

# EchoPulse MPKX – Replay Validation & Repair Logic (v3.0)

## 1. Overview

This document formalizes the robust replay detection and resolution mechanisms within the EchoPulse Multi-Party Key Exchange (MPKX) protocol. Building upon the foundational principles of time-bound symbolic graph mutation and Post-Quantum Forward Secrecy (PQ-FS) as established in prior EchoPulse specifications (Files 43-45), this document details how MPKX participants ( $P_1 \dots P_n$ ) can reliably identify and gracefully recover from replay attempts, inconsistencies, or communication stalls. The core of this logic relies on derived `ReplayToken` values, rigorous epoch validation, and an explicit `MPKX_Repair()` mechanism, ensuring the integrity and synchronized state of the distributed symbolic graph across all participants.

## 2. Replay Defense Architect: Replay Detection Logic

Replay detection in EchoPulse MPKX is a critical function, ensuring that each MPKX contribution is fresh and pertains to the current, expected session. This is achieved through a multi-layered approach centered around the `forward_entropy_epoch (t)` and derived `ReplayToken` values.

### 2.1. `ReplayToken(P_i)` Derivation

Each participant  $P_i$  generating an `MPKX_Contribution` (as defined in "EchoPulse Multi-Party Key Exchange – Architectural Foundation (v3.0)", File 46) *MUST* include a `ReplayToken(P_i)`. This token serves as a session-specific cryptographic fingerprint of  $P_i$ 's contribution within the current epoch.

The `ReplayToken(P_i)` is derived using a cryptographic hash function ( $H$ ) and depends on several ephemeral and session-bound elements:

$$\text{ReplayToken}(P_i) = H(\text{MPKX\_Session\_ID} \parallel \text{P\_i.ID} \parallel \text{forward\_entropy\_epoch}(t) \parallel H(\text{P\_i.ID} \parallel \text{H\_Anchor}(P_i)))$$

Where:

- **MPKX\_Session\_ID**: A unique identifier for the entire MPKX session, derived from the initial common random values exchanged to establish the session.
- **P\_i.ID**: The unique identifier of participant  $P_i$ .
- **t**: The `forward_entropy_epoch` for the current session, deterministically derived and agreed upon by all participants (consistent with File 45). This binds the token to the specific Graph-Time State  $G_t$ .

- $H(P_i.r_i)$ : A hash of  $P_i$ 's ephemeral secret symbolic path  $r_i$ . This ensures the token is uniquely tied to  $P_i$ 's secret contribution, without revealing  $r_i$ .
- $\text{H\_Anchor}(P_i)$ : An "anchor hash" that cryptographically binds  $P_i$ 's contribution to the derived graph state  $G_t$  and the HORM operations performed.
  - $\text{H\_Anchor}(P_i) = H(\text{transcript\_hash\_current} : || : \text{P}_i.v_i.\text{end} : || : \text{aggregated\_HORM\_outputs}_i)$ 
    - `transcript_hash_current`: A running hash of all MPKX messages exchanged *prior* to  $P_i$ 's contribution (similar to Transcript-Hash in TLS).
    - `P_i.v_i.end`: The final vertex reached by  $P_i$ 's traversal using  $r_i$  on  $G_t$ .
    - `aggregated_HORM_outputs_i`: A cryptographic aggregation of the  $h_{\text{local}}$  outputs from  $P_i$ 's HORM traversal.

## 2.2. Validation Logic: $t_{\text{recv}}$ vs. $t_{\text{expected}}$

Each participant  $P_k$  receiving an MPKX\_Contribution from  $P_i$  performs the following validation steps:

### 1. Epoch Consistency Check:

- $P_k$  retrieves the  $t$  value implied by the MPKX\_Session\_ID within the received message (which should match the  $t$  currently computed by  $P_k$  for the session).
- Let this be  $t_{\text{recv}}$ .
- $P_k$  compares  $t_{\text{recv}}$  against its own currently established `forward_entropy_epoch` ( $t_{\text{expected}}$ ).
- If  $t_{\text{recv}} \neq t_{\text{expected}}$ , a **hard mismatch** is detected, indicating a replay attempt from a different epoch or a fundamental synchronization failure.

### 2. ReplayToken( $P_i$ ) Verification:

- $P_k$  re-computes  $\text{ReplayToken}'(P_i)$  using the data from the received message and its own  $G_t$  (which corresponds to  $t_{\text{expected}}$ ).
- If  $\text{ReplayToken}'(P_i) \neq \text{ReplayToken}(P_i)$  (from the received message), a **replay attempt** or message tampering is detected. This also implicitly covers H\_Ancor mismatch.

### 3. HORM Consistency Check (Implicit in ReplayToken):

- The  $\text{H\_Anchor}(P_i)$  component within  $\text{ReplayToken}(P_i)$  indirectly ensures that  $P_i$ 's traversal on *its*  $G_t$  yielded the expected results. If an attacker replays a  $C_i$  but  $P_i$ 's original traversal path cannot be re-produced on the current  $G_t$ , the H\_Ancor will differ, causing the  $\text{ReplayToken}$

validation to fail.

## 2.3. Flow Diagram of Detection Logic

Code-Snippet

graph TD

```
A[MPKX_Contribution Received from Pi] --> B[Verify Signature of Pi];
B -- FAIL --> F[ABORT: Invalid Signature];
B -- SUCCESS --> C[Extract MPKX_Session_ID, Pi.ID, ReplayToken(Pi), ...];
C --> D[Calculate t_expected for current session];
D --> E[Extract t_rcv from MPKX_Session_ID];
E --> F1{t_rcv == t_expected?};
F1 -- NO (Hard Mismatch) --> G[Flag: Hard Replay / Epoch Mismatch];
F1 -- YES --> H[Recompute ReplayToken'(Pi) using local Gt and received data];
H --> I{ReplayToken'(Pi) == ReplayToken(Pi)?};
I -- NO (Soft Replay / Tamper) --> J[Flag: Soft Replay / Tamper];
I -- YES --> K[Proceed with MPKX_Aggregate];
G --> R[Initiate MPKX_Repair() or Abort];
J --> R;
```

## 3. Graph Repair Strategist: MPKX\_Repair() Protocol

When a replay or inconsistency is detected, the MPKX\_Repair() mechanism is invoked to attempt to re-synchronize the group or gracefully abort. The goal is to restore a consistent  $\$G_t\$$  and correct participant contributions.

### 3.1. Formal Replay Classes and Recovery Pathways

- **Hard Mismatch (Epoch Mismatch):**  $t_{rcv} \neq t_{expected}$ .
  - **Cause:** Participant out of sync on forward\_entropy\_epoch, replay from a distant session, or malicious epoch manipulation.
  - **Recovery Pathway:** Highly critical. Immediate attempt to resynchronize  $\$t\$$ .
    1. **Request t Resync:** The detecting participant broadcasts a MPKX\_Repair\_Request(type=EPOCH\_RESYNC, current\_t\_expected).
    2. **Vote/Propose New t:** Participants respond with their  $t_{expected}$  and a hash of their current session transcript. The group attempts to converge on a new, verifiable  $t$  (e.g., using a fresh random nonce and re-hashing

the full MPKX setup transcript).

3. **Abort if No Consensus:** If consensus on  $t$  cannot be reached within a timeout, the MPKX session *MUST* abort.
- **Soft Replay / Tampering (ReplayToken Mismatch):**  $\text{ReplayToken}'(P_i) \neq \text{ReplayToken}(P_i)$ , but  $t_{\text{recv}} == t_{\text{expected}}$ .
  - **Cause:** Replay of an individual message within the same nominal epoch, or subtle tampering with  $P_i$ 's contribution (e.g., altering  $v_i$  or  $\text{public\_nonce}_i$ ).
  - **Recovery Pathway:** Targeted repair.
    1. **Request  $C_i$  Resend:** The detecting participant broadcasts  $\text{MPKX\_Repair\_Request}(\text{type}=\text{RESEND\_CONTRIBUTION}, \text{target\_P\_i\_ID})$ .
    2. **Targeted Resend:**  $P_i$  *MUST* regenerate its  $r_i$  (to ensure freshness, if not already used), recompute its  $C_i$  and  $\text{ReplayToken}(P_i)$ , and resend the  $\text{MPKX\_Contribution}$ .
    3. **Timeout/Abort:** If  $P_i$  fails to resend a valid contribution within a timeout,  $P_i$  is considered a stalling peer, and the group *MAY* proceed without  $P_i$  (if tolerant to reduced group size) or abort.
- **Stalling Peer:** A participant fails to send its  $\text{MPKX\_Contribution}$  within a timeout or consistently fails validation.
  - **Recovery Pathway:** Group decision.
    1. **Exclusion:** The group *MAY* decide to exclude the stalling peer and proceed with a reduced set of  $N' < N$  participants. This requires re-calculation of  $K_{\text{Group}}$  using only the valid contributions.
    2. **Abort:** If the reduced group size is unacceptable or if the stalling peer is critical, the MPKX session *MUST* abort.

### 3.2. Fallback to Previously Agreed $\mu_{\text{ref}}(G_t)$ (Conditional)

In rare cases of extreme graph inconsistency, if a group has a pre-established, cryptographically signed  $\mu_{\text{ref}}(G_t)$  (a reference mutation state for a given  $t$ ) available out-of-band, they *MAY* attempt to revert to it. However, this is generally discouraged for true PQ-FS as it might indicate a persistent compromise or a flaw in the  $\mu(G,t)$  derivation itself. This feature is more relevant for DEBUG or INITIALIZATION phases rather than operational repair.

### 3.3. Repair Contract

Participants must adhere to a Repair Contract:

- A participant receiving an  $\text{MPKX\_Repair\_Request}$  *MUST* attempt to fulfill it by re-

generating/resending information, provided it aligns with their internal state and current  $t$ .

- Participants *MUST* accept or reject MPKX\_Repair messages based on strict HORM validation and epoch consistency. Acceptance means updating internal state (e.g.,  $\$t\$$ , pending contributions) and retrying aggregation. Rejection means continuing to flag the inconsistency or aborting.

#### 4. Protocol State Auditor: Internal State Registers and Buffers

Each EchoPulse MPKX participant maintains specific internal state registers to facilitate replay detection and recovery.

- **Replay\_History (Ring Buffer / Cache):**
  - A memory-bounded register storing hashes or a short digest of recently observed and *valid* ReplayToken values, possibly keyed by MPKX\_Session\_ID and P\_i.ID.
  - Purpose: Detect replayed MPKX\_Contribution messages.
  - Implementation: A fixed-size hash table or Bloom filter to keep memory footprint low. Older entries are "tombstoned" (marked invalid) or evicted as new entries come in.
- **Session\_Anchor (Fixed Register):**
  - Stores the established forward\_entropy\_epoch ( $\$t\$$ ) for the current active MPKX session.
  - Also stores the MPKX\_Session\_ID and the initial Transcript-Hash from which  $\$t\$$  was derived.
  - Purpose: Central reference for epoch validation and consistency checks.
- **Validated\_t (Fixed Register):**
  - Stores the *last successfully validated* forward\_entropy\_epoch. Useful for determining acceptable "drift" or for debugging.
- **Fallback\_mu (Optional, Read-Only):**
  - If a reference  $\mu_{ref}(G_t)$  is pre-provisioned, this register stores its cryptographic hash or identifier. Used only for extreme recovery.

##### 4.1. Symbolic Replay Logging and Flagging

Detected replays and inconsistencies are logged (e.g., as `soft_replay_flag`, `hard_mismatch_flag`) and trigger alerts. These flags inform the MPKX\_Repair() mechanism. ReplayToken tombstoning involves marking a successfully processed ReplayToken as USED in Replay\_History to prevent its re-acceptance.

## 4.2. Memory-Bounded Implementation

All state registers are designed with fixed, pre-allocated memory footprints.

- **Replay\_History:** Size dictated by `REPLAY_HISTORY_WINDOW` (e.g., last 100 or 1000 tokens).
- **Session\_Anchor, Validated\_t, Fallback\_mu:** Fixed size for cryptographic hashes/IDs.
- **Mutation buffers for  $G_t$  evolution** are managed as discussed in File 44, focusing on deterministic memory access and minimal intermediate state.

## 4.3. Time-Windowed Acceptance Logic

To account for minor network latency or clock skew, a participant *MAY* allow a small `Time_Window` for `t_rcv` around `t_expected` (e.g.,  $t_{\text{expected}} \pm \delta$ ). This  $\delta$  should be extremely small (e.g.,  $\pm 1$  or  $\pm 2$  epoch steps) and carefully tuned to avoid security degradation.

## 4.4. State-Reset and Tombstoning

- **State Reset on Inconsistency:** If a hard mismatch occurs or `MPKX_Repair()` fails to converge, the local MPKX state (including `Session_Anchor` and `Replay_History`) *MUST* be reset, and a new MPKX session initiated.
- **ReplayToken Tombstoning:** After a `ReplayToken(P_i)` is successfully validated and its associated contribution processed, its entry in `Replay_History` is marked as USED. Subsequent attempts to use the identical token within the `REPLAY_HISTORY_WINDOW` will be detected as a replay.

## 5. Meta Role – Integration & Compatibility Notes

This MPKX – Replay Validation & Repair Logic document is tightly integrated with the broader EchoPulse framework, ensuring consistency and seamless operation.

- **Alignment with EchoPulse Files 43-46:**
  - The core concept of `forward_entropy_epoch` ( $t$ ) and the Graph-Time State ( $G_t$ ) from File 43 is fundamental to the `ReplayToken` derivation and `t_rcv` vs. `t_expected` validation.
  - The `HORM` and  $K_{EP_t}$  derivation logic from File 44 underpins the `H_Anchor` calculation and the overall key derivation validity.
  - The `MPKX_Init` and `MPKX_Aggregate` phases, as well as the `MPKX_Contribution` message format from File 46, are directly referenced and extended.

- **Reuse of Core Tokens:**  $r_i$ ,  $v_{enc}$  (as  $v_{i\_end}$ ),  $entropy\_epoch$  (as  $t$ ),  $\mu(G,t)$ ,  $KDF\_HORM$  (as HORM's internal hash outputs), and  $transcript\_hash$  are consistently reused, maintaining a unified terminology across the EchoPulse dossier.
- **Modular MPKX\_Repair():** The  $MPKX\_Repair()$  protocol is designed as a distinct, callable module within the MPKX architecture. Its functionality is clearly separated from the primary key exchange, allowing for focused implementation and auditing. It explicitly references concepts like  $\mu$ ,  $KDF\_HORM$  (via HORM validation), and the session  $transcript\_hash$  for recovery and re-synchronization.
- **Resource Efficiency:** The emphasis on memory-bounded state registers and efficient hash computations ensures that the replay detection and repair logic adheres to EchoPulse's low resource footprint requirements for embedded systems.

This document completes the robust security architecture for EchoPulse MPKX, providing critical mechanisms for maintaining session integrity and resilience against various adversarial attacks, particularly replay attempts, in dynamic multi-party environments.