
Symbolic Mutation Compression and RAM-Optimized Buffer Layout in EchoPulse

1. Introduction

The EchoPulse Key Encapsulation Mechanism (KEM) relies on a deterministic graph mutation function $\mu(G,t)$ to evolve the underlying symbolic graph G_t. While the previous "Low-RAM Operating Mode" document addressed overall graph storage in Flash/ROM, this document focuses specifically on optimizing the RAM footprint of the *mutation result* itself and its associated processing buffer. The goal is to define a low-footprint mutation representation that can be efficiently generated, applied, and fully reconstructed deterministically within extremely constrained RAM environments (e.g., ≤512 bytes for mutation-specific data), preserving the integrity and unpredictability properties critical to EchoPulse's security.

2. Symbolic Compression Schemes for Mutation Results (Symbolic Compression Engineer)

The mutation function $\mu(G,t)$ transforms the base graph G_{base} into G_t by altering a subset of its $\delta(v,s) \to v'$ mappings. Instead of storing a full δ_t table in RAM (which is prohibitive), we store a compressed representation of the *changes* or *rules* applied by μ.

- **Core Idea:** μ produces a "patch" or a "set of instructions" that, when applied to G_{base}, yields G_t. This patch is what needs to be compressed.
- **Proposed Bit-Level Encoding Schemes:**
  1. **Symbol Remapping via Delta Tables / XOR Diffs:**
     - **Concept:** For a given state v, the mutation might permute the next states for all symbols $s \in \Sigma$. Instead of storing explicit new target states, store a permutation vector or a bitmask/XOR difference applied to the base state.
     - **Example:** If $\delta_{base}(v,s) = s \oplus C_v$ for some constant C_v, μ could modify C_v or apply a permutation π_t to s. The mutation result could be π_t itself or parameters defining it.
     - **Format:** For each affected state v, store a bitmask or a small index into a pre-defined set of permutations/modifications.
     - **Size Implications:** If μ affects k states, and for each state it applies a chosen permutation from a library of P permutations, this could be $k \times \log_2$

P bits. For small k and P, this is very compact.

2. **Run-Length Encoding (RLE) for Repeating Patterns:**
   - **Concept:** Mutations might not be entirely random; they could follow certain patterns. E.g., "for states vx to vy, symbols sa to sb, apply transformation Z." RLE can compress sequences of identical mutation operations.
   - **Format:** (START_V, END_V, START_S, END_S, MUTATION_TYPE, MUTATION_PARAM_1, ...)
   - **Size Implications:** Highly dependent on the actual patterns produced by μ. Best for structured or "block-wise" mutations. Can lead to variable-size representations, which might require padding to fixed sizes.
3. **Prefix Maps for Grouped Symbol Transitions:**
   - **Concept:** Instead of $\delta(v,s) \rightarrow v'$, think of groups of symbols. If μ affects transitions for a specific range of s from a specific range of v, this can be compressed.
   - **Example:** A "prefix map" could define, for v_prefix || s_prefix, a specific mutation rule.
   - **Format:** A list of (v_prefix_mask, s_prefix_mask, mutation_rule_ID, mutation_params) tuples.
   - **Size Implications:** Reduces overhead if mutations primarily target "blocks" of the state-symbol space.
4. **Static-Size Representations (64-512 bytes):**
   - To simplify buffer management and prevent side-channel leakage through size, all mutation packages should be fixed-size.
   - This might involve padding smaller compressed results to the max allocated size.
   - Example fixed sizes: 64 bytes, 128 bytes, 256 bytes, 512 bytes. The choice depends on the complexity of μ's typical output.

3. RAM Buffer Layout for Mutation Execution (RAM Buffer Architect)

The mutation buffer in RAM must be designed for efficiency, constant-time operations, and minimal overhead.

- **Fixed-Size Mutation Buffer (≤512 bytes):**
  - This buffer will hold the compressed mutation result generated by μ(Gbase,t).
  - **Layout:**

```
+----------------------------------------------+
| Header (4-8 bytes)                           |
```

```
| - Mutation Type ID (e.g., 1 byte for method)    |
| - Number of Rules/Entries (e.g., 1 byte)        |
| - CRC/Checksum (e.g., 2 bytes)                  |
| - Time Index 't' (e.g., 2 bytes)                |
+-------------------------------------------------+
| Subfield 1: Mutation Rule Data                  |
|   (e.g., list of (v_idx, s_idx, new_target_v_idx) |
|   or (rule_ID, params))                         |
|   (Bit-packed and aligned to 4-byte boundaries) |
|   ...                                           |
+-------------------------------------------------+
| Padding to Fixed Size (e.g., 512 bytes)         |
+-------------------------------------------------+
```

- **Subfields:**
  - **Mutation Index/ID:** A simple integer identifying the specific mutation operation or rule set to be applied. This can be derived from the time index t.
  - **Symbol Patch Table / Transition Rewriter:** This is the core data. It could be:
    - A list of (state_idx, symbol_idx, new_target_state_idx) tuples, compactly encoded (e.g., 10 bits for state ID, 8 bits for symbol ID, 10 bits for target ID, totaling 28 bits per patch entry).
    - Parameters for a specific rule generator.
    - A small lookup table of permutation indices.
  - **Deterministic Hash (e.g., BLAKE2s-32):** A small hash of the mutation package itself, stored in the header. Used by the Reconstruction Verifier to ensure integrity.
- **Minimize Pointer Usage and Stack Depth:**
  - Avoid dynamic memory allocation. All data structures are static arrays or pre-allocated fixed buffers.
  - Prefer array indexing over pointers where possible to reduce aliasing and potential side-channel risks.
  - Strictly limit recursion and function call depth to keep the stack footprint minimal. Iterative algorithms are preferred for applying mutations.

4. Reconstruction Verification (Reconstruction Verifier)

The critical aspect of this approach is ensuring that the compressed mutation data, stored in the low-RAM buffer, can be fully and deterministically reconstructed into the effective Gt for δ lookups.

- **Proof of Full and Deterministic Reconstructibility:**
    - The mutation_result (compressed buffer) is a deterministic function of (Gbase ,t).
    - The reconstruction algorithm Reconstruct(G_base, mutation_result) must produce Gt such that for all $v \in V, s \in \Sigma$, $\delta Gt(v,s)=\delta actual\_Gt(v,s)$.
    - **Formal Argument:** The compression scheme (e.g., delta table, RLE) must be proven lossless. This means that for every unique Gt generated by μ, there is a unique and fully recoverable compressed representation, and vice-versa.
    - **Mechanism:** The Reconstruct function would typically unpack the compressed patch data and apply it to an in-memory (or Flash-resident) representation of Gbase or a logically empty graph.
- **Proposed Verification Mechanisms:**
    1. **Hash Checks (BLAKE2s):**
        - During development and testing, after a mutation_result is generated and compressed, its BLAKE2s hash (e.g., BLAKE2s-32 for 32 bits) is computed and stored.
        - After reconstruction: Generate the *full* (uncompressed) δt table. Compute its BLAKE2s hash. Compare it against a pre-computed hash of the expected Gt.
        - **Runtime Check:** Store a small hash (e.g., 32-bit CRC or BLAKE2s) of the *compressed* mutation data itself within the mutation buffer header. This allows a quick integrity check on load to ensure the data wasn't corrupted.
    2. **Mutation-Seed Replay:**
        - The mutation function μ(Gbase,t) itself is deterministic. The most direct way to verify reconstruction is to:
            1. Compute G_t_true = μ(G_base, t).
            2. Compress this G_t_true to mutation_result_compressed.
            3. Reconstruct G_t_reconstructed = Reconstruct(G_base, mutation_result_compressed).
            4. Assert G_t_true == G_t_reconstructed (i.e., all δ(v,s) mappings are identical).
        - This is typically done offline during testing, not at runtime on the embedded device.
    3. **Incremental Deterministic Generators for Patch Expansion:**
        - For some compression schemes (e.g., rule-based), reconstruction might involve generating the full δt table on the fly or sector-by-sector.

- Verification: Ensure that the underlying mutation_rule_ID corresponds to a unique and deterministic rule_expansion_function.
- **Example:** A "rule ID" maps to a specific small function like delta_new(v, s) = (v XOR s) + some_constant_from_params. The verification involves ensuring that all such rule_expansion_functions are correct and cover the full range of desired mutations.

5. Export Format Management (Meta Role)

Standardizing the mutation package format is crucial for inter-operability, debugging, and formal methods integration.

- **Serialization Format:**
  - **Custom Binary Format:** Recommended for embedded runtime mutation packages due to extreme size constraints and parsing efficiency.
    - **Pros:** Minimal overhead, bit-packed, fast parsing.
    - **Cons:** Not human-readable, requires specific parser implementation.
    - **Spec Example:**
```
[0-3]: uint32_t magic_header (e.g., 0xEPMMUT)
[4-5]: uint16_t version (e.g., 0x0100)
[6-7]: uint16_t mutation_type_id
[8-9]: uint16_t time_index_t
[10-11]: uint16_t num_patch_entries
[12-15]: uint32_t crc32_checksum // or BLAKE2s-32 hash
[16-N]: byte[] compressed_patch_data // Bit-packed as per
Symbol_Compression_Engineer's design
[N+1-M]: byte[] padding_to_fixed_size // e.g., 512 bytes
```
  - **JSON/CBOR:** Preferred for debug/test mutation logs and formal proof simulation inputs.
    - **JSON:** Human-readable, easy for scripting, but verbose. Good for initial prototyping.
    - **CBOR (Concise Binary Object Representation):** Binary-encoded JSON, more compact than JSON but still self-describing. Good for test vectors exchange.
    - **Spec Example (JSON):**
```json
{
  "magic_header": "EPMMUT",
```

```
  "version": "1.0",
  "mutation_type": "delta_remap",
  "time_index": 1234,
  "checksum_blake2s": "abcdef1234567890", // Hash of 'patch_data'
  "patch_data": [
    {"v_start": 0, "s_val": 1, "v_target": 12},
    {"v_start": 5, "s_val": 255, "v_target": 1023},
    // ... compressed representation of changes
  ]
}
```

- **Validation Rules for Formats:**
  - **Schema Definition:** For JSON/CBOR, provide a JSON Schema to ensure correct structure.
  - **Checksum Verification:** Mandate the use of a checksum (e.g., CRC32 or BLAKE2s) in the header of all mutation packages to detect transmission or storage errors.
  - **Range Checks:** Ensure all indices and values within the compressed patch data fall within valid ranges (e.g., $v \in [0, |V|-1]$, $s \in [0, |\Sigma|-1]$).

6. Final Recommendations for Integrators

- **Prioritize Bit-Packing:** For the actual compressed_patch_data, bit-packing and custom encoding will yield the most significant RAM savings.
- **Fixed Size is Key:** Design all mutation buffers and packages to be fixed sizes to eliminate timing side-channels related to data length. Pad smaller packages if necessary.
- **Offline Compression & Verification:** The most complex compression and reconstruction verification (e.g., G_t_true == G_t_reconstructed) should happen offline during development and testing. The embedded device only performs the simpler runtime reconstruction.
- **Test with Worst-Case Mutations:** Benchmark performance and memory usage with mutation patterns that are least compressible to ensure the chosen fixed buffer size is sufficient for all t.
- **Integrate with Low-RAM Operating Mode:** This mutation compression scheme is a vital component of the overall "Low-RAM Operating Mode" for EchoPulse. Both designs must be implemented in conjunction.

This design document provides a robust framework for managing and optimizing the

RAM footprint of EchoPulse's dynamic mutation, enabling its deployment on the most memory-constrained platforms while upholding its cryptographic integrity.