EchoPulse Test Vector Documentation (A1)

Version: 1.0

Date: May 11, 2025

Author: EchoPulse Initiative

This document outlines the structure, purpose, and usage of symbolic test vectors designed to verify the correctness of the key encapsulation process within the EchoPulse Key Encapsulation Mechanism (KEM). These vectors serve as a fundamental audit base for ensuring the integrity of the symbolic key derivation path.

1. Purpose

The primary purpose of these symbolic test vectors is to:

* **Verify Symbolic Key Derivation:** Demonstrate and validate the sequence of state transitions within the graph $G(V, E)$ initiated by the secret key (SK).

* **Illustrate Random Payload Processing:** Showcase how the random symbol sequence (r) is used to traverse the graph from the public key-derived state to the encapsulation state (v_{enc}).

* **Validate Shared Key Derivation:** Confirm the correct computation of the shared secret key (K) through the cryptographic hash of the encoded final state (v_{enc}) and the random symbol sequence (r).

2. Test Vector Structure

Each test vector within the provided datasets adheres to the following structure:

* **SK (Symbol Array):** A byte array representing the secret key, consisting of a sequence of symbols from Σ (e.g., `[0x1A, 0xF3, 0x0D, ...]`).

* **r (Random Symbol Sequence):** A byte array representing the random symbol sequence used during encapsulation (e.g., `[0xBC, 0x22, 0x91, ...]`). This sequence has a fixed length (e.g., 28 bytes).

* **δ-Path Resolution:** A trace of the state transitions within $G(V, E)$. This includes the sequence of (current state, input symbol) pairs and the resulting next state, starting from the initial private key state (v_{init_priv}) and continuing for the length of SK to reach the private key's terminal state (v_{priv}), and similarly from the initial public key state (v_{init_pub}) using a derived public key symbol sequence to reach the public key's terminal state (v_{pub}), followed by the application of r to reach the encapsulation state (v_{enc}).

* **Hash Input:** The raw byte string formed by the concatenation of the encoded final state (v_{enc}) and the random symbol sequence (r).

* **Shared Key K Output:** The expected 32-byte shared secret key resulting from applying the SHA3-256 hash function to the "Hash Input."

3. Encoding Specification

Consistent encoding rules are applied to ensure unambiguous interpretation of the test vectors:

* **Symbols:** Each symbol is represented as a single raw byte, ranging from `0x00` to `0xFF`.

* **Vertex (v):** Each vertex state within the graph $G(V, E)$ is encoded as a 2-byte unsigned integer in little-endian byte order.

* **Random Symbol Sequence (r):** The sequence r is treated as a fixed-length array of raw bytes (e.g., 28 bytes).

* **Hash Input:** The input to the SHA3-256 hash function is formed by the direct concatenation of the 2-byte little-endian encoding of the final encapsulation state (v_{enc}) followed by the raw byte sequence of r.

4. Hash Function

The cryptographic hash function used for deriving the shared secret key K is SHA3-256. Implementations must use a standard and correctly implemented SHA3-256 library. The output of this function is a 32-byte shared secret key.

5. Vector Validity Conditions

For a test vector to be considered valid:

* **Resolvable δ Transitions:** All specified state transitions ($\delta(v, \sigma)$) along the paths derived from SK and r must be resolvable within the defined graph $G(V, E)$.

* **Valid State Range:** The resulting private key terminal state (v_{priv}) and encapsulation state (v_{enc}) must fall within the valid range of vertex indices defined for $G(V, E)$.

* **Error Handling:** If the application of SK or r leads to an undefined transition or an out-of-range state, the test case should either be marked as an error condition or, if applicable, follow the recovery procedures outlined in Section 6. In some failure cases, retrying with a derived seed or triggering graph mutation might be necessary to achieve a valid encapsulation.

6. Recovery Case Handling

In scenarios where the decapsulation process might fail due to temporary graph desynchronization or session mismatch, the following recovery procedure is defined:

* **Fallback r Derivation:** If the initial decapsulation attempt fails, a fallback random symbol sequence r' should be derived using the following deterministic function:

```
r' = H(session_index || salt || retry_count)
```

where `session_index` is the current session identifier, `salt` is the shared secret salt used for graph mutation, and `retry_count` is an integer starting from 1.

* **Retry Limit:** The decapsulation should be retried with the derived r' up to a maximum of 3 times, incrementing the `retry_count` for each attempt.

* **Graph Remutation:** If decapsulation continues to fail after the retry limit is reached, it indicates a more severe desynchronization, and a graph remutation procedure should be triggered based on the current session index and salt.

Test vectors in the `A3_FailureCase_Handling.docx` file will specifically address these recovery scenarios.

7. Test Vector Files

The symbolic test vectors are provided in the following files:

* **`A2_SK_to_K_DerivationSet.csv`:** This CSV file contains a set of valid test vectors demonstrating successful key derivation from the secret key SK through the encapsulation process to the shared key K. The format will be clearly defined within the file.

* **`A3_FailureCase_Handling.docx`:** This document (Microsoft Word format) contains specific test cases designed to evaluate the protocol's behavior under various failure conditions, including potential desynchronization and the application of the fallback r' derivation.

* **`A4_Hash_Function_Reference.md`:** This Markdown file provides detailed information about the SHA3-256 hash function used, including encoding examples for the input and the expected output format, as well as any relevant API usage notes.

8. Conclusion

These symbolic test vectors form the foundational audit base for verifying the correctness and integrity of the EchoPulse key encapsulation mechanism. They provide concrete examples of the symbolic manipulations, state transitions, and cryptographic hash operations that underpin the protocol's functionality. Adherence to these test vectors is crucial for ensuring compliant and secure implementations of EchoPulse.

Document A1 — Symbolic Vector Integrity Layer — EchoPulse Initiative