Nautilus Public Key Infrastructure Code Documentation

1. Overview

The `identity/pki/src` module provides a flexible and extensible Public Key Infrastructure (PKI) framework for our chat application. It incorporates post-quantum cryptographic (PQC) algorithms to ensure secure communication and data integrity while also supporting classical algorithms like RSA for interoperability and performance considerations.

The PKI framework is structured around key traits:

- **`PKITraits`** – Defines core PKI functionalities like key generation, signing, and verification.
- **`KeyExchange`** – Manages secure exchange of session keys.
- **`KeySerialization`** – Standardizes methods for serializing and deserializing keys.

This module includes PQC implementations such as:

- **Dilithium** – Signature scheme for message authentication.
- **Kyber** – Key Encapsulation Mechanism (KEM) for secure key exchange.
- **Falcon** – Lattice-based signature scheme.
- **SPHINCS+** – Hash-based stateless signature scheme.

2. Key Components

2.1. `pki_traits.rs`

Defines the `PKITraits` trait, which provides a unified interface for cryptographic operations:

- **`fn generate_key_pair()`** – Generates a new key pair.
- **`fn sign(&self, data: &[u8])`** – Signs data using the private key.
- **`fn verify(&self, data: &[u8], signature: &[u8])`** – Verifies the authenticity of a signature.
- **`fn get_public_key_raw_bytes(&self)`** – Retrieves the raw byte representation of the public key.
- **`fn key_type()`** – Identifies the cryptographic algorithm.

2.2. `pki_error.rs`

Defines the `PKIError` enum for handling errors in PKI operations. Key error variants include:

- **`KeyPairGenerationError(String)`** – Indicates issues in key pair generation.
- **`SigningError(String)`** – Captures errors in signing operations.
- **`VerificationError(String)`** – Reports verification failures.
- **`UnsupportedOperation(String)`** – Flags operations unsupported by a cryptographic scheme.
- **`EncodingError(String) / DecodingError(String)`** – Handles serialization and deserialization errors.
- **`KeyExchangeError(String)`** – Captures key exchange failures.

2.3. `key_serde_trait.rs`

Defines the `KeySerialization` trait for handling key serialization and deserialization:

- **`fn to_bytes(&self) -> Vec<u8>`** – Converts a key to its byte representation.
- **`fn from_bytes(bytes: &[u8]) -> Result<Self, PKIError>`** – Reconstructs a key from bytes.

2.4. `key_exchange.rs`

Defines the `KeyExchange` trait for secure key exchange operations:

- **`fn encapsulate(&self) -> (Vec<u8>, Vec<u8>)`** – Generates a shared secret and encapsulated ciphertext.
- **`fn decapsulate(&self, ciphertext: &[u8]) -> Result<Vec<u8>, PKIError>`** – Extracts the shared secret from ciphertext.
- **`fn key_exchange_type(&self) -> &str`** – Returns the key exchange mechanism type.

2.5. `cipher_suite.rs`

Defines the `CipherSuite` enum, representing supported cipher suites in the Nautilus Handshake Protocol:

- **`pub fn name(&self) -> &str`** – Returns the cipher suite name.
- **`pub fn is_supported(&self) -> bool`** – Checks if a cipher suite is enabled.
- **`pub fn supported_signature_schemes()`** – Lists available signature schemes.
- **`pub fn supported_kem_schemes()`** – Lists supported key encapsulation mechanisms.

3. Cryptographic Implementations

3.6. Dilithium (`dilithium_keypair.rs`)

A post-quantum signature scheme for secure message authentication.

- **`fn generate_key_pair()`** – Generates a `DilithiumKeyPair`.
- **`fn sign(&self, data: &[u8])`** – Creates a digital signature.
- **`fn verify(&self, data: &[u8], signature: &[u8])`** – Verifies a signature.
- **`fn to_bytes(&self) / fn from_bytes(bytes: &[u8])`** – Serializes and deserializes key pairs.

3.7. Falcon (`falcon_keypair.rs`)

A lattice-based post-quantum signature scheme.

- **`fn generate_key_pair()`** – Generates a `FalconKeyPair`.
- **`fn sign(&self, data: &[u8])`** – Signs data using the private key.

- `fn verify(&self, data: &[u8], signature: &[u8])` – Verifies a signature.

3.8. RSA (`rsa_keypair.rs`)

A classical cryptographic algorithm used for signatures and key exchange.

- `fn generate_key_pair()` – Generates an `RSAkeyPair` .
- `fn sign(&self, data: &[u8])` – Signs data.
- `fn verify(&self, data: &[u8], signature: &[u8])` – Verifies signatures.
- `fn encapsulate()` / `fn decapsulate()` – Implements RSA-OAEP for key exchange.

3.9. SECP256K1 (`secp256k1_keypair.rs`)

An elliptic curve cryptographic scheme used for signing and key exchange.

- `fn generate_key_pair()` – Generates an `SECP256K1KeyPair` .
- `fn sign(&self, data: &[u8])` – Signs data.
- `fn verify(&self, data: &[u8], signature: &[u8])` – Verifies signatures.
- `fn encapsulate()` / `fn decapsulate()` – Implements ECDH key exchange.

3.10. Kyber (`kyber_keypair.rs`)

A post-quantum key encapsulation mechanism (KEM) used for secure key exchange.

- `fn generate_key_pair()` – Generates a `KyberKeyPair` .
- `fn encapsulate()` / `fn decapsulate()` – Performs key exchange.

3.11. Ed25519 (`ed25519_keypair.rs`)

An elliptic curve-based digital signature scheme.

- `fn generate_key_pair()` – Generates an `Ed25519KeyPair` .
- `fn sign(&self, data: &[u8])` – Signs data.
- `fn verify(&self, data: &[u8], signature: &[u8])` – Verifies a signature.
- `fn encapsulate()` / `fn decapsulate()` – Implements X25519 key exchange.

3.12. ECDSA (`ecdsa_keypair.rs`)

An elliptic curve digital signature algorithm supporting signing and ECDH key exchange.

- `fn generate_key_pair()` – Generates an `ECDSAKeyPair` .
- `fn encapsulate()` / `fn decapsulate()` – Implements ECDH for key exchange.

3.13. SPHINCS+ (`spincs_keypair.rs`)

A hash-based, stateless post-quantum signature scheme.

- `fn generate_key_pair()` – Generates an `SPHINCSKeyPair` .
- `fn sign(&self, data: &[u8])` – Creates a SPHINCS+ signature.
- `fn verify(&self, data: &[u8], signature: &[u8])` – Verifies a SPHINCS+ signature.

4. Conclusion

This PKI module ensures forward secrecy and quantum-resistant security for our chat application while maintaining interoperability with classical cryptographic systems. By implementing a modular and extensible framework, it facilitates secure communication and data integrity in a post-quantum era.