```
# Cell 1 — install cryptography
!pip install -q cryptography

# Cell 2 — imports and small helpers
import os, json, base64, hashlib, datetime, glob, re
from cryptography.hazmat.primitives.kdf.scrypt import Scrypt
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
from cryptography.hazmat.primitives import constant_time
from google.colab import files

# --- Key-Derivation parameters (tuned for Colab CPU) ---
KDF_N, KDF_R, KDF_P = 2**14, 8, 1
SALT_LEN, NONCE_LEN = 16, 12  # bytes

def derive_key(password: str, salt: bytes, length=32):
    """Derive a 32-byte AES-256 key from password + salt using scrypt."""
    kdf = Scrypt(salt=salt, length=length, n=KDF_N, r=KDF_R, p=KDF_P)
    return kdf.derive(password.encode())

def sha256_bytes(data: bytes) -> str:
    return hashlib.sha256(data).hexdigest()

def b64(x: bytes) -> str:
    return base64.b64encode(x).decode()

def ub64(s: str) -> bytes:
    return base64.b64decode(s.encode())

# Cell 3 — core AES-256-GCM logic

def encrypt_file_local(input_path: str, password: str, out_path: str=None):
    """Encrypt a file, produce .enc, .meta.enc and .header.json."""
    if out_path is None:
        out_path = input_path + ".enc"

    with open(input_path, "rb") as f:
```

```
    plaintext = f.read()

# derive key
salt = os.urandom(SALT_LEN)
key = derive_key(password, salt)
aesgcm = AESGCM(key)

# encrypt main data
nonce_file = os.urandom(NONCE_LEN)
ciphertext = aesgcm.encrypt(nonce_file, plaintext, None)
with open(out_path, "wb") as f:
    f.write(nonce_file + ciphertext)

# build & encrypt metadata
metadata = {
    "original_filename": os.path.basename(input_path),
    "timestamp_utc": datetime.datetime.utcnow().isoformat() + "Z",
    "sha256_plaintext": sha256_bytes(plaintext),
    "enc_filename": os.path.basename(out_path)
}
meta_bytes = json.dumps(metadata, indent=2).encode()
nonce_meta = os.urandom(NONCE_LEN)
meta_cipher = aesgcm.encrypt(nonce_meta, meta_bytes, None)
meta_path = out_path + ".meta.enc"
with open(meta_path, "wb") as mf:
    mf.write(nonce_meta + meta_cipher)

# header (kept plaintext)
header = {
    "salt_b64": b64(salt),
    "kdf": {"n": KDF_N, "r": KDF_R, "p": KDF_P},
    "meta_filename": os.path.basename(meta_path),
    "enc_filename": os.path.basename(out_path)
}
header_path = out_path + ".header.json"
with open(header_path, "w") as hf:
```

```
        json.dump(header, hf, indent=2)

    return {"enc_path": out_path, "meta_path": meta_path,
            "header_path": header_path, "metadata": metadata}


def decrypt_file_local(enc_path: str, password: str):
    """Decrypt AES-256-GCM file & verify SHA-256 integrity."""
    header_path = enc_path + ".header.json"
    meta_path = enc_path + ".meta.enc"

    if not os.path.exists(header_path):
        return {"success": False, "message": f"Header missing: {header_path}"}
    if not os.path.exists(meta_path):
        return {"success": False, "message": f"Metadata missing: {meta_path}"}

    # derive key
    with open(header_path, "r") as hh:
        header = json.load(hh)
    salt = ub64(header["salt_b64"])
    key = derive_key(password, salt)
    aesgcm = AESGCM(key)

    # decrypt main data
    with open(enc_path, "rb") as ef:
        raw = ef.read()
    nonce_file, ciphertext = raw[:NONCE_LEN], raw[NONCE_LEN:]
    plaintext = aesgcm.decrypt(nonce_file, ciphertext, None)

    # decrypt metadata
    with open(meta_path, "rb") as mf:
        raw_meta = mf.read()
    nonce_meta, meta_cipher = raw_meta[:NONCE_LEN], raw_meta[NONCE_LEN:]
    meta_plain = aesgcm.decrypt(nonce_meta, meta_cipher, None)
    metadata = json.loads(meta_plain.decode())
```

```
    # integrity check
    computed = sha256_bytes(plaintext)
    stored = metadata.get("sha256_plaintext", "")
    ok = constant_time.bytes_eq(computed.encode(), stored.encode())

    # write decrypted file
    base, ext = os.path.splitext(metadata.get("original_filename", "output"))
    out_path = base + "_decrypted" + ext
    with open(out_path, "wb") as f:
        f.write(plaintext)

    return {"success": True,
            "message": "Decryption succeeded. Integrity: " + ("PASSED" if ok else "FAILED"),
            "out_path": out_path, "integrity_ok": ok,
            "metadata": metadata}

# Cell 4 — Upload file(s) to encrypt
print("📂 Upload the file(s) you want to encrypt:")
uploaded = files.upload()

if uploaded:
    password = input("Enter a password to derive AES-256 key (remember it; no recovery): ").strip()
    if not password:
        print("❌ No password entered — aborting.")
    else:
        for fname in uploaded.keys():
            print(f"\nEncrypting {fname} ...")
            res = encrypt_file_local(fname, password, out_path=fname + ".enc")
            print("☑ Encrypted:", res["enc_path"])
            print("🗒 Metadata + header created.")
            files.download(res["enc_path"])
            files.download(res["meta_path"])
            files.download(res["header_path"])
        print("\n☑ Encryption complete! Keep all 3 files safely.")
else:
```

```
    print("No file uploaded.")
```

```
📂 Upload the file(s) you want to encrypt:
   task 1.pdf(application/pdf) - 339100 bytes, last modified: 9/22/2025 - 100% done
Saving task 1.pdf to task 1.pdf
Enter a password to derive AES-256 key (remember it; no recovery): ABC123@

Encrypting task 1.pdf ...
☑ Encrypted: task 1.pdf.enc
📄 Metadata + header created.
/tmp/ipython-input-196406346.py:25: DeprecationWarning: datetime.datetime.utcnow() is deprecated and scheduled for removal in a future version. Use timezone-aware objects to represent datetimes in UTC: datetime.datetime.now(datetime.UTC).
  "timestamp_utc": datetime.datetime.utcnow().isoformat() + "Z",

☑ Encryption complete! Keep all 3 files safely.
```

```
# Cell 5 — Smart decryption helper (handles spaces/(1), auto-detects correct .enc file)


# 1️⃣ Rename messy files automatically (remove spaces & "(1)")
for f in glob.glob("*"):
    nf = re.sub(r"\s*\(1\)", "", f)          # remove "(1)"
    nf = nf.replace(" ", "_")                # replace spaces with _
    if nf != f:
        os.rename(f, nf)
        print("Renamed:", f, "→", nf)


# 2️⃣ Upload 3 encrypted parts
print("\n📥 Upload these 3 files:")
print("  • main file (.enc)")
print("  • metadata (.meta.enc)")
print("  • header (.header.json)")
uploaded = files.upload()


# 3️⃣ Find correct main .enc file
enc_files = [f for f in glob.glob("*.enc") if not f.endswith(".meta.enc")]
if not enc_files:
    print("❌ No main .enc file found. Check uploads.")
else:
    main_file = enc_files[0]
    print("☑ Using:", main_file)
    password = input("Enter the password used during encryption: ").strip()
```

```
    result = decrypt_file_local(main_file, password)
    print("\n📜 ", result["message"])
    if result["success"]:
        print("📄 Decrypted file saved as:", result["out_path"])
        print("\nMetadata:")
        print(json.dumps(result["metadata"], indent=2))
        try:
            files.download(result["out_path"])
        except Exception as e:
            print("⚠️ Auto-download may not start:", e)
```

 task 1.pdf.enc.header.json(application/json) - 181 bytes, last modified: 10/23/2025 - 100% done
 task 1.pdf.enc.meta.enc(n/a) - 243 bytes, last modified: 10/23/2025 - 100% done
 task 1.pdf.enc(n/a) - 339128 bytes, last modified: 10/23/2025 - 100% done
Saving task 1.pdf.enc.header.json to task 1.pdf.enc.header.json
Saving task 1.pdf.enc.meta.enc to task 1.pdf.enc.meta.enc
Saving task 1.pdf.enc to task 1.pdf.enc
☑ Using: task 1.pdf.enc
Enter the password used during encryption: ABC123@

📜 Decryption succeeded. Integrity: PASSED
📄 Decrypted file saved as: task 1_decrypted.pdf

Metadata:

```
{
  "original_filename": "task 1.pdf",
  "timestamp_utc": "2025-10-23T05:14:42.994444Z",
  "sha256_plaintext": "236f7870b8fa46fedfc6c575c7fda5750d1a1d82b940a55820edffb190f83a61",
  "enc_filename": "task 1.pdf.enc"
}
```