
Low-RAM Operating Mode for EchoPulse: KEM Optimization for Constrained Environments (<6KB RAM)

1. Introduction

The EchoPulse Key Encapsulation Mechanism (KEM), while designed for its novel symbolic-graph-based security, faces significant memory constraints on ultra-low-power embedded systems such as ARM Cortex-M0+ microcontrollers, JavaCards, and Secure Elements. Standard implementations might require 8-10 KB of RAM, which exceeds the typical available RAM on these platforms (often 4-8 KB). This document defines a specialized "Low-RAM" operating mode for EchoPulse, detailing architectural optimizations, graph compression techniques, and crucial constant-time security considerations to enable deployment on systems with less than 6 KB of available RAM.

2. RAM-Optimized EchoPulse Variant Architecture (Embedded Memory Systems Engineer)

Achieving a RAM footprint of ≤6 KB (and ideally ≤4 KB for M0+) requires a radical re-evaluation of memory allocation. The primary memory consumer in EchoPulse is the transition function $\delta: V \times \Sigma \rightarrow V$, which, for $V=1024$ states and $\Sigma=256$ symbols, directly implies 1024×256 mappings. If each mapping is a 2-byte state ID, this is 512 KB, clearly unacceptable for RAM. This leads to ROM/Flash storage for δ and on-demand loading.

2.1. Memory Allocation Strategy (Target ≤6 KB):

- **Global/Static Data:**
  - **Public Key (PK) & Base Graph** G0 **Parameters:** Stored primarily in Flash/ROM. Only small pointers/indices are in RAM.
  - **Secret Key (SK):** For Decapsulation, SK (which defines the inverse path logic or lookup) can also be largely Flash-based, potentially using a minimal RAM buffer for decryption primitives.
- **Dynamic / Working RAM Budget:**
  - **r Buffer:** The encapsulated random payload r (e.g., 256 bits / 32 bytes). This must be in RAM for processing.
  - **Hash Input Buffer:** v_enc || r (e.g., up to 36 bytes for venc and r). Used for

the final hash computation.

- **Current State Register:** A single 2-byte (10-bit) register for the current vj.
- **Transition Lookup Buffer/Cache:** This is the critical component. Instead of storing the full δ table, we implement a "sector-based transition loading" or "partial table construction" approach.
  - **Sector-Based Transition Loading (Paging):**
    - Divide the complete δt table (residing in Flash/ROM) into fixed-size "sectors" or "pages."
    - A small RAM buffer (e.g., 256 bytes to 1 KB) acts as a cache for the currently active sector(s).
    - When δ(v, s) is needed:
      1. Calculate the sector_ID and offset_within_sector for (v,s) in the Flash/ROM table.
      2. Check if the required sector is already in the RAM buffer.
      3. If not, load the entire sector from Flash/ROM into the RAM buffer. This might involve a small overhead but minimizes random Flash accesses.
  - **Minimal Stack:** Restrict function call depth and local variable usage to keep stack footprint low (e.g., < 200 bytes).
  - **Fixed-size Buffers:** All intermediate buffers (e.g., for hash computation, small data transfers) must be pre-allocated and fixed-size, avoiding dynamic memory allocation (malloc).

2.2. Memory Allocation Per Component (Example for ≈4 KB Target):

Component	Allocation Type	Size (Bytes)	Notes
r payload buffer	RAM	32	256-bit random payload.
Hash Input Buffer	RAM	36	`v_enc
Current State v_j	RAM (register)	2	10-bit state, typically fits in CPU register or small var.

H_adapter context	RAM (static)	≈ 128-256	Internal state for BLAKE2s, SHAKE128, etc.
Transition Cache (RAM)	RAM	≈ 1024 - 2048	This is the crucial variable. Caches 1-2 sectors of δ.
Stack	RAM	≈ 256 - 512	Minimal stack usage for function calls.
SK/PK (minimal)	RAM	≈ 32 - 64	For small context, pointers, not full key. Full SK/PK in Flash.
Total Estimated RAM		**≈ 1.5 - 3.0 KB**	Achievable with efficient cache sizing.
δ **Table**	**Flash/ROM**	≈ 512 KB (compressed)	Primary storage for graph transitions.
Code (firmware)	Flash/ROM	≈ 10-20 KB	EchoPulse KEM logic, hash functions, μ.

3. Graph Compression Design (Graph Compression Designer)

The 512 KB full δ table must be heavily compressed to fit within typical embedded Flash/ROM constraints (e.g., 32-64 KB).

- **Prefix Symbol Caching / Huffman Encoding:**
  - **Concept:** Analyze the distribution of δ(v,s) outputs. If certain next states are more probable for specific symbols or from specific states, Huffman encoding or similar variable-length encoding can reduce the storage size of the *output* states.
  - **Implementation:** Store a compressed table of next_state values. A small

lookup table (LUT) or decompressor circuit/function translates compressed data to full state IDs.

- **Bit-Packed Transition Matrix:**
  - **Concept:** A 1024×256 matrix where each entry is a 10-bit state ID. Directly store these 10-bit values tightly packed.
  - **Implementation:** Store as (1024 * 256 * 10) / 8 bytes. This is 320 KB. Still too large. Requires further compression.
- **Sparse Representation / Partial Table Construction On-Demand:**
  - **Concept:** If δ(v,s) is *not* a dense, arbitrary mapping but derived from a mathematical function or a smaller set of rules (even if complex and mutation-aware), then the full table doesn't need to be stored. Instead, a "rule engine" computes δ(v,s) on the fly.
  - **Implementation:**
    - **Rule-based Generation:** The δ values are not explicitly stored. Instead, δ(v,s) is computed by a small algorithm (e.g., (v + s) % |V| + f(v,s,t) where f depends on μ). This implies μ also modifies these underlying rules.
    - **Differential Encoding:** Store δbase and then store diff_tables for μ(G0,t) at specific t. Only store changes from the base or previous t.
    - **Block-wise Compression:** Instead of a full matrix, divide the δ table into smaller logical blocks (e.g., 16×16 entries). Compress each block using dedicated algorithms (e.g., LZW, RLE, or custom bit-packing) and store indices to these blocks. Load relevant blocks into RAM cache.
  - **Compatibility with δ:V×Σ⟶V:** Regardless of the compression, the δ function must be callable as get_next_state(current_v, symbol, time_t) which internally handles decompression/rule application.

4. Constant-Time Security Evaluation (Constant-Time Security Evaluator)

Memory access patterns and control flow variations based on secret data or timing-sensitive operations can leak side-channel information. This is particularly critical in constrained environments where countermeasures are difficult to implement.

- **Problem:** On-demand transition loading (paging) or symbolic path resolution based on input r could result in variable execution times due to cache misses or different computation paths for sparse representations. This could leak information about r or the traversed path.
- **Proposed Countermeasures:**
  - **Lazy Resolution with Padded Delay:**
    - If a cache miss occurs during δ(v,s) lookup, the system initiates the

Flash/ROM read for the required sector.
        - To hide this variable latency, a fixed maximum delay is enforced. If the data is loaded quickly, the CPU waits for the remaining time. This ensures all δ lookups appear to take the same amount of time.
        - **Trade-off:** Adds fixed overhead, impacting overall encapsulation time.
    - **Masked Access for Uniform Execution:**
        - If using a rule-based δ computation where different inputs might cause different execution paths (e.g., if (v > X)), transform the logic into bitwise operations or lookups that execute in constant time, regardless of the values.
        - For table lookups, always read the entire "cache line" or block even if only one value is needed. This avoids data-dependent memory accesses.
        - For Flash/ROM access, always read the same number of bytes for a "sector load," even if the requested data is smaller, then mask off the unneeded bits.
        - **Trade-off:** Can increase RAM usage for larger read buffers or slightly higher Flash read bandwidth.
- **Recommendation:** All memory access and computational steps within the δ transition and μ application logic, as well as the final hash computation, must be audited for constant-time behavior. This is paramount for Secure Element and Smartcard deployments.

5. Mode Integration Plan (Mode Integration Planner)

EchoPulse will support distinct operational profiles, allowing integrators to choose the best fit for their target hardware and security requirements.

5.1. Operational Profiles:

1. **Full-RAM EchoPulse (Standard Mode):**
    - **Description:** The entire (or a heavily pre-computed and cached) δ table resides in RAM. This offers the fastest performance for δ lookups.
    - **RAM Footprint:** ≈ 8-10 KB (assuming pre-computed Gt for a range of t).
    - **Timing Variance:** Minimal, as all lookups are fast RAM accesses.
    - **Platform Fit:** Cortex-M4F/M33, higher-end RISC-V with ≥ 16KB RAM.
    - **Best for:** High-throughput applications, systems where RAM is abundant.
2. **Low-RAM EchoPulse (Paged Transitions Mode):**
    - **Description:** δ table resides in Flash/ROM, with a small RAM cache for active sectors. μ applies updates to the Flash table (logically) or derives pointers to

different Flash segments.
    - **RAM Footprint:** ≈ 1.5 - 3.0 KB (as detailed in Section 2.2).
    - **Timing Variance:** Moderate. Introduces variance due to Flash/ROM page loading, *unless* padded delays are implemented.
    - **Platform Fit:** Cortex-M0+, JavaCards, Secure Elements with ≥ 4KB RAM.
    - **Best for:** Ultra-constrained IoT devices, smartcards where RAM is scarce, and performance is less critical than footprint. Requires careful CT implementation.
3. **Compile-time Static Layout Mode (No RAM δ Cache, All Lookup):**
    - **Description:** The δ table is entirely in Flash/ROM. Each δ(v,s) lookup directly accesses Flash/ROM. No RAM buffer for transitions.
    - **RAM Footprint:** ≈ 0.5 - 1.0 KB (only for r, hash context, stack, vj).
    - **Timing Variance:** Potentially High. Each δ lookup is a direct Flash/ROM access, which can vary significantly due to bus contention, Flash read times, and physical layout, making CT challenging.
    - **Platform Fit:** Extreme edge cases with < 4KB RAM. Highly specialized Secure Elements.
    - **Best for:** Absolute minimal RAM. **Warning:** Requires *extremely* rigorous side-channel analysis and low-level hardware-specific CT countermeasures (e.g., constant-time Flash controllers).

5.2. Operational Profile Comparison Table:

Feature	Full-RAM EchoPulse	Low-RAM EchoPulse	Compile-time Static Layout
Target RAM	≥8 KB	≈1.5–3.0 KB	≤1.0 KB
δ Storage	RAM (full/large cache)	Flash/ROM (paged)	Flash/ROM (direct access)
Encaps. Time	Fast (low µs)	Moderate (higher µs)	Slowest (highest µs)
Timing Variance	Low	Moderate (reducible w/ CT)	High (challenging CT)

Flash/Code Size	Moderate	Moderate (plus compression/paging logic)	Moderate (plus rule engine)
Side-Channel Risk	Moderate (SW dependent)	Higher (HW+SW interaction)	Highest (HW+SW interaction)
Platform Fit	M4F+, desktop	M0+, JavaCard, SE	Extreme SE

6. Recommendations for Integrators

- **Prioritize RAM:** Start with the "Low-RAM EchoPulse" mode. Its balance of efficiency, footprint, and manageable security trade-offs makes it suitable for most constrained embedded systems.
- **Hardware Acceleration:** If AES hardware acceleration is available, strongly consider the "Prehash + AES" hash backend within the chosen H_adapter configuration for optimal performance.
- **Constant-Time Verification:** For deployments on smartcards or Secure Elements, *rigorously verify* the constant-time properties of the chosen hash function and, crucially, the δ transition logic and any paging mechanisms. This includes low-level assembly review and timing experiments on the actual target hardware.
- **Flash/ROM Optimization:** Collaborate with the Graph Compression Designer to implement efficient schemes for δ table storage. The optimal scheme will depend on the characteristics of the δ values generated by μ.
- **Profiling:** Always profile the final firmware on the target hardware to identify unexpected memory usage or timing bottlenecks.

This Low-RAM operating mode enables EchoPulse to extend its reach to a wider array of embedded and ultra-constrained cryptographic devices, making it a viable Post-Quantum Cryptography (PQC) solution for edge computing.