```rust
// src/tee_execution.rs
//! TEE-Attested Proof Execution for NERV
//!
//! This module provides secure, attested proof generation using Trusted Execution Environments (TEEs).
//! It integrates seamlessly with the hybrid Halo2 + Nova system in `halo2_nova_integration.rs`.
//!
//! Key Features:
//! - Proof generation (Halo2 client proofs and Nova folding) runs inside a simulated/hardware TEE.
//! - Remote attestation: Enclave measurement + quote signed with Dilithium (post-quantum secure).
//! - Key management: Proving keys sealed; Kyber768 for secure key exchange/derivation if needed.
//! - Attested outputs: Proof + Dilithium signature over (proof || public_inputs || measurement).
//! - Verification: Check Dilithium signature + enclave measurement against trusted MRENCLAVE.
//!
//! Design:
//! - Client-side: Optional TEE (e.g., ARM TrustZone on mobile) for private witness protection.
//! - Validator-side: Mandatory TEE (SGX/SEV-SNP) for large Nova folding and dispute proofs.
//! - Post-quantum: Dilithium3 for signatures, Kyber768 for encapsulation (e.g., sealed data).
//! - Simulation mode: For development/testing (real hardware crates can replace stubs).
//!
//! Dependencies (add to Cargo.toml):
//! pqcrypto-dilithium = "0.5"   # Dilithium3 signatures
//! pqcrypto-kyber = "0.5"       # Kyber768 KEM
//! pqcrypto-traits = "0.3"
//! rand = "0.8"
//! blake3 = "1.5"
//! bincode = "2.0"
//! sha3 = "0.10"              # For measurement hashing
//!
//! For real SGX: Add `fortanix-sgx-tools` or `apache-teaclave-sgx-sdk` (hardware mode).
//! This code uses simulation stubs; replace with real enclave calls in production.

use crate::halo2_nova_integration::{UnifiedProofManager, ClientDeltaCircuit, ValidatorFoldingCircuit,
CompressedSNARK};
use pqcrypto_dilithium::dilithium3::{
    keypair as dilithium_keypair,
    sign as dilithium_sign,
    verify as dilithium_verify,
    PublicKey,
    SecretKey,
    SignedMessage,
};
use pqcrypto_kyber::kyber768::{
    keypair as kyber_keypair,
    encapsulate,
    decapsulate,
    Ciphertext,
    SharedSecret,
};
use pqcrypto_traits::sign::{DetachedSignature, PublicKey as PqPublicKey};
```

```rust
use rand::rngs::OsRng;
use blake3::Hasher;
use bincode::{serialize, deserialize};
use sha3::{Digest, Sha3_256};
use std::fmt::Debug;

// Simulated enclave measurement (MRENCLAVE hash of code + data)
// In real SGX: Use sgx_report and remote attestation (IAS/PCA).
pub const TRUSTED_MRENCLAVE: [u8; 32] = [0x42; 32]; // Replace with actual in production

/// TEE Prover: Generates attested proofs
#[derive(Clone)]
pub struct TeeProver {
    /// Dilithium secret key (sealed inside enclave)
    dilithium_sk: SecretKey,
    /// Dilithium public key (public for verification)
    dilithium_pk: PublicKey,
    /// Kyber keypair for secure sealing/unsealing
    kyber_sk: pqcrypto_kyber::kyber768::SecretKey,
    kyber_pk: pqcrypto_kyber::kyber768::PublicKey,
    /// Enclave measurement (fixed for this code version)
    measurement: [u8; 32],
    /// Wrapped proof manager (state sealed)
    proof_manager: UnifiedProofManager,
}

impl TeeProver {
    /// Initialize enclave prover (run inside TEE)
    pub fn new() -> Self {
        // In real TEE: Load/seal keys if not present
        let (dilithium_pk, dilithium_sk) = dilithium_key_pair();
        let (kyber_pk, kyber_sk) = kyber_key_pair();

        // Compute measurement (hash of code + constants)
        let mut hasher = Hasher::new();
        hasher.update(b"NERV_TEE_PROVER_CODE_V1"); // In real: include enclave binary
        let measurement = hasher.finalize().into();

        Self {
            dilithium_sk,
            dilithium_pk,
            kyber_sk,
            kyber_pk,
            measurement,
            proof_manager: UnifiedProofManager::new(18),
        }
    }

    /// Generate attested Halo2 client proof (inside TEE)
```

```rust
pub fn prove_client_attested(
    &mut self,
    circuit: ClientDeltaCircuit,
) -> Result<AttestedProof, String> {
    // Generate proof (witness protected in TEE)
    let proof = self.proof_manager.prove_client(circuit);

    // Serialize data to sign
    let message = serialize(&(proof.clone(), self.measurement)).unwrap();

    // Dilithium sign
    let signature = dilithium_sign(&message, &self.dilithium_sk);

    Ok(AttestedProof {
        proof,
        public_inputs: vec![], // Fill if needed
        measurement: self.measurement,
        signature,
        dilithium_pk: self.dilithium_pk.clone(),
    })
}

/// Fold validator batch + attest (Nova step)
pub fn fold_and_attest(
    &mut self,
    batch_circuit: ValidatorFoldingCircuit<pasta_curves::pallas::Point>,
) -> Result<AttestedFold, String> {
    self.proof_manager.fold_validator_batch(batch_circuit);

    let intermediate_state = serialize(&self.proof_manager.nova_manager.current_snark).unwrap();
    let message = serialize(&(intermediate_state, self.measurement)).unwrap();
    let signature = dilithium_sign(&message, &self.dilithium_sk);

    Ok(AttestedFold {
        intermediate_state,
        measurement: self.measurement,
        signature,
        dilithium_pk: self.dilithium_pk.clone(),
    })
}

/// Final compressed Nova SNARK + attestation
pub fn finalize_attested(&self) -> Result<AttestedCompressedSnark, String> {
    let compressed = self.proof_manager.final_validator_proof();
    let serialized = bincode::serialize(&compressed).unwrap();

    let message = serialize(&(serialized.clone(), self.measurement)).unwrap();
    let signature = dilithium_sign(&message, &self.dilithium_sk);
```

```
        Ok(AttestedCompressedSnark {
            compressed_snark: serialized,
            measurement: self.measurement,
            signature,
            dilithium_pk: self.dilithium_pk.clone(),
        })
    }

    /// Seal data with Kyber (e.g., for persistent storage)
    pub fn seal_with_kyber(&self, data: &[u8]) -> (Ciphertext, SharedSecret) {
        encapsulate(&self.kyber_pk, data)
    }

    /// Unseal (inside TEE only)
    pub fn unseal_with_kyber(&self, ciphertext: Ciphertext, shared: SharedSecret) -> Vec<u8> {
        decapsulate(&ciphertext, &self.kyber_sk, &shared)
    }

    /// Get public keys for remote attestation setup
    pub fn public_keys(&self) -> (PublicKey, pqcrypto_kyber::kyber768::PublicKey) {
        (self.dilithium_pk.clone(), self.kyber_pk.clone())
    }
}

/// Attested proof structures
#[derive(Clone, Debug)]
pub struct AttestedProof {
    pub proof: Vec<u8>,
    pub public_inputs: Vec<Fp>,
    pub measurement: [u8; 32],
    pub signature: DetachedSignature,
    pub dilithium_pk: PublicKey,
}

#[derive(Clone, Debug)]
pub struct AttestedFold {
    pub intermediate_state: Vec<u8>,
    pub measurement: [u8; 32],
    pub signature: DetachedSignature,
    pub dilithium_pk: PublicKey,
}

#[derive(Clone, Debug)]
pub struct AttestedCompressedSnark {
    pub compressed_snark: Vec<u8>,
    pub measurement: [u8; 32],
    pub signature: DetachedSignature,
    pub dilithium_pk: PublicKey,
}
```

```rust
/// TEE Verifier: Checks attestation + proof
pub struct TeeVerifier {
    trusted_measurement: [u8; 32],
}

impl TeeVerifier {
    pub fn new() -> Self {
        Self { trusted_measurement: TRUSTED_MRENCLAVE }
    }

    /// Verify attested client proof
    pub fn verify_client_proof(&self, attested: &AttestedProof) -> Result<bool, String> {
        if attested.measurement != self.trusted_measurement {
            return Err("Untrusted enclave measurement".to_string());
        }

        let message = serialize(&(attested.proof.clone(), attested.measurement)).unwrap();
        dilithium_verify(&message, &attested.signature, &attested.dilithium_pk)
            .map_err(|_| "Dilithium verification failed".to_string())
    }

    /// Verify final compressed SNARK
    pub fn verify_compressed_snark(&self, attested: &AttestedCompressedSnark) -> Result<bool, String>
{
        if attested.measurement != self.trusted_measurement {
            return Err("Untrusted measurement".to_string());
        }

        let message = serialize(&(attested.compressed_snark.clone(), attested.measurement)).unwrap();
        dilithium_verify(&message, &attested.signature, &attested.dilithium_pk)
            .map_err(|_| "Signature invalid".to_string())?;

        // Additionally verify the SNARK itself (using Nova params)
        // let snark: CompressedSNARK<_> = deserialize(&attested.compressed_snark).unwrap();
        // snark.verify(&pp, &public_inputs)
        Ok(true)
    }
}

// ========================
// Integration Example
// ========================

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
```

```
    fn test_tee_attested_client_proof() {
        let mut prover = TeeProver::new();
        let circuit = ClientDeltaCircuit::new([1; 32], [2; 32], FixedPoint::from_f64(10.0).unwrap());

        let attested = prover.prove_client_attested(circuit).unwrap();

        let verifier = TeeVerifier::new();
        assert!(verifier.verify_client_proof(&attested).is_ok());
    }

    #[test]
    fn test_kyber_sealing() {
        let prover = TeeProver::new();
        let data = b"sensitive witness data";

        let (ciphertext, shared) = prover.seal_with_kyber(data);
        // In real TEE: Store ciphertext externally, shared discarded

        // Simulate unseal (only possible inside same enclave)
        let unsealed = prover.unseal_with_kyber(ciphertext, shared);
        assert_eq!(unsealed, data);
    }
}
```

Integration Notes

- **Seamless with Halo2 + Nova**: TeeProver wraps UnifiedProofManager. Proofs/folds are generated identically but attested.
- **Dilithium**: Signs (proof/state + measurement) for non-repudiation and authenticity.
- **Kyber768**: For sealing sensitive data (e.g., proving keys, witnesses) to persistent storage.
- **Attestation**: Measurement check ensures code integrity (real SGX would use IAS quote verification).
- **Validator Usage**: Replace direct proof_manager calls with tee_prover.prove_* in ValidatorNode.
- **Client Usage**: Optional for mobile (TrustZone); protects private commitments.
- **Production**: Replace simulation with real enclave (e.g., #[sgx_enclave] macros).