# EchoPulse Implementation Notes

Version: 1.0

Date: May 11, 2025

Author: EchoPulse Initiative

This document provides guidance for developers implementing and deploying the EchoPulse Key Encapsulation Mechanism (KEM) in real-world systems. It outlines key considerations for performance, security, and interoperability, targeting developers working at a low-level.

## 1. Transition Function Implementation

The state transition function $\delta(v, \sigma)$ is a critical component of EchoPulse. Its implementation should prioritize both speed and resistance against timing side-channel attacks.

* **Constant-Time Array Lookup:** The recommended approach is to represent $\delta$ using a lookup table (array). Accessing the next state based on the current state `v` and symbol `sigma` should involve a single, predictable memory access. This access pattern must be independent of the values of `v` and `sigma` to prevent timing leaks.

* **Storage Options:**
    * **Fixed-Size Transition Table:** A 2D array `next_state[1024][256]` where `next_state[v][sigma]` directly stores the index of the next state. This offers the simplest and potentially fastest constant-time lookup. However, it requires significant memory (e.g., 256 KB if state indices are 1 byte).
    * **Sparse Matrix Encoding (Considerations):** If the average out-degree of the graph is significantly less than 256, a sparse representation (e.g., an array of structures or linked lists per state, storing (symbol, next_state) pairs) could save memory. However, ensuring constant-time lookup with sparse structures is challenging and requires careful design (e.g., padding each state's representation to a fixed size or using specialized constant-time sparse matrix access techniques). If choosing this, ensure that the lookup mechanism iterates through a fixed number of entries or uses a constant-time search algorithm.

## 2. Memory Management

Efficient memory usage is paramount, especially for embedded systems with limited resources.

* **Recommended Minimum Memory:** For constrained devices, aim for a total RAM footprint in the range of 48–64 KB for the EchoPulse implementation, including the graph representation, key buffers, and working memory. This might necessitate careful selection of graph storage and key/payload sizes.

* **Memory Allocation Breakdown:**

    * **Graph Table:** Allocate a contiguous block of memory to store the chosen representation of the state transition graph. Optimize this based on the target device's memory constraints.

    * **Symbol Buffers:** Reserve separate, fixed-size buffers for the secret key (`SK`, e.g., 28 bytes), the public key (`PK`, e.g., 28 bytes or size of seed + derivation parameters), and the random symbol sequence (`r`, e.g., 28 bytes).

    * **Working Registers:** Allocate a small amount of stack or heap memory for temporary variables used during the KEM operations (e.g., loop counters, intermediate hash values, current state during traversal).

## 3. Graph Mutation Strategy

The deterministic mutation of the state transition graph based on the session index and a shared salt is a key security feature.

* **Precomputation (Recommended for Performance):** On devices with sufficient memory, precompute the mutated graph for a small window of future sessions (e.g., 10-100 sessions) and store them. At the beginning of each session, the implementation retrieves the appropriate graph based on the current session index. This minimizes the computational overhead during the time-critical encapsulation and decapsulation processes.

* **On-Device Computation (Memory Conscious):** For very memory-constrained devices, the mutation function $\mu(G, \text{salt}, \text{session\_index}) \rightarrow G'$ can be computed on demand at the start of each session. Ensure that the mutation algorithm is efficient and

deterministic. The shared salt must be securely stored on both communicating parties. The session index needs to be maintained and synchronized between the parties.

## 4. Hash Function Selection

The cryptographic hash function is used for deriving the shared secret key.

* **Primary Recommendation:** SHA3-256 is the preferred choice due to its strong security properties and resistance against known quantum algorithms. Utilize well-vetted and optimized implementations of SHA3-256.

* **Optional Alternatives (Performance Trade-offs):** On platforms where SHA3-256 performance is a significant bottleneck, consider SHAKE256 or BLAKE2s-256. These might offer better performance in some environments while still providing strong cryptographic security. However, thorough security analysis and benchmarking on the target platform are crucial before opting for an alternative.

## 5. Integration Recommendations

Integrating EchoPulse into existing communication protocols requires careful design of data formats and handshake procedures.

* **TLS-Style Handshake:** For secure communication channels like TLS, EchoPulse can be integrated as a Key Exchange Algorithm. Define new cipher suites that specify EchoPulse for key establishment. The public key ($PK$) would be exchanged during the handshake (e.g., in a ServerKeyExchange or similar message), and the derived shared secret would be used for subsequent symmetric encryption.

* **Key/Payload Export Format:** Define a clear and unambiguous binary format for serializing and deserializing the secret key ($SK$), public key ($PK$), and the ciphertext ($r$). Using structured formats like CBOR (Concise Binary Object Representation) blocks with specific tags for each data element is highly recommended for interoperability and clarity.

* **Certificate Interface (Optional):** For systems employing public key certificates, consider defining how EchoPulse public keys can be included. This might involve defining new object identifiers (OIDs) and structures within X.509 certificates to represent EchoPulse public keys.

## 6. Side-Channel Mitigations

Protecting against side-channel attacks is crucial for the security of EchoPulse, especially in embedded systems.

* **Constant-Time Processing (Mandatory):** Ensure that all core cryptographic operations, particularly the evaluation of the $\delta$ function and the hash function, are implemented using constant-time techniques. This means avoiding data-dependent branches, memory accesses, and execution times.

* **Dummy δ-Transitions (Optional):** As an additional mitigation against timing analysis of the graph traversal, consider inserting a fixed number of dummy state transitions after each real transition. The target states of these dummy transitions should not affect the protocol's outcome but can help to mask the actual path taken through the graph.

* **Avoid Data-Dependent Branching:** Carefully review the implementation of the transition function and any related logic to eliminate any conditional branches whose execution path depends on the values of the symbols or states.

## 7. Interoperability Layers

Standardized data formats are essential for ensuring interoperability between different EchoPulse implementations.

* **CBOR Encoding (Recommended):** CBOR offers a compact and flexible binary serialization format suitable for resource-constrained environments. Define CBOR schemas for representing $SK$, $PK$, and $r$.

* **ASN.1 Encoding (For Legacy Systems):** If interoperability with systems using ASN.1 is required, define the appropriate ASN.1 modules and encoding rules for EchoPulse data structures.

* **Compact Binary Formats (Resource-Constrained):** For very limited environments, a simple, fixed-length binary format or a TLV (Type-Length-Value) structure for messages containing EchoPulse keys and ciphertexts can be defined. Ensure this format is well-documented.

## 8. Conclusion

Implementing EchoPulse requires careful consideration of the target platform's constraints and potential security vulnerabilities. Prioritizing constant-time execution, efficient memory management, and adherence to standardized data formats will be crucial for building secure and interoperable EchoPulse implementations. Developers should consult the formal specifications of the EchoPulse protocol and conduct thorough testing and security analysis of their implementations.

Version 1.0 — Implementation Layer — EchoPulse Initiative