

EchoPulse – Aggregated Proof Chains & Recursive zkSNARK Logic (v3.0)

1. Motivation and ZK Need

The EchoPulse framework provides Post-Quantum Forward Secrecy (PQ-FS) and Multi-Party Key Exchange (MPKX) capabilities, securing communications for constrained devices. "EchoPulse – Zero Knowledge Bridge Layer (v3.0)" (File 49) introduced individual Zero-Knowledge Proofs (ZKPs) to publicly attest to the correctness of EchoPulse session derivations. However, for continuous operation, particularly with numerous MPKX sessions or high-frequency FS key derivations across many devices, generating and verifying individual zk-SNARKs on-chain can become prohibitively expensive in terms of gas fees and computational resources.

This document extends the ZKP Bridge with **recursive zk-SNARKs** and **aggregated proof chains**. The motivation is to:

1. **Reduce On-Chain Footprint:** Compress a large number of individual EchoPulse session proofs into a single, compact, and verifiable recursive proof.
2. **Enable Scalable Auditability:** Allow for efficient, periodic verification of long sequences of EchoPulse sessions, confirming correct graph mutation, key derivation, and replay defense over extended time windows.
3. **Enhance Trustless Audit:** Provide a cryptographically sound way for auditors or external parties to confirm the integrity of EchoPulse operations without needing to verify every single session individually.

2. Recursive zkSNARK Structure

The core innovation is a recursive zk-SNARK design where a proof for the current session verifies the correctness of a previous session's proof alongside its own.

2.1. zkSNARK_P(i) Proof Statement

Let $\text{zkSNARK_P}(i)$ denote the zk-SNARK for session i . It asserts:

- **MPKX Completion (or KEM derivation) of P_j within Session Window S_i :**
 $\text{zkSNARK_P}(i)$ proves that for a set of participants P_j in session S_i :
 - Each P_j correctly derived its contribution C_j by traversing its secret r_j on G_{ti} via the HORM.
 - The $H_Anchor(P_j)$ for each participant was correctly computed and matches the expected values.
 - The overall group key K_{Group} (or session key KE_{Pt}) was deterministically derived.

- **Replay Token Validity:** For each participant P_j 's contribution, it was verified that their $\text{ReplayToken}(P_j)$ adheres to the validity rules for t_i (i.e., $\text{ReplayToken}(P_j) \in \text{ValidEpochs}(t_i)$), confirming it's not a replay from a previous epoch.
- **Transcript Commitment Matching $\mu_i(G_{ti})$:** The proof confirms that the derived G_{ti} (represented by μ^* for efficiency) correctly reflects the entropy_epoch (t_i) and the transcript_hash of the session's setup. This ensures the graph used for key derivation was indeed the one expected for that specific epoch.
 - Crucially, this check occurs *without leaking* r_j or specific internal graph states.

2.2. Recursive Chain Compression: $zk_P(i) \rightarrow zk_P(i+1)$

A recursive zk-SNARK takes as input a proof from a previous instance of the same SNARK. This creates a chain of proofs, where $zk_P(i+1)$ verifies the correctness of $zk_P(i)$'s statement *and* the correctness of session $i+1$'s operations.

Code-Snippet

graph TD

subgraph Prover Side (Off-Chain)

Zk_P_i[zk_P(i): Proof for Session i]

S_i_data[Session i data (r_i, G_ti, ...)]

S_i_plus_1_data[Session i+1 data (r_i+1, G_ti+1, ...)]

Zk_P_i_plus_1[zk_P(i+1): Proof for Session i+1]

S_i_data -- input --> Zk_P_i

Zk_P_i --> Zk_P_i_plus_1

S_i_plus_1_data -- input --> Zk_P_i_plus_1

end

subgraph Verifier Side (On-Chain)

Zk_P_final_Commitment[zk_P(Final): Public Commitment]

Zk_Verify[On-Chain Verifier]

Zk_P_i_plus_1 -- output --> Zk_P_final_Commitment

Zk_P_final_Commitment -- input --> Zk_Verify

end

Workflow:

1. **zk_P(1):** A participant computes a zk-SNARK proving the correctness of their operations for the first session (S1). Public inputs include t_1 , μ^*_1 , $H_Anchor(P_j)_1$, etc.
2. **zk_P(i+1):** For the (i+1)-th session, the participant creates a new zk-SNARK that has as input:
 - All public inputs for session S_{i+1} .
 - The *output proof* from $zk_P(i)$.
 - The circuit for $zk_P(i+1)$ contains sub-circuits to:
 - Verify the embedded $zk_P(i)$ proof.
 - Verify the correctness of all EchoPulse operations for session S_{i+1} . This process continues, creating a recursive chain where the final proof $zk_P(N)$ compactly attests to the validity of all N preceding sessions.

2.3. Batch Proof Aggregation: $ZKP_Aggregate(S_{i:n}) = ZK_final_commit$

A recursive proof itself is a form of aggregation. However, if multiple recursive chains are generated by different groups or in parallel, a final "batch proof" can be created:

- $ZKP_Aggregate(S_start:S_end)$: A single overarching zk-SNARK that takes as input a set of final recursive proofs (e.g., $zk_P(N)_{groupA}$, $zk_P(M)_{groupB}$) and verifies all of them in a single, even smaller proof.
- The final output is a single, compact ZK_final_commit (a hash or cryptographic commitment) that can be posted on-chain.
- **Bound Proof Size < 32KB:** The goal is to ensure the final proof size is extremely small, typically achieved by zk-SNARKs like Groth16. This makes on-chain storage and verification economical.

3. Entropy Consistency Layer Architect: Cross-Epoch Consistency

A dedicated zk-layer ensures critical consistency properties across multiple entropy_epoch states (t_i).

3.1. Epoch Drift Tolerance Definition

While t is strictly monotonic, practical implementations may experience minor, legitimate deviations or network latency. The ZKP logic can incorporate:

- **Prove($t_{i+1} = t_i + \Delta t$ for expected Δt):** A sub-circuit that proves the new entropy_epoch t_{i+1} was correctly derived from t_i and the

expected incremental change, allowing for a small, defined tolerance (δ). This prevents arbitrary jumps in t .

- This would allow a Time_Windowed Acceptance Logic (File 47) to be verified within the ZKP itself.

3.2. Symbolic Anchor Rebinding

The $H_Anchor(P_i)$ (File 47) binds a participant's contribution to $transcript_hash_current$ and G_{t_i} . In recursive proofs, this extends:

- **$Prove(H_Anchor(P_j)_i \rightarrow G_{t_i})$** : The recursive SNARK proves that each individual H_Anchor was correctly tied to the specific G_{t_i} and the session's $transcript_hash$.
- **$Prove(G_{t_i} \rightarrow G_{t_{i+1}} \text{ via } \mu(G_{t_i}, \Delta t))$** : A critical sub-circuit that proves that $G_{t_{i+1}}$ was correctly derived from G_{t_i} by applying the mutation function μ for Δt steps. This ensures the integrity of the graph evolution over time.

3.3. FallbackChain(μ_ref) Resolution Path

In scenarios where $MPKX_Repair()$ (File 47) might fallback to a μ_ref (a pre-agreed reference mutation state), the ZKP can prove the validity of this fallback:

- **$Prove(G_t \text{ uses } \mu_ref \text{ with valid reason})$** : The ZKP circuit includes logic to prove that the current G_t is either a standard derivation from G_0 and t , OR it is a valid application of μ_ref for a specific, proven reason (e.g., error recovery, detected inconsistency). This adds an auditable trail for deviations from the standard mutation.

3.4. Symbolic Proofs for Consistency: $Prove(\mu_i \wedge \mu_j \wedge transcript_hash_j \mid \forall j \in [i, i+k])$

This represents a high-level statement proven by the recursive SNARK. It asserts that for a batch of k consecutive sessions (from i to $i+k$):

- The μ^* representation of each graph G_{t_j} was consistent with its respective $entropy_epoch(t_j)$.
- The $transcript_hash_j$ for each session was correctly processed.
- The overall chain of graph mutations $\mu(G_{t_j}, \Delta t)$ was correctly applied and verified for all j .

4. On-Chain Contract Compression Specialist: Smart Contract Logic

The on-chain smart contract is optimized for minimal gas cost, primarily verifying the final recursive proof.

4.1. Submission of Recursive Proof Result

- **EchoPulseRecursiveVerifier.sol (or similar):** A Solidity smart contract that implements the verifier algorithm for the chosen recursive zk-SNARK scheme (e.g., Groth16 verifier).
- **submitRecursiveProof(bytes memory _proof, bytes32 _publicInputsHash):**
 - This function is called by a trusted off-chain prover (e.g., a dedicated ZKP aggregator service or a gateway node) after a batch of EchoPulse sessions.
 - **_proof:** The compressed, final recursive zk-SNARK proof.
 - **_publicInputsHash:** A hash of all the public inputs (e.g., t_{final} , μ^*_{final} , aggregated H_Anchor hashes) that the recursive proof commits to. This saves gas by not sending all public inputs on-chain.
 - The contract calls its internal `verifyProof(_proof, _publicInputsHash)` function.
 - If valid, the contract records the `_publicInputsHash`, the block number, and the timestamp, serving as an immutable, verifiable anchor for the entire proof chain.

4.2. Minimal Validation Stub `ZK_Verify(z_final)`

The `verifyProof` function within the `EchoPulseRecursiveVerifier` contract serves as the ultimate `ZK_Verify` stub. Its output (boolean: valid/invalid) is the concise result of validating potentially thousands of underlying EchoPulse operations. The size of this verifier on-chain is fixed and small, regardless of the number of aggregated proofs.

4.3. Challenge/Response Model in Case of Contradiction

While zk-SNARKs are computationally sound, a challenge/response model can be implemented for practical scenarios (e.g., if a proof is submitted but later a discrepancy is detected off-chain).

- **challengeRecursiveProof(bytes32 _publicInputsHash):** A function on the contract allowing a designated challenger to dispute a submitted proof.
- **Dispute Resolution:** This would typically trigger an off-chain process where a trusted auditor re-computes portions of the proof chain or delves into the individual session data. If a contradiction is found, the submitted proof on-chain might be invalidated, and potential penalties could be applied.
- **Re-proof/Re-submission:** The original prover may be required to re-submit a corrected recursive proof.

4.4. Audit Logging Structure for Recursive Chain Footprint

- The EchoPulseRecursiveVerifier contract will store minimal audit logs:
 - event ProofVerified(bytes32 indexed publicInputsHash, address indexed prover, uint256 blockNumber, uint256 timestamp)
- This on-chain log provides an immutable, transparent record of all successfully verified proof chains, serving as a high-level audit trail without storing the full details of every session.
- Off-chain audit tools can then retrieve these publicInputsHash values and query the underlying ZKP system (e.g., a ZKP archival service) for the full proof and public inputs for detailed investigation.

5. Security Model

The security of recursive zk-SNARKs relies on:

- **Cryptographic Soundness of zk-SNARKs:** The underlying zk-SNARK scheme guarantees that if a proof is valid, the statement is true (soundness), and no secret information is revealed (zero-knowledge).
- **Correct Circuit Implementation:** The zk-SNARK circuits *must* correctly encode all EchoPulse logic (HORM, mutation, path traversal, replay token, H_Anchor, KDF) without flaws.
- **Monotonic entropy_epoch (t):** The secure and verifiable derivation of t (File 45) is critical for ensuring non-reusability of graph states and for linking sessions in the recursive chain.
- **Replay Defense (File 47):** The ReplayToken validation, as proven within the SNARK, ensures that even aggregated proofs do not implicitly validate replayed or inconsistent messages.
- **Side-Channel Resilience (File 48):** The underlying EchoPulse implementation should be side-channel resistant. The ZKP layer confirms the *correctness* of the computation, but does not inherently protect against physical leakage during prover execution. The circuit should reflect the SCA-hardened computational steps.

6. Meta Role – Framework Consistency Check

This document builds upon and extends the EchoPulse framework, maintaining full consistency across its various layers.

- **Compatibility with Files 43–49:**
 - **File 43 (PQFS Structure):** The entropy_epoch (t) and $\mu(G,t)$ are central to

the graph evolution proven correct in the SNARK.

- **File 44 (PQFS Key Derivation Model):** The correctness of key derivation based on G_t and r_i is implicitly proven.
- **File 45 (TLS Handshake Extension):** The derivation of t from `transcript_hash` is a public input to the ZKP, linking the handshake's integrity.
- **File 46 (MPKX Architecture):** The MPKX contributions and the overall group key derivation are precisely what the recursive SNARK is designed to verify.
- **File 47 (Replay Validation & Repair):** The `ReplayToken` validation is a direct component of the zk-SNARK circuit. `MPKX_Repair()` outputs could also trigger ZKP submissions for verification of repair attempts.
- **File 48 (Side Channel Defense):** The recursive SNARK proves the correctness of the *logical* execution. If the implementation used SCA defenses (like r_i rotation fuzzing), the circuit would verify the result of the *fuzzed* computation, not the raw r_i or G_t data.
- **Consistent Use of Symbolic μ , r_i , `transcript_hash`:** These core EchoPulse terms are used consistently throughout the ZKP design, preserving semantic clarity. μ^* is introduced as a specific ZKP-optimized representation of $\mu(G,t)$.
- **Aggregation Does Not Disrupt FS or MPKX Timing Logic:** The aggregation is an *off-chain* process for proving correctness. It does not interfere with the real-time, per-session timing and forward secrecy properties of individual EchoPulse sessions. The individual sessions are conducted normally, and their proofs are aggregated *after* the fact.
- **Clearly Documented Recursive Assumptions and Logic Paths:** This document clearly outlines the recursive nature and the specific sub-proofs necessary for the aggregate proof. All assumptions (e.g., trusted setup for SNARKs, if applicable) are stated.

This document marks a significant leap in EchoPulse's auditability and scalability, providing a cryptographically verifiable and highly efficient mechanism for confirming the integrity of numerous EchoPulse sessions, even in complex multi-party and dynamic environments.