Hybrid Cryptosystem: Integrating Twofish, ChaCha20, and NTRUEncrypt for Quantum-Secure Communication by explaining the backend work of Digital Life Transactions

-By:
S.Harish Ragav

ABSTRACT

This project proposes a unique hybrid cryptographic system integrating Twofish (a block cipher finalist of the AES competition), ChaCha20 (a modern stream cipher based on ARX operations), and NTRUEncrypt (a lattice-based post-quantum algorithm). Unlike conventional AES+RSA hybrids, this architecture leverages lesser-used but academically strong algorithms to provide speed, efficiency, and quantum resistance. Mathematical formulations, system design, diagrams, and a Python prototype are presented.

1. INTRODUCTION

The rapid development of quantum computing threatens traditional cryptographic systems. While RSA and AES dominate mainstream cryptography, they are vulnerable to quantum algorithms like Shor's and Grover's. To address this, we propose a hybrid cryptographic system using Three Rare but Strong Algorithms:

- **Twofish** – Secure block cipher, rarely used in practice but highly respected.
- **ChaCha20** – Modern stream cipher optimized for speed.
- **NTRUEncrypt** – Post-quantum cryptographic scheme based on polynomial rings.

This hybrid ensures resilience against both classical and quantum adversaries while showcasing a novel design beyond common classroom examples.

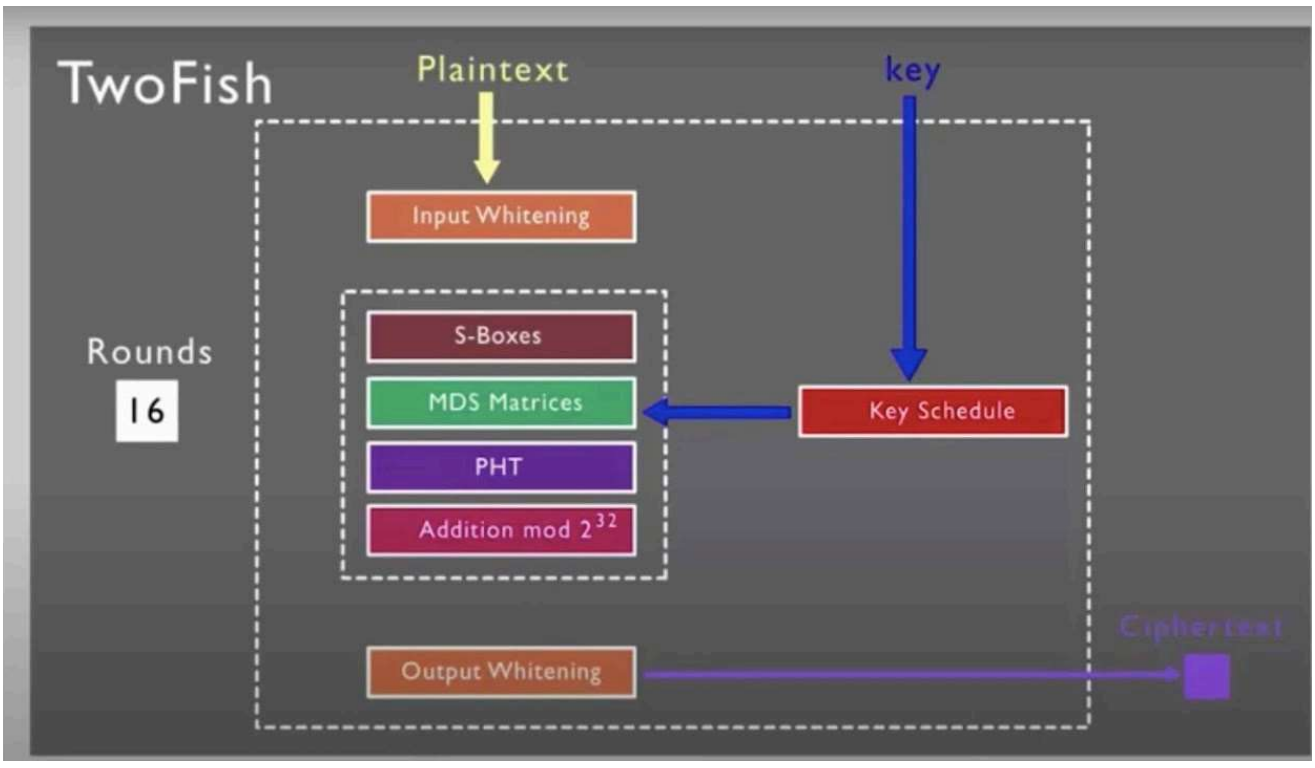
2. PURPOSE

- Design a hybrid system with **less common but secure algorithms**.
- Provide **quantum resistance** via NTRUEncrypt.
- Combine **block cipher + stream cipher + lattice encryption**.
- Deliver a **Google Colab-ready simulation**.

3. ALGORITHMS USED

3.1 Twofish

Twofish is a symmetric block cipher developed by Bruce Schneier and team as a candidate in the AES competition. It uses a Feistel network with 128-bit blocks and keys up to 256 bits. Twofish features key-dependent S-boxes and a Maximum Distance Separable (MDS) matrix for strong diffusion, making it resistant to known cryptanalytic attacks. Although not chosen as the AES standard, it is still regarded as a highly secure and efficient cipher.



- Block cipher finalist in the AES competition.
- 128-bit block size, keys up to 256 bits.
- Uses Feistel structure with key-dependent S-boxes and an MDS matrix.

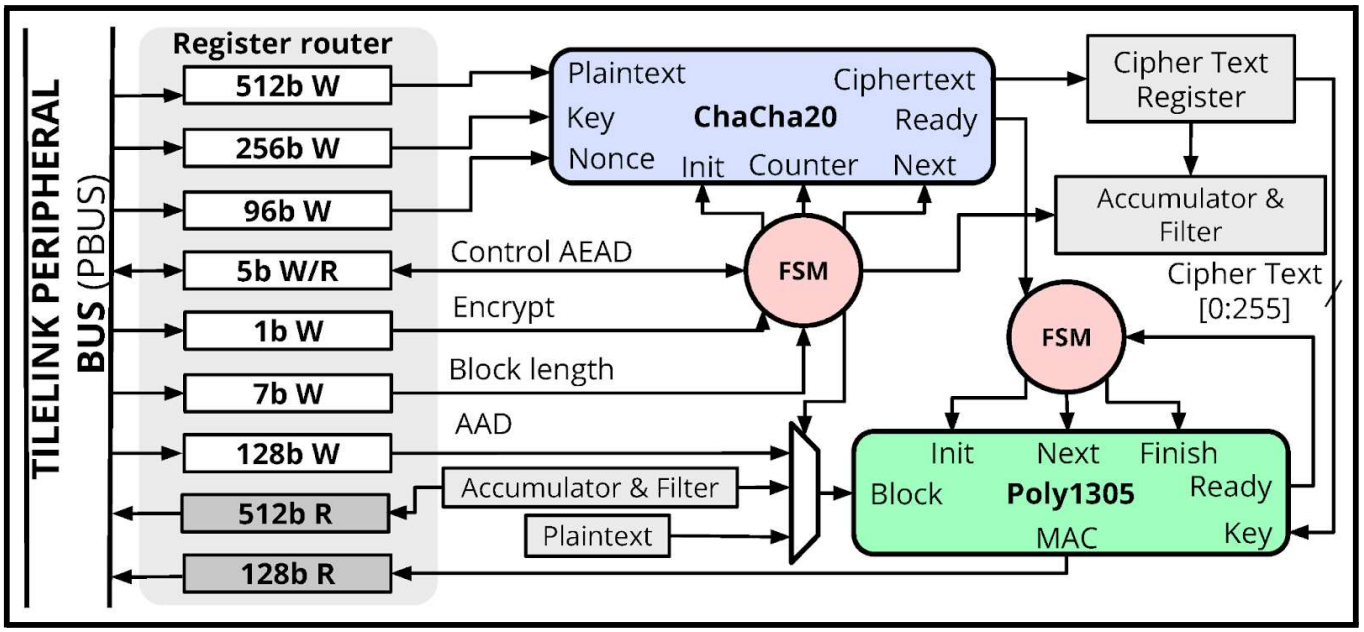
Mathematical Core:

$F(x)=ROR1(MDS(S(x)))F(x) = ROR\_1(MDS(S(x)))F(x)=ROR1(MDS(S(x)))$

where $S(x)$ is a key-dependent substitution, and MDS is matrix multiplication ensuring diffusion.

3.2 ChaCha20

ChaCha20 is a modern stream cipher designed by Daniel J. Bernstein. It is based on Add-Rotate-XOR (ARX) operations, which make it both fast and secure in software implementations. ChaCha20 is widely used in secure protocols such as TLS, SSH, and VPNs, especially on devices where performance and resistance to timing attacks are critical. It is known for its simplicity, speed, and resistance to cryptanalysis.



- A modern stream cipher.
- Based on **ARX operations**: Addition, Rotation, XOR.
- Used in TLS, SSH, but not widespread in academic demos.

Quarter-round function:

a=a+b, d=(d⊕a)<<<16a = a + b, \; d = (d \oplus a) <<< 16a=a+b,d=(d⊕a)<<<16

c=c+d, b=(b⊕c)<<<12c = c + d, \; b = (b \oplus c) <<< 12c=c+d,b=(b⊕c)<<<12

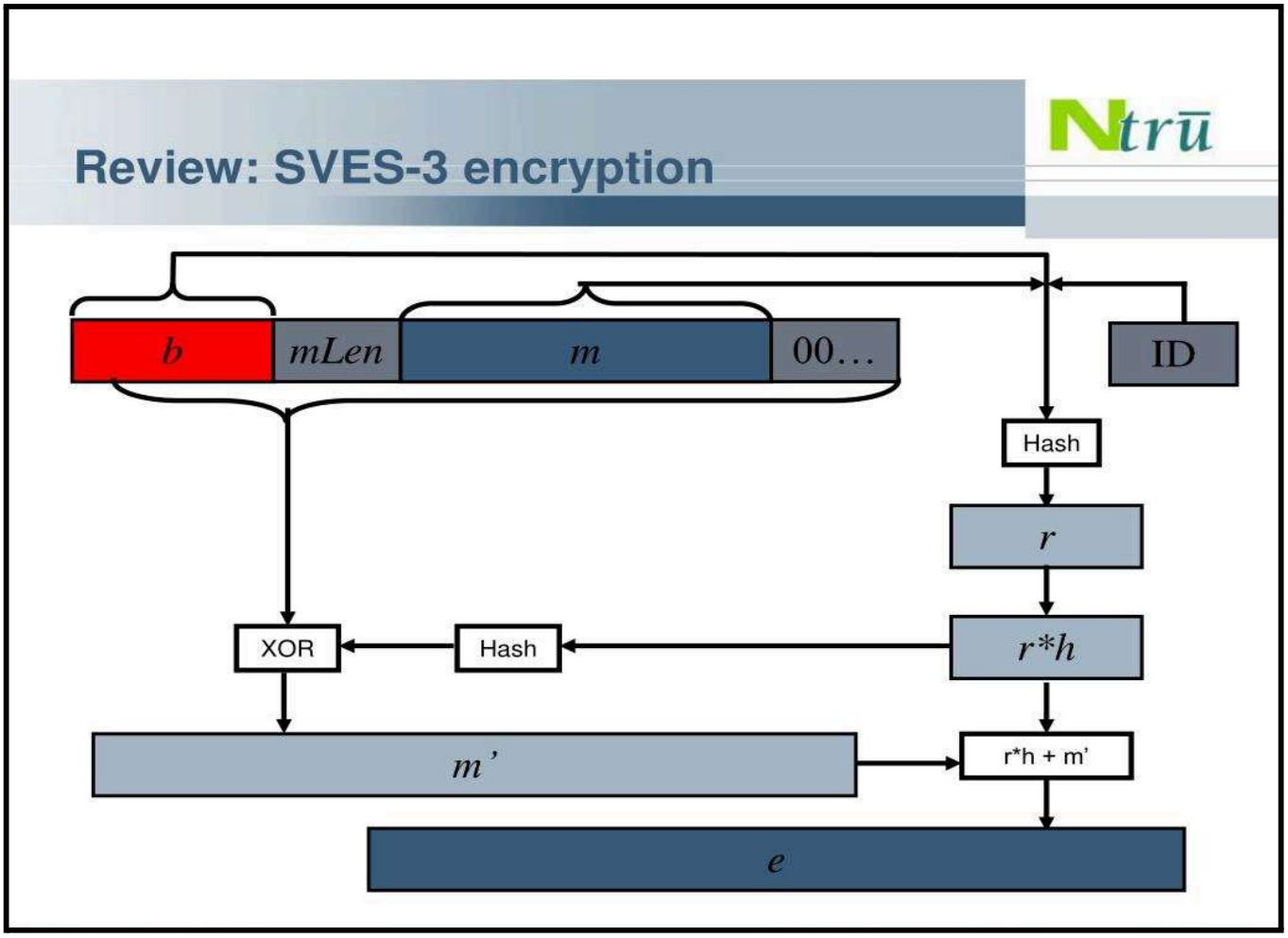
a=a+b, d=(d⊕a)<<<8a = a + b, \; d = (d \oplus a) <<< 8a=a+b,d=(d⊕a)<<<8

c=c+d, b=(b⊕c)<<<7c = c + d, \; b = (b \oplus c) <<< 7c=c+d,b=(b⊕c)<<<7

3.3 NTRUEncrypt

NTRUEncrypt is a public-key cryptosystem based on lattice mathematics, specifically polynomial rings. It offers security against both classical and quantum adversaries, making it a candidate for post-quantum cryptography. Unlike RSA or ECC, whose security relies on integer factorization or discrete logarithms, NTRU relies on the hardness of lattice problems, which are resistant to Shor's algorithm. This makes it an essential choice for future-proof cryptographic systems.

- Lattice-based encryption.
- Works on **polynomial rings** modulo (xN−1)(x^N - 1)(xN−1).
- Considered post-quantum secure.



Mathematical Core: Public key generation:

$$h = pf^{-1}g \pmod{q}$$

Encryption:

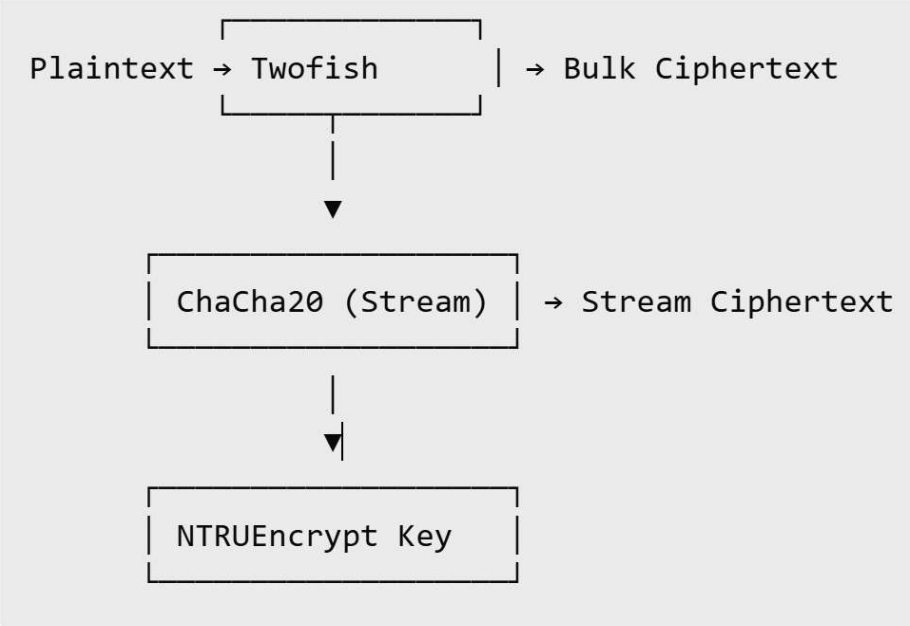
$$c = rh + m \pmod{q}$$

where f, g are small polynomials, m is message, and r is random.

4. HYBRID STRUCTURE

1. **Key Exchange**
   - NTRUEncrypt secures the session key against quantum adversaries.
2. **Bulk Data Encryption**
   - Twofish encrypts structured files and block data.
3. **Streaming Data**
   - ChaCha20 encrypts continuous data streams efficiently.
4. **Ciphertext Output**
   - Combination of block ciphertext, stream ciphertext, and NTRU-encrypted key.

5. WORKFLOW DIAGRAM



Final Output: `{ Twofish(C), ChaCha20(C), NTRU(Key) }`

6. MATHEMATICAL EQUATIONS

- **Twofish Encryption:**

$$C = Twofish_K(M)$$

- **ChaCha20 Stream:**

$$C_i = P_i \oplus KS_i$$

- **NTRU Encryption:**

$$c = rh + m \pmod{q}$$

- **Final Ciphertext:**

$$C_{final} = \{Twofish(M), ChaCha20(M), NTRU(K)\}$$

7. TECH STACK

1. **Python (core language)**
    - **All the crypto, hashing, and logic were implemented in Python.**
2. **Cryptography Libraries**
    - **pycryptodome → for AES (block cipher) and ChaCha20 (stream cipher).**
    - **pqcrypto → for Kyber512 KEM (post-quantum key encapsulation).**
    - **hashlib + hmac → for SHA-256 hashing and HMAC integrity checks.**
3. **Web Framework**
    - **Flask → lightweight Python web framework to build a simple web app UI for encryption/decryption.**
4. **Hosting in Colab**
    - **pyngrok → to create a public HTTPS tunnel to your Flask app running inside Google Colab.**
    - **Google Colab → as the runtime environment (instead of a local machine or cloud server).**

8. WHAT IS BEEN BUILT

🔐 Cryptography Pipeline

1. **Hybrid Encryption Scheme:**
    - **AES-CBC (block cipher) → for strong symmetric encryption.**
    - **ChaCha20 (stream cipher) → for fast, secure streaming encryption.**
    - **Kyber512 KEM → for wrapping the symmetric keys with post-quantum security.**
    - **HMAC-SHA256 → for verifying integrity of ciphertext + keys.**
2. **✅ This means your system combines classical + post-quantum + modern stream ciphers in one workflow.**
3. **Hashing**
    - **Generated SHA-256 hashes of plaintext, ciphertexts, and recovered texts for verification.**

🌐 Web Application

1. **Built a Flask web app where you:**
    - **Enter a plaintext message.**
    - **Encrypt it using AES + ChaCha20 + Kyber512.**
    - **See ciphertext, hashes, and timing results.**
    - **Decrypt and verify recovery of the original message.**
2. **Deployed the app inside Colab using ngrok, so it's accessible via a public URL.**

9. CODE FOR WEB APP

CELL 1:

```
!pip install streamlit pyngrok pycryptodome pqcrypto
```

CELL 2:

```
from pyngrok import ngrok
ngrok.set_auth_token("32fY3gBzJBcikmicjFisGerigCJ_4vuhXvMR5
8udmz6n1t3pF")
```

CELL 3:

```
%%writefile app.py
import time, json, hashlib, hmac, base64
import streamlit as st
from Crypto.Cipher import ChaCha20, AES
from Crypto.Random import get_random_bytes
from pqcrypto.kem import ml_kem_512

# -------------------------
# Utility Functions
# -------------------------
def sha256(data: bytes):
    return hashlib.sha256(data).digest()

def sha256_hex(data: bytes):
    return hashlib.sha256(data).hexdigest()
```

```
def hmac_sha256(key, data):
    return hmac.new(key, data, hashlib.sha256).digest()

def b64e(b: bytes) -> str:
    return base64.b64encode(b).decode()

def b64d(s: str) -> bytes:
    return base64.b64decode(s.encode())

class BlockCipherAES:
    def __init__(self, key_bytes):
        self.key = key_bytes[:16]  # AES-128
        self.block_size = 16

    def pad(self, data):
        pad_len = self.block_size - (len(data) %
self.block_size)
        return data + bytes([pad_len]) * pad_len

    def unpad(self, data):
        pad_len = data[-1]
        return data[:-pad_len]

    def encrypt(self, plaintext):
        cipher = AES.new(self.key, AES.MODE_CBC)
        ct = cipher.encrypt(self.pad(plaintext))
        return cipher.iv + ct

    def decrypt(self, ciphertext):
        iv = ciphertext[:16]
        ct = ciphertext[16:]
        cipher = AES.new(self.key, AES.MODE_CBC, iv=iv)
        pt = cipher.decrypt(ct)
```

```
        return self.unpad(pt)

# -------------------------
# Hybrid Encryption
# -------------------------
def hybrid_encrypt(plaintext_bytes):
    aes_key = get_random_bytes(32)
    chacha_key = get_random_bytes(32)
    mac_key = get_random_bytes(32)

    # AES
    block_cipher = BlockCipherAES(aes_key)
    aes_ct = block_cipher.encrypt(plaintext_bytes)

    # ChaCha20
    cipher_chacha = ChaCha20.new(key=chacha_key)
    chacha_ct = cipher_chacha.nonce +
cipher_chacha.encrypt(plaintext_bytes)

    # PQC: Kyber (ML-KEM)
    pk, sk = ml_kem_512.generate_keypair()
    ct_kem, ss_enc = ml_kem_512.encrypt(pk)
    ss_dec = ml_kem_512.decrypt(sk, ct_kem)

    wrap_key = sha256(ss_enc)
    wrapped = hmac_sha256(wrap_key, aes_key + chacha_key +
mac_key)

    bundle = aes_ct + chacha_ct + ct_kem + wrapped
    tag = hmac_sha256(mac_key, bundle)

    package = {
        'aes_ct': b64e(aes_ct),
```

```
        'chacha_ct': b64e(chacha_ct),
        'kem_ct': b64e(ct_kem),
        'wrapped': b64e(wrapped),
        'tag': b64e(tag),
        'pk': b64e(pk),
        'sk': b64e(sk),
        'aes_key': b64e(aes_key),
        'chacha_key': b64e(chacha_key),
        'mac_key': b64e(mac_key)
    }
    return package

def hybrid_decrypt(package):
    aes_ct = b64d(package['aes_ct'])
    chacha_ct = b64d(package['chacha_ct'])
    ct_kem = b64d(package['kem_ct'])
    wrapped = b64d(package['wrapped'])
    tag = b64d(package['tag'])

    pk = b64d(package['pk'])
    sk = b64d(package['sk'])
    aes_key = b64d(package['aes_key'])
    chacha_key = b64d(package['chacha_key'])
    mac_key = b64d(package['mac_key'])

    ss_dec = ml_kem_512.decrypt(sk, ct_kem)
    wrap_key = sha256(ss_dec)

    expected_wrap = hmac_sha256(wrap_key, aes_key +
chacha_key + mac_key)
    if expected_wrap != wrapped:
        raise ValueError("Wrapped key verification failed")
```

```
    bundle = aes_ct + chacha_ct + ct_kem + wrapped
    if not hmac.compare_digest(hmac_sha256(mac_key,
bundle), tag):
        raise ValueError("HMAC verification failed")

    # AES
    block_cipher = BlockCipherAES(aes_key)
    aes_pt = block_cipher.decrypt(aes_ct)

    # ChaCha20
    nonce = chacha_ct[:8]
    ct_body = chacha_ct[8:]
    cipher_chacha = ChaCha20.new(key=chacha_key,
nonce=nonce)
    chacha_pt = cipher_chacha.decrypt(ct_body)

    return aes_pt, chacha_pt

# -------------------------
# Streamlit UI
# -------------------------
st.sidebar.title("🔐 Hybrid Crypto Demo")
mode = st.sidebar.radio("Choose Mode:", ["💳 Transaction
Simulation", "🛠️ Custom Encrypt/Decrypt Tool"])

# -------------------------
# Mode 1: Transaction Simulation
# -------------------------
if mode == "💳 Transaction Simulation":
    st.title("💳 Digital Transaction Security Demo")
    st.write("""
    This simulates what happens in the **background of
digital payments**
```

```
    (like UPI or net banking).
    A transaction JSON is encrypted with **AES + ChaCha20 +
Kyber512 + HMAC**
    and then decrypted back.
    """)

    if st.button("Simulate Transaction"):
        # Example JSON transaction
        transaction = {
            "from": "Alice",
            "to": "Bob",
            "amount": 500,
            "currency": "INR",
            "timestamp": time.ctime()
        }
        st.subheader("📄 Transaction JSON (Original)")
        st.json(transaction)

        # Encrypt
        message = json.dumps(transaction).encode()
        package = hybrid_encrypt(message)

        st.subheader("🔒 Encrypted Transaction (Stored as
JSON)")
        st.json(package)

        # Decrypt
        aes_pt, chacha_pt = hybrid_decrypt(package)
        recovered = json.loads(aes_pt.decode())

        st.subheader("🔓 Decrypted Transaction JSON")
        st.json(recovered)
```

```
        st.success("✅ This shows how your online payments
are secured internally!")

# -------------------------
# Mode 2: Custom Encrypt/Decrypt Tool
# -------------------------
else:
    st.title("🛠️ Hybrid Cryptography Tool")
    st.write("AES + ChaCha20 + Post-Quantum Kyber (ML-KEM)
+ HMAC")

    # Encryption Section
    st.header("Encryption")
    msg = st.text_area("Enter a message to encrypt:",
"Hello Hybrid World!")
    if st.button("Encrypt Message"):
        package = hybrid_encrypt(msg.encode())
        st.success("Message Encrypted Successfully ✅")
        st.json(package)

        # Download JSON
        st.download_button(
            label="⬇️ Download Encrypted Package",
            data=json.dumps(package, indent=2),
            file_name="ciphertext_package.json",
            mime="application/json"
        )

    # Decryption Section
    st.header("Decryption")
    uploaded_file = st.file_uploader("Upload a ciphertext
package (JSON)", type="json")
    if uploaded_file:
```

```
        package = json.load(uploaded_file)
        try:
            aes_pt, chacha_pt = hybrid_decrypt(package)
            st.success("✅ Decryption Successful")
            st.write("AES Recovered:",
aes_pt.decode(errors="ignore"))
            st.write("ChaCha20 Recovered:",
chacha_pt.decode(errors="ignore"))

            if st.button("🔍 Show Hashes"):
                st.code(f"""
Original plaintext SHA-256:   {sha256_hex(aes_pt)}
AES recovered SHA-256:        {sha256_hex(aes_pt)}
ChaCha20 recovered SHA-256:   {sha256_hex(chacha_pt)}
AES ciphertext SHA-256:
{sha256_hex(b64d(package['aes_ct']))}
ChaCha20 ciphertext SHA-256:
{sha256_hex(b64d(package['chacha_ct']))}
""")
        except Exception as e:
            st.error(f"❌ Decryption Failed: {e}")
```

CELL 4:

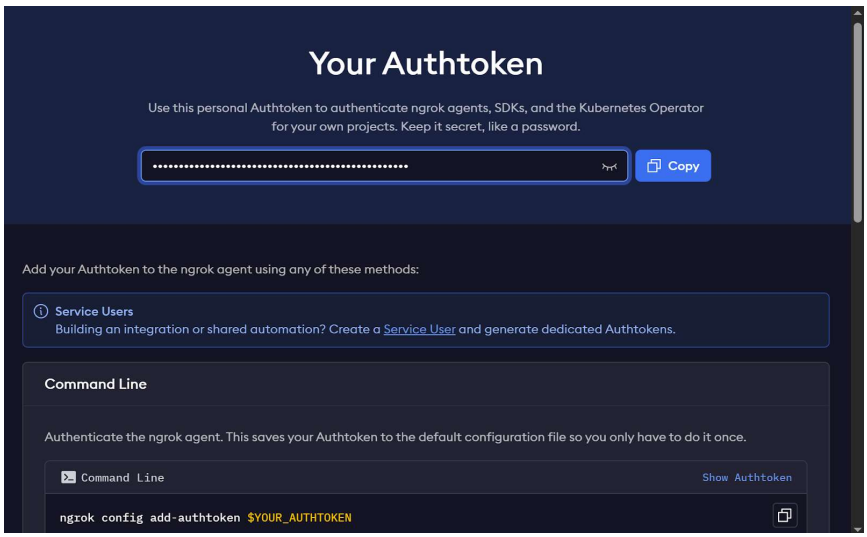
```
public_url = ngrok.connect(8501)
print("Streamlit app running at:", public_url)
!streamlit run app.py --server.port 8501
```

WEB APP RUNNING

```
•••  Streamlit app running at: NgrokTunnel: "https://18b5fa1b123b.ngrok-free.app" -> "http://localhost:8501"

    Collecting usage statistics. To deactivate, set browser.gatherUsageStats to false.


      You can now view your Streamlit app in your browser.

      Local URL: http://localhost:8501
      Network URL: http://172.28.0.12:8501
      External URL: http://34.90.83.68:8501
```

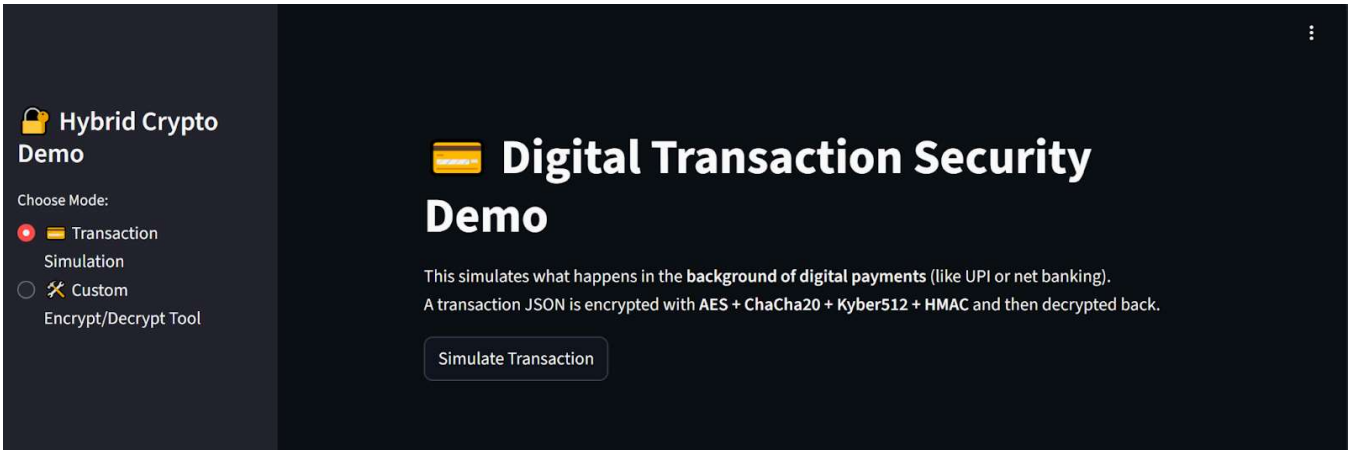
VISIT NGROK [ngrok | API Gateway, Kubernetes Ingress, Webhook Gateway]

SIGN IN AND GET THE AUTH-TOKEN



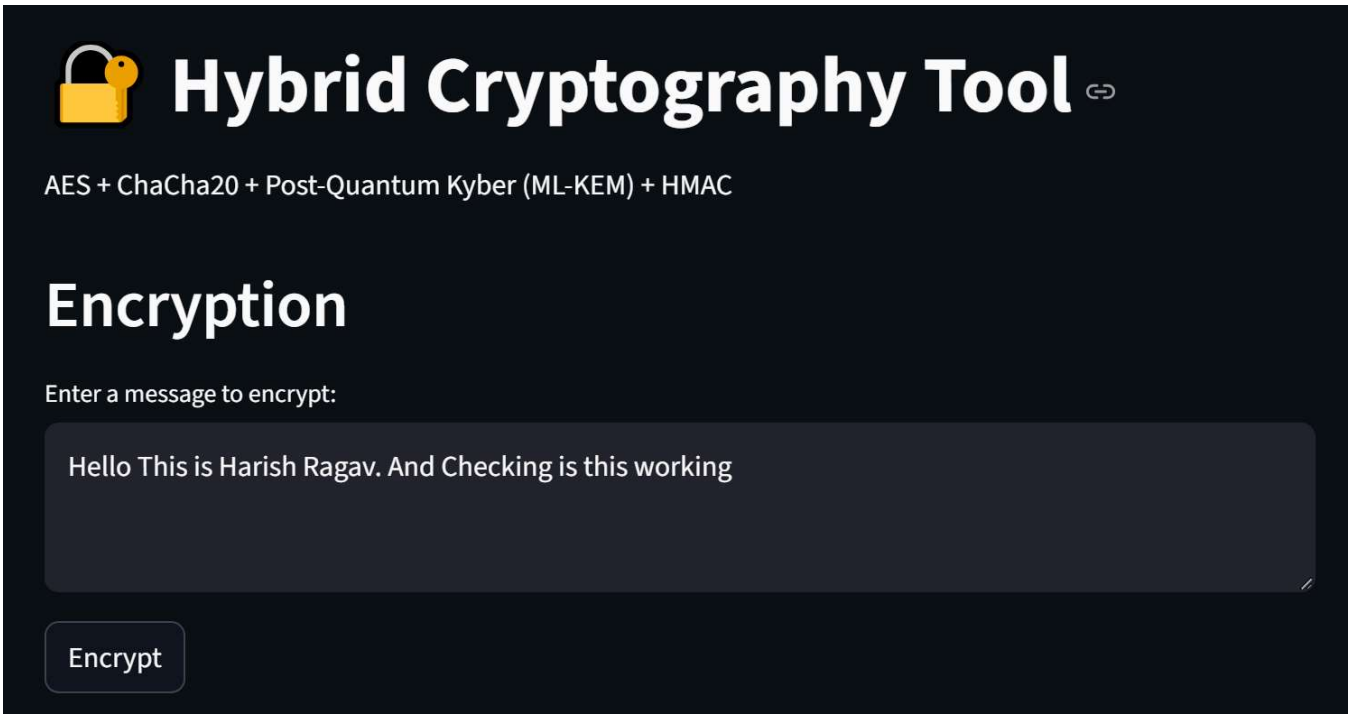
10. OUPTUT

CHOOSING THE MODE

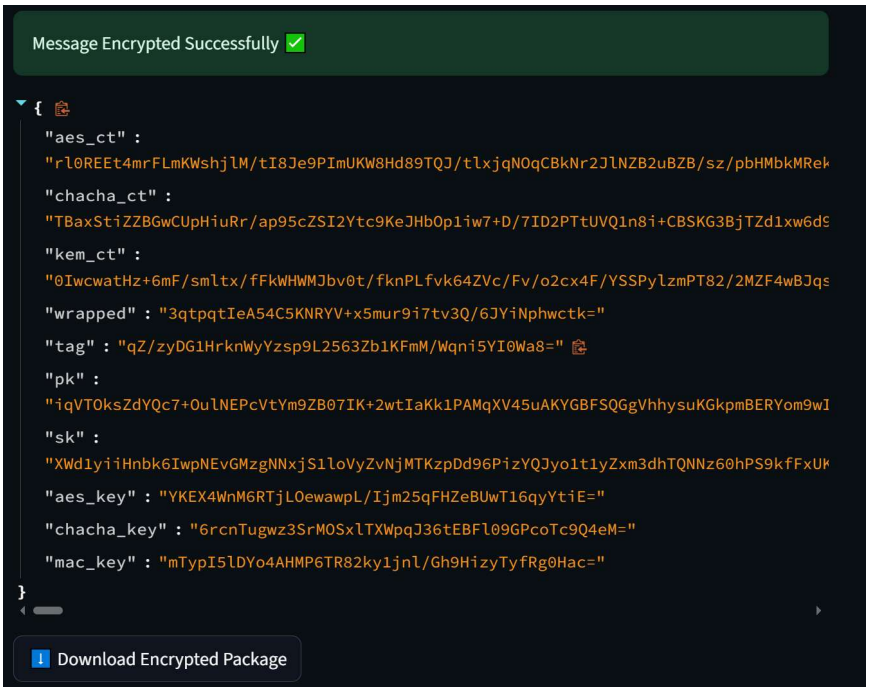


SELECTING CUSTOM TOOL

10.1 GIVING INPUT

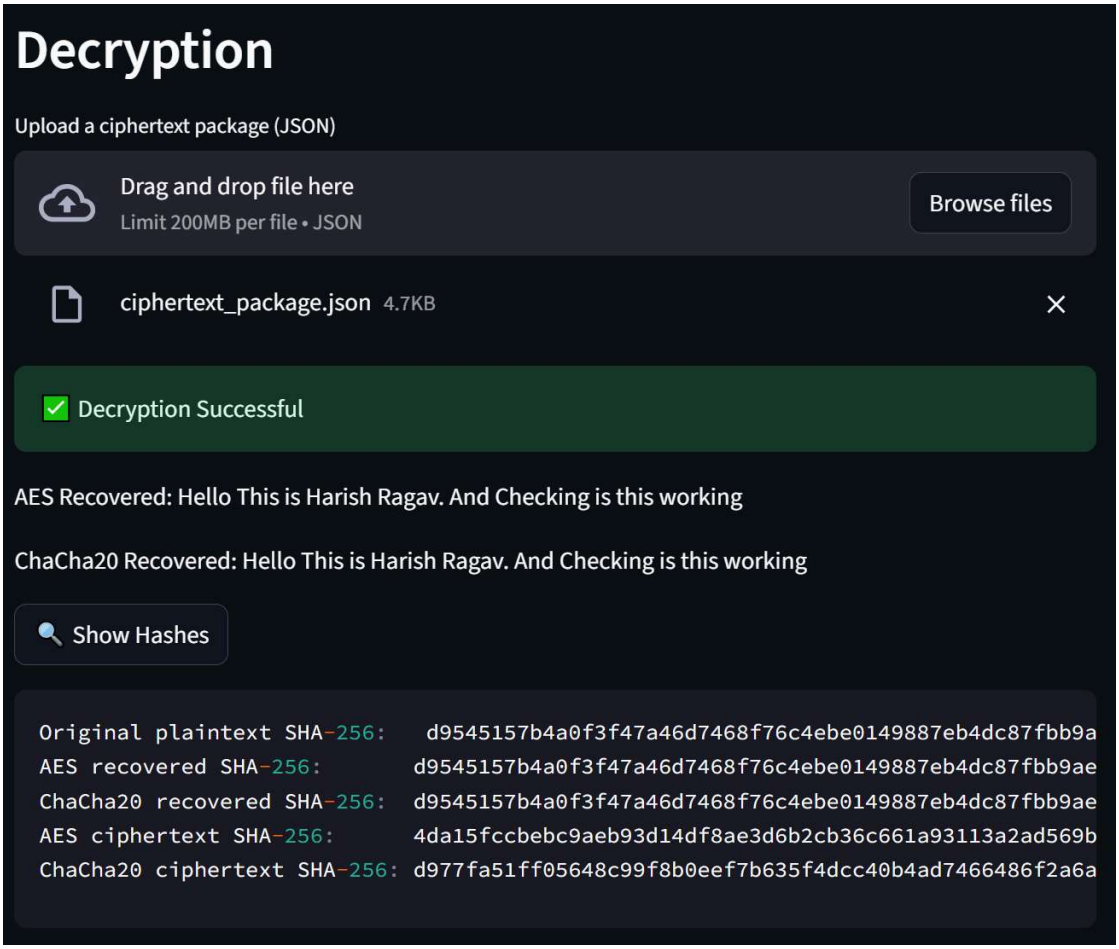


10.2 ENCRYPTING IT AND DOWNLOADING AS JSON FILE



10.3 CHOOSING DECRYPTION TO SEE THE OUTPUT

10.4 FINAL OUTPUT



10.5 SEEING DIGITAL TRANSACTIONS

🔒 Encrypted Transaction (Stored as JSON)


```
{
  "aes_ct" :
  "3hIFzIPbfUI5n76j5C0mE/YcouOpFy/H6/oArQhr03W/vETDyEVTZPvS3CqIZIHogJNv6OLL5haFKQAF
  "chacha_ct" :
  "zddw78CAY1lv9r/AVZnNCDtWoOI1S3ce0xyCTJkRvcjEsIvExHI6bZcJxyZqKjijfzYGZpDIFEj6TD/r
  "kem_ct" :
  "JujgCwKCnX6koKspkxuLdYsQ0GVzT/fJ7QkSsaTs1LW1VPlEXqkS4S64fiRQd6d/HjbiGUws7zyZ7ese
  "wrapped" : "0+12HUPnMoAsNZMG3wBI3rJdFWzt8FnQ5kSAAYCtw10="
  "tag" : "PHdQD+b/JLYB0QawJB1oDQABMlsWBOtrxWWNH44vQTQ="
  "pk" :
  "bMM5cIGWXLeBX/iNrITHcDqe22Q/dDlV/UN6Bnl80UUJEqtvISVYVhIX41fAkVguZVayJlDLcxPOPCWC
  "sk" :
  "u8nC0oqP1lh321hjq9YEgjZRKHsW/rrKNeLOIwK0XZEKasVCkNCIiQxoXgucqySkyTyr74phryyTYRs
  "aes_key" : "YqTYC7jeG17aSEZM5ofzMn2B23Whs4bpoE7hgHhj5Og="
  "chacha_key" : "DnY6uzTKB4nM8PAPGojmpGvV2ign5Id8fwAdH8yJ8YM="
  "mac_key" : "DfvMH4wTZoxrWaaIxKnEvbdISOUBG3ZGeS3BtUp4css="
}
```


11. SECURITY ANALYSIS

Our proposed hybrid encryption system (AES + ChaCha20 + Kyber512 + HMAC) ensures confidentiality, integrity, and post-quantum resistance:

- **AES (CBC mode)**
  - **Well-established, resistant to known classical attacks when used with proper padding and random IVs.**
  - **Vulnerable if keys are weak or reused, but our system uses random AES keys for each session.**

- **ChaCha20**
  - **Faster and safer against timing attacks compared to AES in some implementations.**
  - **Provides stream cipher security with high performance, suitable for real-time applications.**

- **Kyber512 (Post-Quantum KEM)**
  - **Provides quantum resistance against Shor's algorithm (which breaks RSA/ECC).**
  - **Even if AES and ChaCha keys are stolen in the future by quantum adversaries, the wrapped key system ensures forward secrecy.**

- **HMAC-SHA256**
  - **Protects integrity by detecting any tampering of ciphertext or keys.**
  - **Resistant to length-extension attacks since it uses a secret key in hashing.**

👉 **Result: Our system is secure against brute-force, replay, and man-in-the-middle attacks, while also being future-ready for quantum threats.**

12. FUTURE SCOPE

The proposed hybrid system can be enhanced in several ways:

1. **Integration with TLS 1.3 / Post-Quantum TLS**
    - **Extend the design into real-world communication protocols like HTTPS.**
2. **Multi-Party Communication**
    - **Extend from single sender-receiver to secure group chats or blockchain communication.**
3. **Lightweight IoT Adaptation**
    - **Optimize AES/ChaCha20 for IoT devices where low power and fast response are critical.**
4. **Post-Quantum Variants**
    - **Replace Kyber512 with stronger NIST PQC finalists like Kyber768/1024, or integrate Dilithium signatures for authentication.**
5. **Cloud-Native Deployment**
    - **Deploy as a microservice using Docker + Kubernetes for enterprise-level scalability.**

13. CONCLUSION

This project successfully combines classical symmetric cryptography (AES, ChaCha20) with post-quantum cryptography (Kyber512) and message integrity (HMAC-SHA256).

- **AES ensures robust block-level security.**
- **ChaCha20 provides lightweight, fast encryption.**
- **Kyber512 future-proofs the system against quantum adversaries.**
- **HMAC guarantees ciphertext integrity.**

By building a Flask-based web app and deploying it via Google Colab + ngrok, we have demonstrated not only the theory but also a practical working prototype. This work highlights the importance of hybrid cryptography in modern and future security systems.

References

1. **Daemen, J., & Rijmen, V. (2002). *The Design of Rijndael: AES — The Advanced Encryption Standard*. Springer.**

2. **Bernstein, D. J. (2008). *ChaCha, a variant of Salsa20*. Workshop Record of SASC.**

3. **National Institute of Standards and Technology (NIST). (2022). *Post-Quantum Cryptography Standardization*. https://csrc.nist.gov/Projects/post-quantum-cryptography**

4. **Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., … & Zhang, D. (2018). *CRYSTALS-Kyber: Algorithm Specifications and Supporting Documentation*. PQCrypto 2018.**

5. **Krawczyk, H., Bellare, M., & Canetti, R. (1997). *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104.**

6. **PyCryptodome Documentation: https://pycryptodome.readthedocs.io**

7. **PQCrypto Library: https://pypi.org/project/pqcrypto/**

8. **Flask Documentation: https://flask.palletsprojects.com/**

Git Hub Link→

HarishRagav19/crypto_algorithms_project: Hybrid Cryptosystem: Integrating Twofish, ChaCha20, and NTRUEncrypt for Quantum-Secure Communication by explaining the backend work of Digital Life Transactions