap.read() → Reads one frame (picture) from webcam.
* cap.release() → Closes the webcam.
* cv2.destroyAllWindows() → Closes any OpenCV windows.
* face\_recognition.face\_encodings(frame) → Converts face to a **128-D embedding vector**.
* If no face is found → returns None.
* Otherwise → returns the first face encoding.