Jormungandr Post Quantum Safe VPN (Python3) – System Documentation

Xaoc Industries Inc.

By: William J Appleton

Date: December 2025

Introduction

This document provides a technical description and functional breakdown of the *Jormungandr Quantum Safe VPN* implementation. It is intended to accompany the published source code and to describe the structure, purpose, and behavior of the system at a component level.

The VPN system described herein is a user-space virtual private network implementation designed for experimental and research purposes. It utilizes post-quantum cryptographic primitives for key exchange and authentication, and authenticated symmetric encryption for data transport.

This document is intended for a technical audience, including software engineers, security researchers, reviewers, and auditors, who seek to understand how the system operates, how responsibilities are divided between components, and what assumptions the system makes about its execution environment.

This document does not constitute a security proof, formal verification, or guarantee of safety. The system is not represented as production-ready software, and no claims are made regarding resistance to all classes of attack, regulatory compliance, or suitability for deployment in high-risk or safety-critical environments.

The purpose of this document is descriptive rather than promotional. It explains *what the system does*, *how it is structured*, and *where its boundaries lie*, without asserting claims beyond what is directly implemented in the source code.

Usage

Generating Certificates

python3 snek-vpn.py -g /path/to/store/certificates/

Server Mode

sudo python3 snek-vpn.py -l -x /path/to/authorized/pub/certs/

Client Mode

sudo python3 snek-vpn.py -c 192.168.0.123 -x /path/to/priv/cert/

Optional Flags

to control keyport (TCP) and dataport (UDP) numbers:

Server:

sudo python3 snek-vpn.py -l -x /path/to/authorized/pub/certs/ -kp 80 -dp 1024

Client:

sudo python3 snek-vpn.py -c 192.168.0.123 -x /path/to/priv/cert/ -kp 80 -dp 1024

Breakdown

1. Imports

```
#Begin Imports

from cryptography.hazmat.primitives.ciphers.aead import ChaCha20Poly1305

from cryptography.hazmat.primitives.kdf.hkdf import HKDF

from cryptography.hazmat.primitives import hashes

from concurrent.futures import ThreadPoolExecutor

import oqs, time, netifaces, random, base64, json, argparse, os, secrets, socket, fcntl, struct, threading

#End Imports
```

1.1. Purpose

The Imports section serves to call external dependencies. These dependencies are explained in 1.2. Discussion.

1.2. Discussion

"from cryptography.hazmat.primitives.ciphers.aead import ChaCha20Poly1305" – The quoted line serves to import ChaCha20Poly1305 symmetric cryptographic functionality from the standard cryptography module. This functionality supports encryption of the data layer over UDP.

"*from cryptography.hazmat.primitives.kdf.hkdf import HKDF"* – The quoted line serves to import HMAC-based Key Derivation Function cryptographic functionality from the standard cryptography module. This functionality supports the derivation of keys from shared secrets on the client and server sides of the VPN connection.

"from cryptography.hazmat.primitives import hashes" - The quoted line serves to import Hashing cryptographic functionality from the standard cryptography module. This functionality supports the derivation of keys from shared secrets on the client and server sides of the VPN connection.

"from concurrent.futures import ThreadPoolExecutor" – The quoted line serves to import the ThreadPoolExecutor functionality from the concurrent futures module. This functionality supports the server-side in recovering from disconnects and re-establishing connectivity.

"import oqs, time, netifaces, random, base64, json, argparse, os, secrets, socket, fcntl, struct, threading" – The quoted line imports functionalities for the following purposes:

- **oqs** – This functionality supports post-quantum safe cryptography as per the project located at (https://github.com/open-quantum-safe/liboqs-python). The purpose of this functionality in the VPN system is to enable post-quantum safe key exchange and identity management.
- **time** – This functionality supports time based actions. The purpose of this functionality in the VPN system is to allow the server side of the connection time (500ms) to listen for incoming udp traffic on the data channel.
- **netifaces** – This functionality supports operations on network interfaces. The purpose of this functionality in the VPN system is to allow for configuration of the virtual TUN interface snk#.
- **random** – This functionality supports non-cryptographic random generation. The purpose of this functionality in the VPN system is to allow for a random index to be selected for packet bucket fill selection within the pre-generated random padding.
- **base64** – This functionality supports base64 encoding/decoding operations. The purpose of this functionality in the VPN system is to encode the raw byte data during the Kyber1024 key exchange and during the Dilithium3 certificate signing.
- **json** – This functionality supports the packing and parsing of data into JSON format. The purpose of this functionality in the VPN system is to pack the shared ChaCha20Poly1305 key and the associated Dilithium3 signature for exchange between nodes.
- **argparse** – This functionality supports the parsing of arguments from the command line in a structured manner. The purpose of this functionality in the VPN system is to parse user arguments.
- **os** – This functionality supports several operating system functions. The purpose of this functionality in the VPN system is to allow for reading to and from data storage, devices such as /dev/urandom, and for reading directory listings.
- **secrets** – This functionality supports generating cryptographically random bits. The purpose of this functionality in the VPN system is to generate fragmentation identifiers for fragmented packets.

- **socket** – This functionality supports low-level network communication using TCP and UDP sockets. The purpose of this functionality in the VPN system is to establish both the reliable TCP-based control and key-exchange channels and the UDP-based encrypted data transport channel between VPN peers.
- **fcntl** – This functionality supports file control and ioctl system calls on Unix-like operating systems. The purpose of this functionality in the VPN system is to configure and control the virtual TUN network interface, including binding the interface to a file descriptor and applying interface flags.
- **struct** – This functionality supports packing and unpacking of binary data into and from byte sequences. The purpose of this functionality in the VPN system is to construct and parse protocol headers, encode sequence numbers, fragmentation metadata, payload lengths, and other binary protocol fields in a platform-independent manner.
- **threading** – This functionality supports the creation and management of lightweight execution threads within a single process. The purpose of this functionality in the VPN system is to allow concurrent reading from and writing to the TUN interface and network socket, enabling full-duplex encrypted packet transport.

2. Banner

```
#Start Banner
print("--------------------------------------------------------------------------------")
print("|            Jormungandr Quantum Safe VPN by Mephistopheles                |")
print("--------------------------------------------------------------------------------")
#End Banner
```

2.1. Purpose

The purpose of the banner section is to print a banner to the user terminal.

2.2. Discussion

See Python3 print function documentation for more information.

3. Constant Assignment

```
#Begin Constant Assignment
TUN_PATH = "/dev/net/tun"
interfaces = os.listdir('/sys/class/net/')
iface_count = 0
IFACE_NAME = ""
while not IFACE_NAME:
    testname = "snk" + str(iface_count)
    if testname not in interfaces:
        IFACE_NAME = testname
        print(f"[+] Selected interface name {IFACE_NAME}")
    else:
        iface_count += 1
BUF_SIZE = 4096
NONCE_LEN = 12
HEADER_LEN = 25
CTRL_FLAG = 0x01
FRAG_FLAG = 0x02
FRAG_EXT_LEN = 12
MAX_CHUNK = 1536
TUNSETIFF = 0x400454ca
IFF_TUN = 0x0001
IFF_NO_PI = 0x1000
SIOCSIFMTU = 0x8922
PRECOMP_RANDOM_POOL = os.urandom(1536)
REASSEMBLY = {}
SEQ_BLKLST = set()
#End Constant Assignment
```

3.1. Purpose

The purpose of the Constant Assignment section is to set the state of global variables used throughout the execution of the VPN system.

3.2. Discussion

3.2.1 Operating System and Interface Constants

TUN_PATH = "/dev/net/tun"

This constant defines the filesystem path to the Linux TUN/TAP virtual network device. The VPN system assumes execution on a Unix-like operating system that provides a TUN character device for user-space network tunneling.

interfaces = os.listdir('/sys/class/net/')

This value is populated at runtime to enumerate existing network interfaces. The purpose of this operation is to avoid interface name collisions when creating the VPN tunnel device.

IFACE_NAME and related selection logic

The interface naming logic selects the first available interface name of the form snk#. This deterministic naming scheme ensures predictable interface identification while avoiding overwriting or interfering with existing network interfaces.

3.2.2 Protocol and Cryptographic Constants

NONCE_LEN = 12

This constant defines the nonce length used for ChaCha20-Poly1305 authenticated encryption. A 96-bit nonce length is required by the ChaCha20-Poly1305 AEAD construction and aligns with standard cryptographic usage.

HEADER_LEN = 25

This constant defines the minimum protocol header length for unfragmented packets. The header includes control flags, sequence number, and ciphertext length fields. Fragmented packets extend this header as defined elsewhere in the protocol.

CTRL_FLAG = 0x01

This flag indicates a control or protocol-level packet. Control flags are used to distinguish protocol metadata from pure data payloads.

FRAG_FLAG = 0x02

This flag indicates that the packet is part of a fragmented ciphertext sequence. Fragmented packets include additional header fields describing fragment ordering and reassembly requirements.

3.2.3 Fragmentation and Transport Constraints

BUF_SIZE = 4096

This constant defines the maximum read size when pulling packets from the TUN interface. This value is intentionally larger than the maximum transmission unit to ensure complete packet reads.

MAX_CHUNK = 1536

This constant defines the maximum ciphertext payload size per UDP packet. The value is chosen to remain safely within standard Ethernet MTU limits while accounting for protocol headers and encryption overhead.

FRAG_EXT_LEN = 12

This constant defines the additional header length required when fragmentation is enabled. The extended fragment header contains total ciphertext length, fragment identifier, fragment index, and fragment count.

3.2.4 Kernel Interface Control Constants

TUNSETIFF = 0x400454ca

This constant defines the ioctl command used to create and configure a TUN interface. It is platform-specific to Linux systems.

IFF_TUN = 0x0001

This flag specifies that the created interface is a TUN device operating at layer 3.

IFF_NO_PI = 0x1000

This flag disables the additional packet information header normally

prepended by the kernel, allowing the VPN system to operate directly on raw IP packets.

SIOCSIFMTU = 0x8922
This constant defines the ioctl command used to configure the MTU of the virtual network interface.

3.2.5 Runtime State Containers

PRECOMP_RANDOM_POOL = os.urandom(1536)
This constant defines a pool of pre-generated random bytes used exclusively for traffic padding. Precomputing padding material reduces runtime entropy generation overhead while maintaining non-deterministic packet lengths.

REASSEMBLY = {}
This dictionary serves as a runtime state container for fragmented packet reassembly. Fragmented ciphertext chunks are stored by fragment identifier until all fragments are received and reassembled.

SEQ_BLKLST = set()

This set is to serve as a list of seen sequence numbers. The list is added to for every sequence number seen, preventing replay.

4. System Functions

```
#Begin System Functions
def create_tun():
    tun = os.open(TUN_PATH, os.O_RDWR)
    ifr = struct.pack('16sH14s', IFACE_NAME.encode(), IFF_TUN | IFF_NO_PI, b'\x00' * 14)
    fcntl.ioctl(tun, TUNSETIFF, ifr)
    print(f"[+] Opened TUN device: {IFACE_NAME}")
    return tun


def set_tun_mtu(iface, mtu):
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    ifr = struct.pack('16sH14s', iface.encode(), mtu, b'\x00' * 14)
    fcntl.ioctl(s, SIOCSIFMTU, ifr)
    s.close()
#End System Functions
```

4.1. Purpose

The purpose of the System Functions section is to define low-level operating system interactions required to create and configure the virtual network tunnel interface used by the VPN system.

4.2. Discussion

4.2.1 create_tun()

The create_tun() function is responsible for creating and initializing a virtual TUN network interface within the host operating system.

This function opens the system TUN character device defined by TUN_PATH and issues an ioctl system call to request creation of a new virtual interface. The interface is configured using the IFF_TUN flag, indicating layer-3 (IP) operation, and the IFF_NO_PI flag, disabling the additional packet information header normally prepended by the kernel. Disabling packet information allows the VPN system to operate directly on raw IP packets without additional kernel metadata.

The function binds the newly created interface to a file descriptor, which is returned to the caller for subsequent read and write operations. This file descriptor serves as the primary interface between the VPN system and the host network stack.

This function assumes execution on a Linux-based operating system that supports /dev/net/tun, and that the executing process has sufficient privileges to create virtual network interfaces.

The function does not assign IP addresses, routing rules, firewall policies, or any other network configuration to the interface. Such configuration is expected to be handled externally by the system administrator or calling environment.

4.2.2 set_tun_mtu(iface, mtu)

The set_tun_mtu() function is responsible for configuring the Maximum Transmission Unit (MTU) of a specified network interface.

This function creates a temporary socket and issues an ioctl system call using the SIOCSIFMTU command to update the MTU value associated with the interface name provided as input. The MTU value is selected to align with the protocol's fragmentation and transport constraints defined elsewhere in the system.

The function assumes that the specified interface already exists and that the executing process has sufficient privileges to modify interface parameters.

This function does not validate network reachability, enforce MTU compatibility with remote peers, or adjust protocol parameters dynamically. It performs a single, explicit configuration operation and returns control to the caller.

4.2.3 Scope and Limitations

The System Functions defined in this section are intentionally minimal and narrowly scoped. They exist solely to bridge the VPN system with the host operating system's virtual networking facilities.

These functions do not implement cryptographic operations, protocol logic, authentication, authorization, or traffic filtering. Their purpose is limited to interface creation and configuration, with all higher-level behavior implemented in subsequent sections of the system.

5. Protocol Functions

```
#Begin Protocol Functions
def fragmentor(ct, seq, nonce):
    frag_id = secrets.randbits(32)
    total = len(ct)
    frag_cnt = (total + MAX_CHUNK - 1) // MAX_CHUNK
    offset = 0
    frag_idx = 0
    chunked_packets = []
    while offset < total:
        chunk = ct[offset: offset + MAX_CHUNK]
        offset += len(chunk)
        header = bytearray()
        flags = CTRL_FLAG | (FRAG_FLAG if frag_cnt > 1 else 0)
        header += bytes([flags])
        header += struct.pack("<Q", seq)
        header += struct.pack("<I", len(chunk))
        if frag_cnt > 1:
            header += struct.pack("<I", total)
            header += struct.pack("<I", frag_id)
            header += struct.pack("<H", frag_idx)
            header += struct.pack("<H", frag_cnt)
            frag_idx += 1
        seq = (seq + 1) & 0xFFFFFFFFFFFFFFFF
        packet = header + nonce + chunk
        chunked_packets.append(packet)
    return chunked_packets, seq
```

```
def defragmentor(frag_id, frag_cnt, frag_idx, chunk):
    if frag_id not in REASSEMBLY:
        REASSEMBLY[frag_id] = [None] * frag_cnt
    REASSEMBLY[frag_id][frag_idx] = chunk
    if all(part is not None for part in REASSEMBLY[frag_id]):
        full_packet = b''.join(REASSEMBLY[frag_id])
        del REASSEMBLY[frag_id]
        return full_packet
    return None
def get_bucket_padding(orig_len, bucket_size):
    pad_len = bucket_size - orig_len
    start = random.randint(0, 1536 - pad_len)
    return PRECOMP_RANDOM_POOL[start:start + pad_len]
def select_bucket(length):
    BUCKETS = [64, 256, 512, 1536]
    for b in BUCKETS:
        if length <= b:
            return b
    return BUCKETS[-1]
def recv_exact(sock, n):
    buf = bytearray()
    while len(buf) < n:
        chunk = sock.recv(n - len(buf))
        if not chunk:
            raise SystemExit("Connection closed")
        buf.extend(chunk)
    return bytes(buf)
```

```
def reader(tun, sock, data_key):
    aead = ChaCha20Poly1305(data_key)
    seq = 0
    while True:
        data = os.read(tun, BUF_SIZE)
        if not data:
            continue
        try:
            orig_len = len(data)
            bucket = select_bucket(orig_len)
            padding = get_bucket_padding(orig_len + 4, bucket)
            prefix = struct.pack(">i", orig_len)
            payload = prefix + data + padding
            nonce = os.urandom(NONCE_LEN)
            ciphertext = aead.encrypt(nonce, payload, b"")
            pkt_arr, seq = fragmentor(ciphertext, seq, nonce)
            for pkt in pkt_arr:
                sock.send(pkt)
        except Exception as e:
            print(f"[!] Reader Send error: {e}")
            break
def writer(tun, sock, data_key):
    aead = ChaCha20Poly1305(data_key)
    buf = b""
    while True:
```

```
try:
    incoming = sock.recv(2048)
    if not incoming:
        print("[!] Writer: socket closed by peer")
        break
    buf += incoming
    while len(buf) >= HEADER_LEN:
        flags = buf[0]
        seq, chunk_len = struct.unpack("<Q I", buf[1:13])
        pos = 13
        if flags & FRAG_FLAG:
            if len(buf) < 13 + FRAG_EXT_LEN:
                break
            total_ct_len, frag_id = struct.unpack("<I I", buf[pos:pos+8])
            frag_idx, frag_cnt = struct.unpack("<H H", buf[pos+8:pos+12])
            pos += FRAG_EXT_LEN
        else:
            frag_cnt = 1
            frag_idx = 0
            frag_id = 0
        nonce = buf[pos : pos + NONCE_LEN]
        pos += NONCE_LEN
        total_needed = pos + chunk_len
        if len(buf) < total_needed:
            break
        ct_chunk = buf[pos : total_needed]
        buf = buf[total_needed:]
```

```
if frag_cnt > 1:
    full = defragmentor(frag_id, frag_cnt, frag_idx, ct_chunk)
    if full is None:
        continue
    ciphertext = full
else:
    ciphertext = ct_chunk
if seq in SEQ_BLKLST:
    continue
decoded = aead.decrypt(nonce, ciphertext, b"JORMUNGANDR-V1")
if not decoded:
    continue
if len(decoded) < 4:
    continue
body_len = struct.unpack(">i", decoded[:4])[0]
if body_len <= 0 or body_len > 1536:
    continue
if decoded[4] >> 4 != 4 and decoded[4] >> 4 != 6:
    continue
SEQ_BLKLST.add(seq)
try:
    packet = decoded[4 : 4 + body_len]
    if packet[0] >> 4 not in (4, 6):
        continue
    os.write(tun, packet)
except Exception as e:
```

```
            print(f"[!]TUN write failed: {e}")
        except Exception as e:
            return False
#End Protocol Functions
```

5.1 Purpose

The purpose of the Protocol Functions section is to define the mechanisms by which encrypted data is framed, fragmented, transmitted, received, authenticated, reassembled, and injected into the virtual network interface.

This section implements the core data transport behavior of the VPN system and operates above the operating system interface layer while remaining below the cryptographic key management layer.

5.2 Discussion

5.2.1 Packet Fragmentation and Reassembly

Functions:

- *fragmentor()*
- *defragmentor()*

These functions are responsible for dividing encrypted payloads into transport-safe chunks and reconstructing them upon receipt. Fragmentation occurs only after encryption to preserve cryptographic integrity.

Reassembly state is maintained in memory and keyed by fragment identifiers. Partial fragments are buffered until all expected fragments are received or discarded.

5.2.2 Connection Discovery and Synchronization

Function:

- *listen()*

This function provides a minimal synchronization mechanism used during session establishment. It allows one peer to learn the network address of the remote endpoint prior to initiating key exchange.

This mechanism is not used for data transport and does not carry cryptographic material.

5.2.3 Traffic Padding and Size Obfuscation

Functions:

- *select_bucket()*
- *get_bucket_padding()*

These functions implement deterministic bucket-based padding of plaintext payloads prior to encryption. Padding is intended to reduce information leakage related to packet size variability.

Padding material is drawn from a pre-generated random pool and does not introduce cryptographic dependency on runtime entropy generation.

5.2.4 Reliable Byte Reception

Function:

- *recv_exact()*

This function provides a utility mechanism for receiving an exact number of bytes from a stream-oriented socket. It is used exclusively during control-plane and key-exchange operations.

5.2.5 Encrypted Data Transmission

Function:

- *reader()*

The reader function is responsible for reading raw packets from the TUN interface, applying padding, encrypting the payload, fragmenting the resulting ciphertext, and transmitting the fragments to the peer.

This function operates continuously and assumes exclusive write access to the transport socket.

5.2.6 Encrypted Data Reception

Function:

- *writer()*

The writer function is responsible for receiving encrypted packet fragments from the transport socket, reassembling ciphertext when required, decrypting payloads, validating packet structure, and writing valid packets into the TUN interface.

Packets failing structural or cryptographic validation are silently discarded.

6. Key Exchange

```
#Begin Key Exchange

def derive_keys(raw_shared: bytes, salt: bytes):

    def hk(info: bytes, ln: int):

        return HKDF(algorithm=hashes.SHA256(), length=ln, salt=salt,
info=info).derive(raw_shared)

    return (

        hk(b"VPN handshake v2", 32),

        hk(b"VPN data channel v2", 32),

    )


def ml_kem_client(peer_ip, key_port, cert_data):

    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as conn:

        conn.settimeout(66)

        while True:

            try:

                conn.connect((peer_ip, key_port))

                break

            except:

                print(".", end="")

        else:

            raise SystemExit("Failed to connect to peer for key exchange.")

        with oqs.KeyEncapsulation("Kyber1024") as kem:

            public_key = kem.generate_keypair()

            conn.sendall(struct.pack("<I", len(public_key)))

            conn.sendall(public_key)

            ct_len = struct.unpack("<I", recv_exact(conn, 4))[0]

            ct = recv_exact(conn, ct_len)
```

```
        raw_shared = kem.decap_secret(ct)
    salt = os.urandom(16)
    handshake_key, data_key = derive_keys(raw_shared, salt)
    data_key_sig_b64 = cert_signer(cert_data, data_key)
    aad_bytes = os.urandom(16)
    meta = {
        "DataKeySig": data_key_sig_b64,
        "aad_b64": base64.b64encode(aad_bytes).decode("utf-8")
    }
    meta_bytes = json.dumps(meta).encode("utf-8")
    aead = ChaCha20Poly1305(handshake_key)
    nonce = os.urandom(NONCE_LEN)
    ct_meta = aead.encrypt(nonce, meta_bytes, b"")
    conn.sendall(salt + nonce + struct.pack("<I", len(ct_meta)) + ct_meta)
    return {"data_key": data_key, "aad_bytes": aad_bytes}


def ml_kem_server(key_port, pub_keys):
  with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as conn:
    conn.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    conn.settimeout(666)
    try:
        conn.bind(('0.0.0.0', key_port))
    except:
        raise SystemExit("Failed to bind to port for key exchange.")
    conn.listen(1)
    conn, addr = conn.accept()
    peer_ip = addr[0]
```

```
        print(f"[+] Connection request from {peer_ip}")
        conn.settimeout(20)
        with conn, oqs.KeyEncapsulation("Kyber1024") as kem:
            pk_len = struct.unpack("<I", recv_exact(conn, 4))[0]
            server_pk = recv_exact(conn, pk_len)
            ct, raw_shared = kem.encap_secret(server_pk)
            conn.sendall(struct.pack("<I", len(ct)))
            conn.sendall(ct)
            head = recv_exact(conn, 16 + NONCE_LEN)
            salt, nonce = head[:16], head[16:]
            clen = struct.unpack("<I", recv_exact(conn, 4))[0]
            ct_meta = recv_exact(conn, clen)
            handshake_key, data_key = derive_keys(raw_shared, salt)
            meta_bytes = ChaCha20Poly1305(handshake_key).decrypt(nonce, ct_meta, b"")
            meta = json.loads(meta_bytes.decode("utf-8"))
            data_key_sig_b64 = meta["DataKeySig"]
            aad_bytes = base64.b64decode(meta["aad_b64"].encode("utf-8"))
            data_key_sig = base64.b64decode(data_key_sig_b64)
            ok = cert_checker(pub_keys, data_key_sig, data_key)
            if ok:
                return {"peer": peer_ip, "data_key": data_key, "aad_bytes": aad_bytes}
            else:
                return False
#End Key Exchange
```

6.1 Purpose

The purpose of the Key Exchange section is to securely derive shared session keys between peers using post-quantum key encapsulation mechanisms.

This section establishes cryptographic material used exclusively for subsequent encrypted data transport.

6.2 Discussion

6.2.1 Key Derivation

Function:

- *derive_keys()*

This function derives independent cryptographic keys for control and data channels from a shared secret using HKDF.

6.2.2 Post-Quantum Key Encapsulation

Functions:

- *ml_kem_server ()*
- *ml_kem_client ()*

These functions implement a Kyber1024-based key encapsulation exchange over a TCP control channel. The exchange results in a shared secret known only to the participating peers.

Session metadata is encrypted using a derived handshake key prior to transmission.

7. Certificate System

```
#Begin Certificate System
def dilithium_generator():
    with oqs.Signature("ML-DSA-65") as sig:
        pub = sig.generate_keypair()
        priv = sig.export_secret_key()
        return priv, pub


def cert_signer(private_key_bytes: bytes, msg) -> str:
    with oqs.Signature("ML-DSA-65", secret_key=private_key_bytes) as sig:
        try:
            signature = sig.sign_with_ctx_str(msg, b"SNEK-V1")
        except AttributeError:
            signature = sig.sign(msg)
    return base64.b64encode(signature).decode("utf-8")


def cert_checker(pubkeys: list[str], data_sig: bytes, msg: str) -> bool:
    for pk_path in pubkeys:
        print(f"Trying key: {pk_path}")
        pk_bytes = open(pk_path, "rb").read()


        with oqs.Signature("ML-DSA-65") as sig:
            try:
                if sig.verify_with_ctx_str(msg, data_sig, b"SNEK-V1", pk_bytes):
                    print("OK.")
                    return True
            except AttributeError:
```

```
            try:
                if sig.verify(data_sig, msg, pk_bytes):
                    print("OK.")
                    return True
            except Exception:
                pass
        print("Fail.")
    return False
#End Certificate System
```

7.1 Purpose

The purpose of the Certificate System section is to establish peer identity verification using post-quantum digital signatures.

This system provides explicit authorization control over which peers are permitted to establish VPN sessions.

7.2 Discussion

7.2.1 Certificate Generation

Function:

- dilithium_generator()

This function generates a post-quantum signature keypair for peer identity.

7.2.2 Message Signing

Function:

- cert_signer()

This function produces a digital signature over session key material using a private signing key.

7.2.3 Signature Verification

Function:

- cert_checker()

This function verifies that a provided signature matches one of a set of authorized public keys. Successful verification indicates peer authorization.

8. Role Functions

```
#Begin Role Functions
def client_mode(tun, port, data_key, peer_ip):
    with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
        s.connect((peer_ip, port))
        s.settimeout(None)
        time.sleep(0.5)
        s.send(b"\xff")
        print(f"[+] Connected to {peer_ip}")
        t1 = threading.Thread(target=reader, args=(tun, s, data_key), daemon=True)
        t2 = threading.Thread(target=writer, args=(tun, s, data_key), daemon=True)
        t1.start()
        t2.start()
        t1.join()
        t2.join()


def server_mode(tun, port, data_key, aad_bytes):
    with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
        s.bind(("0.0.0.0", port))
        s.settimeout(30)
        print(f"[+] Waiting for connection on port {port}...")
        try:
            data, addr = s.recvfrom(1)
        except Exception:
            raise SystemExit("Connection failed! (bad auth?)")
        s.connect(addr)
        print("[+] Connected.")
```

```
    with ThreadPoolExecutor(max_workers=2) as executor:
        future_reader = executor.submit(reader, tun, s, data_key)
        future_writer = executor.submit(writer, tun, s, data_key)
        result = future_writer.result()
        if result is False:
            return True
        future_reader.cancel()
#End Role Functions
```

8.1 Purpose

The purpose of the Role Functions section is to define the behavioral differences between client and server execution modes.

These functions coordinate key exchange, session initiation, and concurrent data transport.

8.2 Discussion

8.2.1 Client Mode

Function:

- client_mode()

This function initializes the VPN system in client mode and reaches out to the peer before beginning encrypted data exchange.

8.2.2 Server Mode

Function:

- server_mode()

This function initializes the VPN system in server mode and listens for encrypted communication with an authorized peer.

9. Main Execution

```
#Begin Main

def main():

    parser = argparse.ArgumentParser(description="JormungandrVPN V1")

    parser.add_argument("-c", "--connect", help="Peer IP or DNS")

    parser.add_argument("-l", "--listen", action="store_true")

    parser.add_argument("-kp", "--keyport", help="Port to use for key exchange.")

    parser.add_argument("-dp", "--dataport", help="Port to use for data exchange.")

    parser.add_argument("-x", "--cert", help="Path to dilithium private key or to directory with dilithium public keys.")

    parser.add_argument("-g","--gencert",help="Path to folder for storing the generated dilithium certificate pair.")

    args = parser.parse_args()

    if args.gencert:

        private, public = dilithium_generator()

        pubfile = str(args.gencert) + "/snek-cert.pub"

        privfile = str(args.gencert) + "/snek-cert.priv"

        open(pubfile, "wb").write(public)

        open(privfile, "wb").write(private)

        raise SystemExit("Certificates written")

    if args.keyport:

        key_port = int(args.keyport)

    else:

        key_port = 443

    if args.dataport:

        data_port = int(args.dataport)

    else:

        data_port = 10666
```

```
    tun = create_tun()
    set_tun_mtu(IFACE_NAME, 1536)
    if args.connect:
        if not args.cert:
            raise SystemExit("Must specifiy a certificate for client mode!")
        print(f"[+] Connecting to {args.connect}:{key_port}...")
        cert_data = open(str(args.cert), "rb").read()
        session_keys = ml_kem_client(args.connect, key_port, cert_data)
        data_key = session_keys["data_key"]
        aad_bytes = session_keys["aad_bytes"]
        client_mode(tun, data_port, data_key, args.connect)
    else:
        while True:
            if not args.cert:
                raise SystemExit("Must specifiy directory containing approved public keys for server
mode!")
            pub_keys = [os.path.join(args.cert, f) for f in os.listdir(args.cert) if f.endswith(".pub")]
            authorized = ml_kem_server(key_port, pub_keys)
            if authorized:
                currentpeer = authorized["peer"]
                data_key = authorized["data_key"]
                aad_bytes = authorized["aad_bytes"]
                server_mode(tun, data_port, data_key, aad_bytes)
            else:
                print(f"Connection not authorized. {currentpeer}")
#End Main
```

9.1 Purpose

The purpose of the Main section is to coordinate argument parsing, execution mode selection, and system initialization.

9.2 Discussion

Function:

- main()

This function parses command-line arguments, initializes system resources, selects certificate generation, client or server execution paths, and initiates session handling.

10. Program Initialization

```
#Begin Init
if __name__ == "__main__":
    main()
#End Init
```

10.1 Purpose

The purpose of the Program Initialization section is to define the entry point of the VPN system.

10.2 Discussion

The __main__ guard ensures that execution occurs only when the module is run as a standalone program and not when imported as a library.