□ XenoCipher Message Transmission Workflow

Below is a comprehensive explanation of the message transmission workflow in the **XenoCipher** system. It details how the system handles **NTRU key generation**, **key derivation** for **LFSR**, **chaotic maps**, and **transposition**, as well as **key transmission**, **channel setup**, and the complete **encryption/decryption process**. This documentation reflects the system's design as implemented in the provided source code.

□ Overview: Hybrid Encryption in XenoCipher

XenoCipher is a hybrid cryptographic framework that integrates **quantum-resistant asymmetric encryption (NTRU)** with layered **symmetric encryption techniques**, including:

- **LFSR (Linear Feedback Shift Register)**
- **Chaotic Map-based stream cipher**
- **Transposition cipher**
- Optionally, **ChaCha20** and **Speck** in **ZTM mode (Zero Trust Mode)**

This multi-layered approach balances **strong security** with **efficient key handling** by combining asymmetric and symmetric mechanisms, ensuring robust message confidentiality and integrity even in quantum-threat environments.

1. □ NTRU Key Generation

□ Where and How It Occurs

- **Location:** The NTRU key pair is generated on the **server side** during the initialization of the Flask application (`app.py`).
- **Process:**

The `NTRU.generate_keys` function is called with these parameters:

  - Polynomial degree `N = 743`
  - Small modulus `p = 3`
  - Large modulus `q = 2048`
  - `df` and `dg` define the number of ±1 coefficients in private/public key polynomials

□ Key Details

- **Private Key (f, f_p):**

    - `f` is a ternary polynomial with `df` randomly placed `+1` and `-1` coefficients.
    - `f_q`: Inverse of `f` modulo `q`, computed using a fallback algorithm for robustness.
    - `f_p`: Inverse of `f` modulo `p`, computed using a simpler inversion method.
- **Public Key (h):**
    - Computed using the formula:

```
h = p * f_q * g mod q
```

where `g` is another polynomial with `dg` ±1 coefficients.

- **Storage:**

The `(h, f, f_p)` key pair is stored globally on the server for use in secure key exchange.

□ Purpose

NTRU provides a **quantum-resistant** mechanism for **securely exchanging the master key** between the sender and receiver.

2. □ Master Key Generation & Key Derivation

□ Master Key Creation

- **Function:** `generate_master_key()`
- **When:** Executed once during server startup.
- **How:** Combines entropy from:
    - `os.urandom(32)`
    - Current timestamp
    - Random bit sequences

Then hashes this entropy using **SHA-512** to produce a 64-byte high-entropy key.

□ Deriving Component Keys

- **Function:** `derive_keys()`
- **Purpose:** Generates unique keys for each cryptographic layer (LFSR, chaotic map, transposition, ChaCha20, Speck).
- **Inputs:** Master key, data length, and encryption mode (`'normal'` or `'ztm'`).

□ *Derivation Process*

1. Generate a **salt** based on:
    a. Data length
    b. Encryption mode

Then hash it using **SHA-256**.

2. Combine the salt and master key using a **PBKDF2-like key-stretching approach** with 1000 iterations and SHA-256.
3. Resulting 64-byte key is segmented:
    a. **LFSR Seed:** First 2 bytes
    b. **Chaotic Map Parameters:**
        i. `x0`: Bytes 2–6 → float in (0, 1)
        ii. `r`: Fixed at **3.9** for chaos
    c. **Transposition Key:** Bytes 6–14
    d. **ChaCha20 Key & Nonce:** Bytes 14–46 (key), 46–62 (nonce)
    e. **Speck Key:** Bytes 32–48

□ Purpose

Ensures each encryption operation is **deterministic** and **synchronized** across both sender and receiver using the **same master key**—no need to transmit individual component keys.

3. □ Key Transmission

□ Public Key (NTRU)

- **In Code:** Not explicitly transmitted.
- **In Practice:** Should be:
    - Pre-installed
    - Shared over HTTPS
    - Or exposed via a secure API endpoint

□ Master Key Exchange

- **Method:**
    - Sender encrypts the master key using the **receiver's NTRU public key**.
    - Encrypted key is transmitted securely.
    - Receiver decrypts it using their **NTRU private key**.
- **Function Used:** `NTRU.encrypt_message()` and `NTRU.decrypt_message()`
- **Security Guarantee:** Provides **quantum-safe confidentiality** during master key exchange.

□ Derived Keys Transmission

- **Not Required:**

Because they are **locally derived** using the master key + deterministic salt, these keys are **never transmitted**, minimizing exposure.

4. □ Channel Creation for Data Transmission

- **Medium:** Flask server over **HTTP**

Data is exchanged via standard HTTP requests and responses—e.g., JSON payloads from `/encrypt`.

□ Security Note

- **HTTP itself is not secure.**
- In this implementation:
    - **Encryption ensures data confidentiality** even over unsecured channels.
    - **However, TLS (HTTPS)** is **recommended in practice** to protect metadata and prevent MITM attacks.

5. □ Encryption & Decryption Pipeline

□ Encryption Process

1. **Key Derivation:** Based on input length and selected mode (`normal` or `ztm`)
2. **Pipeline (inside `encrypt` function):**
    a. **[ZTM Mode]** ChaCha20 encryption
    b. **LFSR encryption:** XOR with LFSR keystream
    c. **Chaotic Map encryption:** XOR with chaotic keystream
    d. **Transposition cipher:** Byte permutation
    e. **[ZTM Mode]** Speck in CTR mode
3. **Output:** Final ciphertext (typically as a hex string)

□ Transmission

- Encrypted data is sent to the receiver as a **JSON payload** in the HTTP response from `/encrypt`.

□ Decryption Process

1. **Key Derivation:** Same inputs yield identical keys (deterministically derived).
2. **Pipeline (inside `decrypt` function, reverse order):**
    a. **[ZTM Mode]** Undo Speck CTR
    b. Inverse transposition
    c. Chaotic keystream XOR
    d. LFSR keystream XOR
    e. **[ZTM Mode]** Undo ChaCha20
3. **Output:** Recovered plaintext

6. □ Complete Workflow Summary

□ Initialization

- On server start:
  - Generate **NTRU key pair** (pub & priv)
  - Generate **master key**

□ Master Key Exchange

- Sender encrypts the master key using NTRU
- Receiver decrypts using private key

□ Encryption Request

- Client submits plaintext and mode to `/encrypt`
- Server derives symmetric keys and encrypts using the full pipeline
- Server returns ciphertext in the response

□ Decryption

- Client uses the shared master key and matching derivation logic
- Client decrypts ciphertext by applying the reverse pipeline
- Original plaintext is recovered

□ Summary

Component	Description
NTRU Keys	Generated on server at startup for secure, quantum-resistant key exchange
Master Key	Shared securely using NTRU encryption
Symmetric Keys	Derived deterministically from the master key—never transmitted
Transmission Channel	HTTP (unencrypted by default); relies on encrypted payloads
Encryption Pipeline	Multi-layered: LFSR, chaotic maps, transposition, ChaCha20/Speck in ZTM
Decryption	Reverse pipeline using identical derived keys

✓Final Note

Yes, the **message is encrypted using three layers**: **LFSR**, **chaotic map**, and **transposition cipher**.

In **ZTM mode**, **ChaCha20** and **Speck** are additionally applied.

The **master key itself is encrypted using the NTRU algorithm** to ensure a **secure and quantum-resistant exchange** between the sender and receiver.

Let me know if you'd like a markdown-styled version or visuals/diagrams to accompany this documentation.