

# EchoPulse – Zero Knowledge Bridge Layer (v3.0)

## 1. Overview

This document introduces the **EchoPulse Zero Knowledge Bridge Layer**, an extension designed to integrate the EchoPulse security framework with blockchain technology through the use of Zero-Knowledge Proofs (ZKPs). This layer enables participants to cryptographically prove the legitimacy of their EchoPulse sessions and contributions (e.g., in MPKX) without revealing any underlying session secrets or graph states. This enhances auditability, provides verifiable transaction integrity, and offers a robust mechanism for dispute resolution in distributed environments. By leveraging ZKPs, EchoPulse can publicly attest to the correctness of its complex, dynamic cryptographic operations while preserving the strict confidentiality of the session key material and graph evolution.

## 2. ZK Cryptosystem Designer: ZK Proof Design

The core idea is to construct a ZKP that verifies the correct execution of EchoPulse's key derivation logic, specifically the traversal of a symbolic path ( $r_i$ ) on a given Graph-Time State ( $G_t$ ) and its interaction with the Hash Oracle Replacement Model (HORM).

### 2.1. ZK Proof: Prove( $\mu(G_t)$ , entropy\_epoch, $r_i$ , transcript\_hash)

The Prover ( $P_i$ ) aims to convince a Verifier (e.g., a smart contract on a blockchain, or an auditor) that they have correctly executed their part of an EchoPulse session (e.g., contributed to an MPKX session) without revealing  $r_i$  or any other secret.

#### Witnesses (Secret Inputs to the Proof):

- $r_i$ : The participant's secret symbolic path.
- **Relevant internal states of  $G_t$  related to  $r_i$  traversal:** Specific nodes and edges traversed, internal HORM states.
- **Ephemeral secret values derived during traversal:** Such as internal components of hlocal outputs.

#### Public Inputs (Known to Prover and Verifier):

- $\mu(G_t)$ : The *mutated graph structure at time t*. Critically, we introduce  $\mu^*$  (mu-star) for ZKP efficiency.  $\mu^*$  is a compressed, public representation of the aspects of  $G_t$  relevant to the ZKP. Instead of providing the entire  $G_t$ ,  $\mu^*$  might include:
  - Merkle roots of relevant graph portions.
  - Commitments to specific node properties.

- A hash of the `mutation_schedule_id` and the `graph_id`.
- **entropy\_epoch (t):** The agreed-upon `forward_entropy_epoch` for the session.
- **transcript\_hash:** The cryptographic hash of the entire session's handshake transcript (from TLS or MPKX setup), which binds the session to the `entropy_epoch`.
- **v\_i\_start:** The initial vertex from which  $P_i$  started its traversal.
- **C\_i:** The public contribution (from MPKX, as defined in File 46) which is a function of  $v_{i,end}$  and aggregated HORM outputs.
- **H\_Anchor(P\_i):** The anchor hash that cryptographically binds  $P_i$ 's contribution to the derived graph state  $G_t$  and the HORM operations performed, as defined in File 47. This includes  $v_{i,end}$  and aggregated HORM outputs.

**zk-Proof Statement:** The Prover has computed  $C_i$  (and implicitly, contributed to the  $K_{Group}$  in MPKX) by traversing a valid, secret path  $r_i$  on  $G_t$ , where  $G_t$  was correctly derived from  $G_0$  and  $t$ , and  $t$  is consistent with the `transcript_hash`.

ZK Circuit Generation Logic:

The ZK circuit will encode the following computations:

1. **G\_t Derivation Verification:** Verify that  $\mu^*$  (or a more complete  $G_t$  representation if the ZKP scheme allows) was correctly derived from  $G_0$  and  $t$ . This might involve checking a hash of the mutation application.
2. **r\_i Traversal Simulation:** Simulate the HORM traversal of the secret  $r_i$  on the relevant portions of  $G_t$  (or  $\mu^*$ ). This involves encoding the HORM's internal logic (symbol lookups, node transitions, local hash computations).
3. **v\_i\_end Computation:** Verify that the end node of the traversal matches the input  $v_{i,end}$  implicitly contained in  $H\_Anchor(P_i)$ .
4. **H\_Anchor(P\_i) Computation:** Verify that the  $H\_Anchor(P_i)$  (public input) was correctly computed from `transcript_hash`, the derived  $v_{i,end}$ , and the `aggregated_HORM_outputs` resulting from the traversal.
5. **C\_i Computation (for MPKX):** Verify that the public contribution  $C_i$  was correctly derived from  $v_{i,end}$  and the aggregated HORM outputs according to the MPKX specification.

**Choice of ZKP Scheme:**

- **zk-SNARKs (e.g., Groth16, Plonk):** Offer very small proof sizes and fast verification, ideal for on-chain verification. However, they require a trusted setup (for some schemes) and can have higher prover computation costs. They are well-suited for proving fixed functions (like HORM steps) in a circuit.

- **zk-STARKs (e.g., StarkWare's Cairo):** Provide larger proof sizes but are transparent (no trusted setup) and scale better for larger computations. Prover costs can be higher, but verification is fast. Ideal for proving large computational integrity, like many mutation steps or long paths.

Symbol-Layer Compression:  $\mu \rightarrow \mu^*$  for ZK Circuit Size Reduction:

Full representation of  $G_t$  within a ZK circuit would be prohibitively large. Therefore, we use  $\mu^*$ :

- $\mu^*$  is a minimal, public commitment to the relevant aspects of  $G_t$  that affect the specific proof.
- For example, instead of the entire graph,  $\mu^*$  could be a Merkle tree commitment to the specific nodes and edges that are potentially traversed by  $r_i$ . The ZKP would then prove that  $r_i$  traverses a path *within* this committed subgraph.
- The circuit would prove that the  $\mu^*$  used for proof is consistent with  $G_0$  and  $t$ .

## 2.2. Proof-of-Consistency: $K_{\text{session}}$ Transcript Validity

While the ZKP does not directly prove  $K_{\text{session}}$  (which is secret), it proves its *derivation context*. By proving  $H_{\text{Anchor}}(P_i)$ 's correctness, we indirectly verify that  $P_i$  correctly computed its contribution to the group key, which implies  $P_i$  also knows the correct  $K_{\text{session}}$  (or  $K_{\text{Group}}$ ). This links the ZKP directly to the integrity of the key exchange.

## 2.3. Verifier Function and Public Proof Commitments

- **Verifier Function:** A smart contract function (e.g., `verifyZKP(proof, public_inputs)`) deployed on the blockchain. This function takes the ZKP and public inputs ( $\mu^*$ ,  $t$ , `transcript_hash`,  $C_i$ ,  $H_{\text{Anchor}}(P_i)$ , etc.) and returns true or false.
- **Public Proof Commitments (`LedgerWrite(P_i)`):**  $\text{\texttt{\$}\text{\texttt{\text{LedgerWrite}}}(P_i) = \text{\texttt{\$}\text{\texttt{\text{Transaction}}}(\text{\texttt{\$}\text{\texttt{\text{BlockchainAddress}}}, \text{\texttt{\$}\text{\texttt{\text{ZKP\_commit}}}(\text{\texttt{\$}\text{\texttt{\text{Proof}}}}, \text{\texttt{\$}\text{\texttt{\text{PublicInputs}}}}))\text{\texttt{\$}}$  Where `ZKP_commit` is the serialized ZKP, and `PublicInputs` includes  $t$ ,  $H_{\text{Anchor}}(P_i)$ ,  $C_i$ ,  $\mu^*$  (or its hash). This transaction is written to the blockchain, making the proof publicly verifiable.

## 3. Blockchain Interface Strategist: Blockchain Integration

The blockchain acts as a public, immutable ledger for verifying EchoPulse session legitimacy and for dispute resolution.

### 3.1. Writing ZKP Commitments to the Chain

- **API:** EchoPulse nodes (or a designated gateway for constrained devices) use a

blockchain client library (e.g., Web3.js for Ethereum, Hyperledger SDK) to interact with the blockchain.

- **Message Exchange:**

1. After an EchoPulse session (or MPKX) is completed and keys are derived, each participant (or a designated prover) computes their ZKP locally.
2. The participant then sends a transaction to the blockchain network.
3. The transaction calls the submitZKP function on a pre-deployed EchoPulseVerifier smart contract.
4. This function takes the ZKP and the public inputs as arguments.
5. The smart contract's verifyZKP function is invoked (or a separate off-chain trusted verifier service does this if the blockchain is only for commitment).
6. If verification passes, the contract records the hash of the ZKP and the public inputs, along with the submitting participant's ID and the entropy\_epoch (t).

- **Cost Efficiency:** For resource-constrained blockchains (like Ethereum L1), proofs are verified off-chain, and only the proof *hash* or a compact commitment is stored on-chain. Verification might happen on an L2 scaling solution or a dedicated ZK-rollup.

### 3.2. Replay Anchoring via Anchor\_BlockHash(t)

To enhance replay detection and provide an indisputable timestamp for session validity:

- During the derivation of the forward\_entropy\_epoch (t), the block hash of the blockchain at or just before the start of the EchoPulse session can be explicitly incorporated. 
$$t = \text{KDF-Extract}(\text{SALT}, \text{IKM} \parallel \text{Anchor\_BlockHash}(\text{t}_{\text{current\_block}}))$$
 Where  $\text{Anchor\_BlockHash}(\text{t}_{\text{current\_block}})$  is the hash of the latest block known at the time of session initiation.
- This means that t is tied to the state of the blockchain at that moment. A replayed session from a different t (and thus different Anchor\_BlockHash) would cause a mismatch during epoch validation, detectable by MPKX\_Repair().
- This provides immutable evidence that a session occurred at a specific time relative to the blockchain's history.

### 3.3. Optional Use of Smart Contract for Dispute Resolution

- **EchoPulseDisputeContract:** A separate smart contract can be deployed to handle disputes.
- **Dispute Trigger:** If an EchoPulse participant believes a session was

compromised, replayed, or a key derived incorrectly, they can submit a dispute transaction to this contract.

- **On-Chain Audit:** The contract can then trigger a public audit. This involves:
  - Retrieving the ZKP commitments (LedgerWrite transactions) related to the disputed session.
  - Calling the EchoPulseVerifier contract's verifyZKP function with the retrieved proof and public inputs.
  - Optionally, requiring other participants to submit their ZKPs for the same session.
- **Resolution:** Based on ZKP verification results and pre-defined rules, the EchoPulseDisputeContract can issue judgments (e.g., invalidate a session, penalize a malicious participant, or affirm legitimacy).

### 3.4. Time-Bounded Proofs: $ZK\_Proof(t_i)$ Validation

- ZKP submissions *MUST* be time-bounded. A  $ZK\_Proof(t_i)$  (a proof generated for epoch  $t_i$ ) must be validated on-chain within a specific  $\delta\_block$  window from the block when  $t_i$  was established (or when the session completed).
- This prevents stale proofs from being submitted and ensures timely dispute resolution. The EchoPulseVerifier smart contract includes this time constraint in its verification logic.

### Target Platforms:

- **Ethereum (L2s like Arbitrum, Optimism, zkSync):** Ideal for its large ecosystem and growing ZKP infrastructure. L2s provide scalability and lower transaction costs for ZKP submission.
- **Hyperledger Fabric/Besu:** Suitable for private/consortium blockchains where participants are known and audited. Provides permissioned access for ZKP submission and internal audit.
- **Custom L2/Dedicated ZK-Rollup:** For very high-throughput or extremely low-cost scenarios, a custom ZK-rollup could be built specifically for EchoPulse proofs.

## 4. Audit Layer Synchronization Expert: Audit Mechanism

The ZKP layer profoundly enhances the replay audit and entropy verification capabilities across the EchoPulse ecosystem.

### 4.1. $zk\_AuditContract(r_i, \mu_i(G_t), t)$

This conceptual contract (or a module within EchoPulseVerifier) formalizes the on-chain audit process.

- **Input:** The contract would receive public parameters like  $t$ ,  $P_i.ID$ , and potentially hashes of suspected  $r_i$  (not  $r_i$  itself) or  $C_i$ .
- **Functionality:** It doesn't store  $r_i$  or  $\mu_i(Gt)$  directly. Instead, it accesses submitted ZKPs. It defines a public interface for:
  - `queryProof( $t$ , participant_id)`: Retrieve a ZKP commitment for a specific participant and epoch.
  - `verifySessionIntegrity( $t$ , list_of_participant_IDs)`: Triggers a collective ZKP verification for all listed participants for a specific epoch.
  - `checkEntropyDrift( $t_{start}$ ,  $t_{end}$ )`: (Conceptual, requires advanced ZKP for statistical properties) A function to check if the overall system's entropy drift properties (as analyzed by File 2.1 Audit Fix) are maintained over a range of epochs by verifying an aggregate ZKP.

#### 4.2. Delay Guardrails

- **Proof Submission Deadline:** ZKPs for a session must be submitted within a specific block timeframe after the session's conclusion. This prevents participants from holding proofs indefinitely and selectively submitting them.
- **Dispute Period:** A specific window during which disputes can be raised after a session's ZKP commitment. This prevents retroactive invalidation far into the future.

#### 4.3. Public Verification Stub using only $\mu^*$ and Hash Anchor

To enable lightweight off-chain verification, or for auditors without full blockchain access:

- A `verify_stub(proof_hash, public_inputs)` function can be provided. This stub takes the hash of the ZKP (as stored on-chain) and the public inputs (like  $\mu^*$  and  $H_{Anchor}$ ).
- It performs a minimal verification using only the hashes and public commitments, leveraging the underlying ZKP's soundness property to imply the correctness of the full computation without needing the full ZKP or blockchain state. This is useful for rapid, first-pass integrity checks.

### 5. Integration with EchoPulse v3

The ZKP Bridge Layer integrates seamlessly with the existing EchoPulse v3 architecture, enhancing its capabilities without compromising core security.

- **Reused Variables:** All fundamental EchoPulse variables ( $\mu$ ,  $r_i$ ,  $t$ , `transcript_hash`, `KDF_HORM`, `v_enc`, `ReplayToken`, `entropy_epoch`) are consistently reused and are central to the ZKP circuits.
- **No Compromise of FS / MPKX Secrecy:** The ZKP design explicitly ensures that no secret information (e.g.,  $r_i$ , `Ksession`, internal graph states) is revealed in the proof. Only the *correctness of the computation* is proven. This maintains the PQ-FS and MPKX secrecy guarantees established in Files 43-46.
- **Optional Layer Design:** The `zkBlockChain()` functionality is designed as a *toggable and modular* layer. This allows deployments to choose whether to enable blockchain integration based on their specific security and audit requirements. This ensures flexibility for environments that do not require public verifiability or on-chain dispute resolution.
- **Compatibility with Side-Channel Hardened Outputs (File 48):** The ZKP circuit must be designed to prove the computations *after* side-channel hardening measures have been applied. This means the circuit verifies the logic including  $r_i$  rotation fuzzing, uniform KDF traversal, entropy masking, etc. This ensures that a correctly proven ZKP also implies a side-channel-resistant execution. For instance, the ZKP would confirm that the output  $C_i$  was derived from a correct (but secretly masked)  $r_i$  on  $G_t$  via the HORM, even if the runtime execution of the HORM was obfuscated for SCA defense.

## 6. Optional Extensions

- **Privacy-Preserving Traceability:** Beyond simple correctness, ZKP could be extended to allow participants to prove they belong to a specific group without revealing their identity, or to prove the sequence of network hops taken by a message without revealing the full path.
- **Delegated Proving:** For highly constrained IoT devices, the generation of complex ZKPs could be delegated to a more powerful, trusted "prover service" (e.g., an edge gateway). The device would send its minimal secret witness to the service, which then generates the ZKP. This introduces a trusted third party, but it can be beneficial for resource management.
- **Aggregated Proofs:** For MPKX with many participants, multiple individual ZKPs could be aggregated into a single, compact proof (e.g., using recursive SNARKs), reducing on-chain verification costs.

The EchoPulse Zero Knowledge Bridge Layer provides a powerful, auditable, and transparent extension to the EchoPulse security framework, enabling verifiable integrity for quantum-safe communications in a blockchain-enabled world.