Key Storage Module Documentation

1. Overview

The Key Storage module provides a flexible and secure framework for managing cryptographic keys across various storage backends, including in-memory, file-based, Linux-based, Windows-based, and cloud-based solutions. Each storage implementation adheres to the `KeyStorage` trait, which defines operations such as saving, loading, removing, listing, and retrieving metadata for keys. A structured error-handling approach via the `KeyStorageError` enum ensures robust handling of potential failures.

Key features of the Key Storage module include:

- Support for multiple storage backends, including local filesystem, cloud (AWS KMS), and OS-specific key management solutions.
- Standardized serialization and deserialization of cryptographic keys using the `FileFormat` trait.
- Secure encryption and decryption of stored keys for enhanced protection.
- Comprehensive testing to validate correctness and resilience across all implementations.
- Unified support for both classical and post-quantum cryptographic (PQC) key storage.

2. Core Components

2.1. `key_storage_trait.rs`

Defines the `KeyStorage` trait, providing a consistent interface for key management operations:

- **`fn initialize(&self, config: Option<&str>)`** – Configures the storage backend.
- **`fn save StoredType, location: &str, encrypt: bool`** – Stores a key with optional encryption.
- **`fn load(&self, location: &str, decrypt: bool)`** – Retrieves and optionally decrypts a stored key.
- **`fn remove(&self, location: &str)`** – Deletes a key from storage.
- **`fn list(&self)`** – Lists all stored keys.
- **`fn metadata(&self, location: &str)`** – Retrieves metadata for a stored key.

Additionally, the `KeyMetadata` struct provides tracking information for stored keys, including timestamps, size, and storage location.

2.2. `key_storage_error.rs`

Defines the `KeyStorageError` enum to standardize error handling across storage implementations. Key variants include:

- **`SaveError(String)`** – Indicates failures during key storage.
- **`LoadError(String)`** – Captures issues encountered while retrieving keys.
- **`RemoveError(String)`** – Represents errors in key deletion operations.
- **`EncryptionError(String)`** – Handles encryption and decryption failures.
- **`BackendError(String)`** – Reports storage backend-specific errors.

3. Storage Implementations

3.3. File-Based Storage (`file_storage.rs`)

Implements a filesystem-based `KeyStorage` mechanism, allowing secure storage and retrieval of keys in JSON or PEM formats. It includes:

- **`fn save`** – Serializes and writes keys to the filesystem.
- **`fn load`** – Reads and deserializes stored keys.
- **`fn remove`** – Deletes key files securely.
- **`fn list`** – Enumerates stored keys within the storage directory.
- **`fn metadata`** – Retrieves file metadata, such as size and creation date.

3.4. In-Memory Storage (`in_memory_key_storage.rs`)

Provides a non-persistent, lightweight key storage implementation using a `HashMap`. Suitable for testing and ephemeral key management.

- **Thread-safe implementation using `Arc<Mutex<...>>`.**
- **Supports all key operations except metadata retrieval**.

3.5. Windows Storage (`windows_storage.rs`)

Implements Windows-specific key storage with:

- **Credential Manager (`windows_key_ring_storage.rs`)** – Securely stores and retrieves keys using the Windows Credential Manager.
- **Trusted Security Module (`windows_tsm_storage.rs`)** – Encrypts and stores keys using Windows DPAPI for enhanced protection.

3.6. Linux Storage (`linux_storage.rs`)

Leverages the `linux-keyutils` library for managing cryptographic keys via user-defined or session-based keyrings.

- **Secure, OS-integrated keyring storage**.
- **Supports operations such as key retrieval, removal, and listing**.

3.7. Cloud Storage (`cloud_storage.rs`)

Integrates with Amazon Web Services (AWS) Key Management Service (KMS) for cloud-based key storage.

- **Manages key creation, retrieval, and deletion via AWS KMS APIs**.
- **Implements secure encryption and metadata retrieval**.

4. File Formats (`file_format_trait.rs`)

Defines the `FileFormat` trait for serializing and deserializing keys into specific formats. Supported formats:

- **JSON (`json_formatter.rs`)** – Serializes keys in a structured JSON format.

- **PEM (`pem_formatter.rs`)** – Encodes keys in Privacy-Enhanced Mail (PEM) format.

5. Conclusion

This module provides a unified and extensible approach to cryptographic key storage, supporting various backends tailored to different security and operational requirements. The integration of multiple storage methods ensures compatibility across environments, making it a robust solution for managing cryptographic keys in both traditional and post-quantum settings.